Recurrent Neural Networks Meet Context-Free Grammar: Two Birds with One Stone

1st Hui Guan
Computer Science
University of Massachusetts, Amherst
Amherst, MA, USA
huiguan@cs.umass.edu

2nd Umang Chaudhary
Computer Science
University of Massachusetts, Amherst
Amherst, MA, USA
uchaudhary@umass.edu

3rd Yuanchao Xu
Computer Science
North Carolina State University
Raleigh, NC, USA
yxu47@ncsu.edu

4th Lin Ning
Computer Science
North Carolina State University
Raleigh, NC, USA
lning@ncsu.edu

5th Lijun Zhang
Computer Science
University of Massachusetts, Amherst
Amherst, MA, USA
lijunzhang@cs.umass.edu

6th Xipeng Shen
Computer Science
North Carolina State University
Raleigh, NC, USA
xshen5@ncsu.edu

***Abstract*—Recurrent Neural Networks (RNN) are widely used for various prediction tasks on sequences such as text, speed signals, program traces, and system logs. Due to RNNs' inherently sequential behavior, one key challenge for the effective adoption of RNNs is to reduce the time spent on RNN inference and to increase the scope of a prediction. This work introduces *CFG-guided compressed learning*, an approach that creatively integrates Context-Free Grammar (CFG) and online tokenization into RNN learning and inference for streaming inputs. Through a hierarchical compression algorithm, it compresses an input sequence to a CFG and makes predictions based on the compressed sequence. Its algorithm design employs a set of techniques to overcome the issues from the myopic nature of online tokenization, the tension between inference accuracy and compression rate, and other complexities. Experiments on 16 real-world sequences of various types validate that the proposed *compressed learning* can successfully recognize and leverage repetitive patterns in input sequences, and effectively translate them into dramatic (1-1762×) inference speedups as well as much (1-7830×) expanded prediction scope, while keeping the inference accuracy satisfactory.**

***Index Terms*—recurrent neural networks, data compression, context free grammar, tokenization**

I. Introduction

Recurrent Neural Network (RNN) [1] is very effective in modeling and predicting temporal sequences. It has been successfully applied to a broad range of machine learning tasks. Because of RNN's high prediction accuracy, there is also an increasing interest in applying RNNs for sequence prediction tasks in other domains such as program analysis [2], data prefetching and cache placement in computer architecture [3], [4], memory management [5], network caching policy design [6], system log analysis [7], and recommender systems [8]. As tasks in these domains have real-time or near real-time requirements, speeding up RNN inference is an important problem.

Due to RNNs' inherently sequential behavior, reducing the time spent on RNN inference is a challenging problem. Although many efforts have been taken to accelerate RNN inference, for example by designing efficient model architectures [9], model compression [10], sparsification [11], and many other approximations [12], the demands for higher speed remain as the application domains and data volume for RNN keep expanding dramatically. Our study showed that a 1-layer RNN model takes milliseconds to predict the next event on GPUs while prediction tasks in computer systems such as data prefetching and cache replacement typically need results in nanoseconds. The performance issue becomes worse when larger models are used to achieve higher accuracy.

Moreover, demands for long-term large-scope predictions are increasingly popular for RNN. Rather than predicting only the next event, many uses of RNN desire predictions of the next N ($N > 1$) events so that they can start the preparations or take actions earlier. That is especially important if the response (e.g., prefetching or system migration) takes time. There are some attempts to enable large-scope predictions [13]–[15], but they are mostly from the traditional angle, trying to adjust the RNN model architecture or hyperparameters. The prior efforts in pursuing the two important objectives of RNN inferences, improving its speed and scope, have been largely going separately. Large room for improvement remains in both.

In this paper, we present *CFG-guided compressed learning*, a novel method that, by integrating CFG and online tokenization into RNN inference, simultaneously improves the state-of-the-art on both objectives significantly. Unlike popular Deep Neural Network (DNN) compression which compresses *DNN models*, CFG-guided compressed learning compresses *input data sequences*. It is applicable to sequences that consist of many repeated subsequences. For instance, data from sensors in a factory may show similar patterns along time; system logs can have the same event sequences due to repeated operations; the execution traces of a program often manifest repeated patterns. The *basic rationale* is to compress

the data sequence by automatically identifying and reducing the repeated subsequences to an abstract format (i.e., a non-terminal symbol in CFG). If the learner can directly learn and make predictions on the compressed sequence, it may benefit from the identified repetitions in both inference speed and prediction scope.

There are three research questions (RQ) for realizing the idea effectively:

- **RQ1**: How to compress a sequence to keep its statistical properties such that RNNs can still learn patterns from the compressed sequence?
- **RQ2**: How to conduct inference on an online generated data sequence (that is not compressed) given that the model is trained on the compressed sequences?
- **RQ3**: How to support continual model refinement in an online fashion?

When answering the questions, it is important to note **three principles**. (i) *Domain independent.* The solution should work across domains, which is essential for its general applicability for data sequences with repetitive patterns. It is, for instance, possible to use the knowledge of program code structure to compress program traces [16], but such a method cannot apply to sensor data, health data, or data in many other domains, and hence does not fit the need. (ii) *Beneficial.* The overhead of the solution should not outweigh its benefits. (iii) *Staying accurate.* Users' satisfaction on model accuracy should not be the price for the improvement of inference speed and scope.

This paper presents the first known solution to these open questions by proposing *CFG-guided compressed learning*. It uses no domain knowledge and hence stays completely domain-independent. It learns from compressed sequences and predicts, at one time, not one single event but a sequence of events, achieving both large speedups and also large prediction scopes. It, meanwhile, offers an easy-to-use knob allowing users to keep model accuracy at a satisfying level while enjoying the speed and scope benefits.

CFG-guided compressed learning achieves these by introducing CFG and online tokenization into RNN inference. Specifically, it answers **RQ1** by employing CFG to compactly represent the input data sequence while keeping it in a form amenable for RNN-based learning. It does it by building on an existing linear-time hierarchical compression algorithm, Sequitur [17]. Both RNN training and inference can operate on the CFG representation smoothly. It answers **RQ2** by enabling on-the-fly *incremental compression* via online tokenization as new events arrive and, if necessary, calls the RNN model to make predictions based on the tokenized event sequence. Each prediction is a token in the dictionary, which can be a terminal (one single upcoming event) or a non-terminal (a sequence of upcoming events). It answers **RQ3** through *continuous compression-based refinement* which refines the RNN model on the compressed sequence continuously and efficiently. Section III explains the algorithm in detail.

Section IV reports experiments on 16 real-world data sequences including program function calls, memory traces, and system logs. The results show that compared to RNN predictors, compressed learning achieves 1-1762$\times$ prediction speedups for its large prediction scopes (up to 7830 events per prediction). For the same prediction scope, compressed learning gives as much as 54% higher prediction accuracy than default RNN predictions.

Overall, this work makes the following main contributions:

- To the best of our knowledge, this is the first work integrating CFG and sequence compression into RNN for both faster prediction and larger prediction scope on streaming inputs.
- It proposes CFG-guided compressed learning as a novel learning paradigm for RNN-based sequence modeling. The algorithm is applicable to domains whose data sequences have repetitive patterns.
- It overcomes the issues caused by the myopic nature of online tokenization through *efficient rollback*, addresses the tension between compression rate and inference accuracy through *accuracy-conscious lowering*, and minimizes runtime overhead through *partial compression*.
- It analytically studies the efficiency benefits of *compressed learning*, and empirically validates its benefits in improving both the prediction scope and the inference speed of RNN.

II. Background

Context-Free Grammar (CFG) is a formal grammar that consists of production rules. Each production rule is of the form $A \rightarrow \beta$, where A is a single *non-terminal* symbol and β is a sequence of terminals and non-terminals. It is context-free because the left-hand side (LHS) of the production rule can always be replaced with the right-hand side (RHS) regardless of the context of the LHS (non-terminal). To compress a data sequence, one can transform the sequence into a CFG. The process is called *grammar-based compression*.

Sequitur is a classic grammar-based compression algorithm, originally proposed by Nevill-Manning and Witten [17]. It infers a hierarchical structure from a sequence of discrete symbols in linear time. These discrete symbols are treated as terminal symbols. For a given sequence of symbols, it derives a CFG where each rule reduces a repeatedly appearing sequence of terminals into a non-terminal symbol. For example, if the sequence is "abcab", the algorithm produces CFG: `S -> AcA, A -> ab`. By substituting repeating strings (e.g., "ab") with new non-terminal symbols (e.g., "A"), it produces a concise representation of the input sequence (e.g., "AcA").

We have not seen its use in RNN. This work selects Sequitur as the compression algorithm for the fit of the compression results for RNN and its domain-independent property. Some similar grammar-based compression algorithms [18]–[20] could be used as well.

III. Compressed Learning Algorithm

Compressed learning learns from compressed sequences either offline or online, and predicts not one single event but a sequence of events. It builds on the grammar-based compression algorithm Sequitur, which (incrementally) compresses a

(a)

Stage 1: Offline compression of a sequence:

Sequence:
x abc abc abc abc d abc abc abc abc d abc abc abc abc d abc abc abc abc d ae ae ae ae fabcg abc abc

Compression result (CFG) from Sequitur:

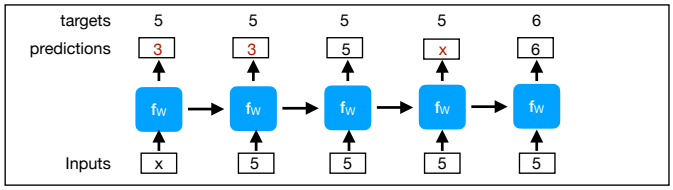
0 → x 1 1 2 2 f 3 g 4	
1 → 5 5	abcabcabcabcdabcabcabcabcd
2 → 6 6	aeae
3 → a b c	abc
4 → 3 3	abcabc
5 → 4 4 d	abcabcabcabcd
6 → a e	ae

Compressed sequence after lowering:
x 5 5 5 5 6 6 6 6 f 3 g 3 3

Dictionary:
3: abc
5: abcabcabcabcd
6: ae
and terminals a,b,c,d,e,f,g,x,y

(b)

Stage 2: Offline RNN training using compressed sequence:



(c)

Stage 3: Online inference (and training) on a test sequence:

Time	Event	Tokenized Seq C	Input to RNN	Prediction
0	S [START]	⬥S	S	x
1	x	S⬥x	x	5 (abcabcabcabcd)
2	a	Sx⬥a		
3	b	Sx⬥ab		
4	c	Sx⬥3		
5	a	Sx⬥3a		
6	b	Sx⬥3ab		
7	**a**	Sx⬥3aba	3aba	b
8	b	Sx3aba⬥b	b	a
9	**d**	Sx3abab⬥d	d	a
10	a	Sx3ababd⬥a	a	b
11	**y**	Sx3ababda⬥y	y	a
12	a	Sx3ababday⬥a	a	b
13	b	Sx3ababdaya⬥b	b	a
14	**f**	Sx3ababdayab⬥f	f	3 (abc)
15	a	Sx3ababdayabf⬥a		
16	b	Sx3ababdayabf⬥ab		
17	c	Sx3ababdayabf⬥3	3	g
18	g	Sx3**ababdayabf3⬥g**	g	3 (abc)
Refine Model (optional)	- Compress the sequence "x⬥ababdayabf~g", - Use the compressed result "x377day7fg" to refine (train) model M, - Update dictionary with a new terminal symbol {7->ab}.			
19	a			
...	... ...			

Fig. 1. Running example of compressed learning: (a) offline compression of a train sequence, (b) offline RNN training using compressed sequence with input length of 5 and batch size of 1, and (c) online inference and optional model updates on a test sequence. In (c), the diamond symbol ⬥ represents "C.cursor" in Figure 2, which indicates the start of the subsequence in C that has not been sent to the predictive model M. Cells in the "Prediction" column are filled in red if the predictions are wrong.

sequence into a CFG. The learning and inference operate on a variant of the CFG. Compressed learning has three stages: (1) offline compression of the training sequences to build the vocabulary and compressed sequences, (2) offline RNN training using compressed sequences, and (3) online RNN inference and optional online model refinements. The three stages are compatible with the typical workflow for RNN-based application development.

In this section, we first introduce a running example to illustrate the three stages, and then explain the major complexities and the general algorithm.

A. Running Example

We use the example in Figure 1 to convey the intuitions of compressed learning. Figure 1(a) illustrates the first stage– that is, how the grammar-based compression works on a sequence. The data sequence is compressed by Sequitur into the CFG shown in Figure 1(a), consisting of seven rules. Each rule shows what sequence (on the RHS of the rule) a non-terminal symbol (LHS) represents. Rule 0 represents the entire sequence, and each of the other rules represents a subsequence in the entire sequence. For reference, we expand the RHS of each rule (except Rule 0) and put the results to the right of the CFG rules. Below the CFG is a lowered representation of the RHS of Rule 0 after some symbols in the RHS are expanded. The expansion process is called *lowering*, which makes the compressed sequence into a form friendly for RNN training. Details will be explained in Section III-C. Below the lowered sequence is a dictionary, which records all the non-terminal symbols that appear in the lowered sequence, along with what subsequence each of them represents. The dictionary, by default, includes all the terminal symbols in the train sequence.

Figure 1(b) shows the offline RNN training in compressed learning. The lowered sequence is used as the input to train the RNN-based predictive model.

Figure 1(c) shows the inference steps for a test sequence. After the inference starts, at time 1, the arriving event `x` triggers a prediction by the trained RNN. The prediction gives out a token `5`, which corresponds to a subsequence of events `abcabcabcabcd` represented by the RHS of rule 5 in Figure 1(a). The events coming in the next 5 time steps (time 2–6) are all consistent with that prediction, and hence the RNN does not need to be invoked to make predictions at those times. Note that as those events arrive, our compressed learning algorithm automatically *tokenizes* them, as illustrated by the replacement of `abc` with token `3` at time 4. Formally, given an event sequence s and a vocabulary V, one can replace subsequences of events in s with the corresponding non-terminal symbols in V. The process is called *tokenization* and the resulting sequence is called *tokenized sequence*. A compressed sequence is also a tokenized sequence.

At time 7, the actual event `a` differs from the prediction (`c`), which prompts the RNN to discard the rest of its prediction, and make another prediction based on the subsequence that has not yet been fed to the RNN—that is, the subsequence

following ♦ in the "Tokenized Seq C" column in Figure 1(c), `3aba` in this case. The prediction is a single event `b`, which matches the actual event at time 8. At this time, because no prediction is there for the next time step, the RNN is triggered to make another prediction. This process continues.

At time 14, the RNN sees `f` and generates another subsequence (`abc` represented by token `3`) as the prediction, which proves to be correct. After time 18, one may want to refine the model based on the newly collected event sequence. Compressed learning starts an online compression on the uncompressed event subsequences in Tokenized Seq C and has the RNN learn from the compressed results to get refined, illustrated by the "Refine Model (optional)" row in Figure 1(c). The process then continues.

B. Issues for Algorithm Design

To make the compressed learning algorithm work in general cases, we must address several issues.

Issue-1: Myopic nature of online tokenization. Tokenization is short-sighted. Consider a simple example that has a dictionary with only two entries:

T1: ab T2: abc

For an input sequence `abcabc`, suppose that the RNN predicts `T1` at the starting point. As the tokenizer sees the first two events `ab`, it tokenizes them into token `T1`, and feeds it to the RNN. The RNN would then update its hidden state and make a prediction of the next token, say another `T1`. But when the third event `c` arrives, the tokenizer may realize that the first two events `ab` are actually part of a larger token `T2` (for `abc`). So for the RNN to make predictions based on `T2`, the compressed learning must be able to deal with the premature tokenizations and allow the RNN to undo its state changes when necessary. Such an issue may appear whenever some tokens in a directory are the prefixes of other tokens (e.g., tokens 3 and 5 in the example in Figure 1(a)). This issue entails further questions on the influence of the rollbacks on the performance of compressed learning. Would they incur extra invocations of the RNN? Would they cancel out the time savings from compressed learning?

Issue-2: Runtime compression overhead. To refine the RNN at runtime, online compression is needed to generate the compressed sequences so that they can be used to continuously train the RNN model on the fly. Although the refinement is only optional, it is still necessary to minimize the runtime overhead in online compression to maximize the performance benefits of compression learning.

Issue-3: Traps of large tokens. Although a large token could help enable large-scope predictions for its representation of a long subsequence, it could also form a trap. It is because large tokens tend to appear less frequently in the compressed sequence, which makes it harder for RNN to learn about the patterns in the compressed sequence. In Figure 1(a), for instance, the top-level tokens in the RHS of rule 0 each occur at most twice, but in the lowered sequence in the same figure, some tokens (e.g., 5, 6) appear twice as often. The compressed learning hence must be able to deal with the tradeoff between token granularities and frequency.

C. Algorithm

In this part, we present the full algorithm of compression learning while highlighting how the three main issues are addressed in the design.

The first two stages in compressed learning produce a vocabulary V (illustrated in Figure 1(a) as a dictionary) and an offline-trained RNN model M (illustrated in Figure 1(b)). The vocabulary V contains both non-terminal symbols V^N in compressed sequences and all the terminal symbols (i.e., unique events) V^E in uncompressed sequences, i.e., $V = V^N \cup V^E$. Each non-terminal symbol represents a subsequence of events, $v^N = v_1^E, \cdots, v_{|v^N|}^E$, where $v_i^E \in V^E$. For the description purpose, we simplify the notation of an event v^E to e, and refer to both $v^N \in V^N$ and $e \in V^E$ as a *token*. As the first two stages are straightforward applications of the Sequitur compression and standard RNN training, we focus our discussion on the third stage.

Problem definition. The problem of the online prediction is that, given an already emitted sequence of events $s = e^1, \cdots, e^t$, our trained model M shall be able to predict the upcoming events $v_{t+1}^* = e^{t+1}, \cdots, e^{t+|v_{t+1}^*|} \in V$ such that:

$$v_{t+1}^* = \arg\max_v Pr(e^1, \cdots, e^t, v|M), \tag{III.1}$$

where $Pr(.|M)$ calculates the probability of the occurrence of a sequence given model M. Here, v_{t+1}^* can be either a non-terminal symbol that represents a sequence of events or a terminal symbol that represents a single event.

Algorithm description. Figure 2 outlines the online inference and model refinement algorithm of compressed learning. Specifically, at a newly emitted event e, the following happens.

1) Tokenization: The algorithm (line 24 in Figure 2) tokenizes e in the context of the earlier events. For a given sequence, a finite state machine (F in Figure 2 line 15) tries to find a token in the vocabulary, the content of which matches with the given sequence. The tokenization subroutine appends the recognized token to the end of the tokenized sequence C; sometimes its old suffix may need to be replaced because a longer match is found. The tokenization algorithm is shown in Figure 3. It appends the recognized token (line 5) to the end of the tokenized sequence C (line 15). Sometimes C's old suffix may need to be replaced because a longer match is found, triggering the rollback process in lines 9-12.

Rollback (solution to *Issue-1*). The tokenizer helps track the starting point (C.cursor) of the part of C that has not yet been fed into the predictive model M. The replacement of C's suffix (the part following ♦) in tokenization could necessitate the update of the cursor. If the current position of the cursor is in the suffix replaced by the new token, the cursor is updated to the position right before the new token. To make M be able to overcome the premature tokenization and conduct predictions based on the new token, compressed learning records the recent hidden states of M in memory so

```
1.  // Predict and learn with compression
2.  Input:
3.    P: trace generator
4.    V: initial vocabulary
5.    M: initial predictive model
6.  Output:
7.    M: updated predictive model
8.    V: updated vocabulary
9.  Constants:
10.   START, EOF: markers of the start and end of input
11.   L: length of a learning interval
12.   FREQ: the minimum frequency for a word to get into the vocabulary
13.
14. // create a tokenizer F to recognize the token in V
15. F = tokenizerCreation (V)
16. n = 0  // count the number of events
17. C = emptyList  // store the tokenized sequence
18. i = 0
19. v = M.predict( START )  // predict the upcoming subsequence of events
20. C.cursor = 1 // track the end of the part of C that has been used by M
21. while (e = P.generate () != EOF) { // a new event is produced
22.    n ++
23.    // recognize the new token and update C, M
24.    Tokenize (F, e, C, M)
25.   // if e doesn't match the predicted or the prediction is exhausted
26.    if (!matches(e, v[i]) || i == v.len-1 ) {
27.      // predict the next token (i.e., a subsequence)
28.      v = M.predict ( C[C.cursor : C.len] )
29.      C.cursor = C.len
30.      i=0
31.    }
32.    else { // no prediction needed
33.      i ++
34.    }
35.    if (n == L){
36.      // compress the tokenized sequence seen so far,
37.      // update tokenzied sequence and return new tokens V_
38.      V_ = PartialCompress(C)
39.      M.train (C) // update the predictive model M
40.      F.update (V_) // update the tokenizer with the new words
41.      V.append (V_)  // update the vocabulary
42.      C = []
43.      n = 0
44.   }
45. }
```

Fig. 2. Algorithm of compressed learning for online inference and optional model refinement.

that M can easily *rollback* its hidden state to the state it had at the new cursor position.

2) Prediction when necessary: After getting the new token, the algorithm (line 26 in Figure 2) checks whether it is time to make a prediction. There are two cases when a new prediction happens: (a) the predicted event for this time point does not match the newly arrived event, which indicates a prediction error; (b) the predicted sequence ends at this time point. In other cases where the prediction is correct so far and the next event is already covered by the recent prediction, there is no need to make a new prediction.

3) Model refinement: Compressed learning supports continuous model refinement. After a certain interval, the algorithm refines the predictive model with the compressed sequence of that interval, as lines 35 to 44 in Figure 2 shows.

Partial compression (solution to Issue-2). The learning starts with compressing the new subsequences in C. A basic design is to run Sequitur on the entire sequence C. But as the tokenizations already compress some parts of the sequence,

```
1.  // Recognize the new token that has e as the final element
2.  // and update tokenized sequence C with the new token
3.  Function Tokenize (F, e, C, M){
4.    lastToken = C.getLast(); // get the last token in C
5.    newToken = F.recognize(e);  // recognize the token
6.    if (newToken.len > 1) { // cover more than a single event
7.       // replace the ending tokens in C with newToken
8.       if (C.cursor > C.len-1) { // rollback is needed
9.          // rollback hidden state of M and update cursor in C
10.         M.rollBack(C.cursor-C.len)
11.         C.cursor=C.len-1
12.      }
13.   }
14.   else {
15.      C.append(newToken)
16.   }
```

Fig. 3. Algorithm of tokenization for online inference.

subroutine *PartialCompress* (line 38) compresses only the uncompressed parts which could save compression time. The subroutine first extracts out all the new subsequences in C that do not match non-terminal tokens. In Figure 1(c), there are three such subsequences, "x", "ababdayabf", "g" ("S" is the start marker, hence not included). Rather than running Sequitur on each of them, our design is to concatenate them together such that one run of Sequitur would suffice. It is important to notice that simple concatenation can cause wrong compression results, as the subsequences are not actually consecutive but Sequitur could be misled by the concatenated sequence to group the end of a subsequence and the start of another subsequence into one token. To avoid the issue, we insert distinctive symbols at the end of a subsequence as separators, as illustrated by the triangles in the "Refine Model" row in Figure 1(c).

Accuracy-Conscious Lowering (solution to Issue-3). Lowering is an important step for striking a good tradeoff between token granularity and frequency. It, from the CFG, derives a compressed sequence friendly to RNN training (both offline and online). It recursively conducts a depth-first expansion of tokens in an input compressed sequence (s). If a token's frequency is no smaller than a threshold ($FREQ$), the subroutine stops expanding it, and puts it into the vocabulary as a valid token. Such a design avoids unnecessary expansions to keep the sequence as compact as possible while meeting the frequency requirement.

The frequency threshold ($FREQ$) offers a knob to adjust the tradeoff between the compression rate and the frequency of tokens. In the extreme case where the frequency threshold is too large, there will be no non-terminal symbols and our algorithm becomes the same as default learning. In another extreme case where the frequency threshold is too small, the data sequence will be compressed into a very abstract format that contains mainly less-frequent non-terminal symbols. Although a highly compressed sequence may lead to less frequent hidden state updates and yield larger speedups and prediction scopes, it is not friendly for learning and could result in accuracy degradation. During offline training, compressed learning uses binary search to automatically find the suitable frequency

threshold that meets a user-specified accuracy requirement, as detailed in Section IV.

D. Algorithm Analysis

In this part, we analyze the computational complexity of compressed learning, and how rollbacks affect the efficiency. The total time cost of predictions in compressed learning is $\alpha \cdot \gamma$ fraction of the cost of the default RNN inference on the original sequence, where $\alpha = N_c/N_d$ and $\gamma = T_c/T_d$, N_c, N_d are the number of predictions conducted in compressed learning and in default RNN inference, and T_c, T_d are the average times taken by one prediction in the two cases. The time taken for one prediction is essentially the time to execute recurrence function f_W for updating the hidden state and producing outputs.

We claim that $\alpha \cdot \gamma$ must be no greater than one, even in the presence of rollbacks. Formally, we have the following propositions.

Proposition 1: Compressed learning, even with rollbacks, does no more predictions than the number of input events– that is, $\alpha <= 1$.

Proof: The correctness is easy to see if we notice that as Figure 2 shows, a rollback does not directly trigger a prediction. Predictions are triggered only on line 27 of the algorithm, which is executed at most once for a new event. □

Proposition 2: The total number of inputs to the predictive model in compressed learning is no greater than the number of input events– that is, $\gamma <= 1$.

Proof: The only time when the cursor moves backward is at a rollback time. Notice that the movement is to put the cursor right before the new token which is the token at end of the compressed input. In effect, it just adds one token (i.e., the new token) into the set of inputs possibly sent to the predictive model, as a response to the advent of the new event that triggers the rollback. Therefore, even with rollbacks, the number of inputs to the predictive model is no more than the number of input events. □

As the default run of RNN takes in each input event and makes one prediction per event, the two propositions entail that even in the worst case, neither α nor γ would be greater than one. In practice, because in compressed learning, predictions are invoked only at some events (line 26 in Figure 2) and the input to the model is compressed, $\alpha \cdot \gamma$ is typically much smaller than one, leading to a better prediction efficiency.

For the same reasons, the refinement of the predictive model in compressed learning also costs no more than the default given the same number of training epochs.

Compressed learning adds extra operations. They are primarily tokenization, the recording of hidden states (for possible future rollbacks), and compression if optional model refinement is enabled. Both tokenization and compression have a linear time complexity in terms of the length of the input sequence. As inference involves a number of matrix multiplications, the times these extra operations take are marginal relative to the inference time savings, as Section IV will show. Recording hidden states in memory does not take time as they are already created in memory, but consumes space. The number of hidden states needed to record is bounded by $m = min(L, K)$, where L is the length of a learning interval, and K is the length of the longest token in the vocabulary. If m exceeds memory budget, the implementation can limit it to fit the memory budget, and accordingly, limit rollbacks to tokens shorter than m.

TABLE I
SEQUENCE STATISTICS. (EVERY SEQUENCE CONTAINS 500K EVENTS.)

Sequences		Compression	#non-terminal	token length stats		
No.	Name	ratio (X)	symbols	min	mean	max
1	fluid-calls	3759.4	6	4	1507.3	8192
2	go-calls	12.2	436	2	14.3	80
3	molecule-calls	96.0	155	2	78.2	1024
4	perl-calls	79.8	116	2	88.4	1880
5	ocean-calls	747.4	27	2	293.7	2194
6	waves-calls	2066.1	16	2	1051.1	8192
7	fluid-mem	2487.6	8	2	1128.9	5120
8	go-mem	86.2	30	2	339.8	3216
9	molecule-mem	4.3	980	2	10.6	85
10	ocean-mem	5.0	916	2	11.7	71
11	perl-mem	13.4	216	2	36.0	577
12	waves-mem	3.5	29	2	16.4	88
13	windows-log1	28.7	213	2	53.3	914
14	windows-log2	29.7	269	2	34.3	469
15	thunderbird-log1	17.0	403	2	22.1	2048
16	thunderbird-log2	20.9	428	2	17.1	1536

* The frequency threshold in the lowering step is set to 5 when the reported statistics are collected. *token length stats* consider only non-terminals in the compressed seq; a token is a sequence of events. *X-calls*: function call seq.; *X-mem*: memory address traces; *X-log*: system logs.

TABLE II
THE APPLICATIONS AND SYSTEMS ON WHICH TRACES ARE OBTAINED.

Name	KLOC	Description
fluid [21]	1	A dynamic fluid simulation algorithm using Lattice Boltzmann Method.
go [22]	21	A go playing engine using Monte Carlo tree search.
molecule [23]	24	A molecular modeling application.
ocean [24]	210	A regional ocean modeling algorithm.
perl [25]	362	Perl interpreter.
wave [26]	1	A 3D wave modeling algorithm.
windows [27]	N/A	Windows 7 event log that keeps track of package installation and updates.
thunderbird [27]	N/A	Thunderbird supercomputer log that contains alert and non-alert messages. Used for alert detection and prediction research [28].

IV. EVALUATION

We conducted a set of experiments to examine the efficacy of the proposed technique, trying to answer the following questions: (1) How much benefit can we get from compressed learning for inference speed and prediction scope? (2) How does compressed learning affect the model quality? (3) What is the runtime overhead of incremental tokenization for online inference and the partial compression for online model refinement?

A. Methodology

Datasets. When collecting traces for the experiments, in order to get a comprehensive assessment of the technique, we try to ensure that the traces (i) come from the real-world workloads or systems; (ii) exhibit a spectrum of regularities; (iii) cover several different types of events and domains.

Table I lists the sixteen traces we experiment with. They are of three types: The first six are function call sequences, the second six are memory address traces (in 64-byte data blocks), and the final four are system log traces. Prediction on these sequences can help guide just-in-time optimizations [29], prefetching [30]–[32], and system anomaly detection [33].

The system log traces come from LogHub [27]; they are real-world system traces of Microsoft Windows OS and Thunderbird Linux Cluster from Sandia National Lab. Following prior works in log parsing [7], [27], we replaced date, timestamp, package specs, and parameter values in each log entry as a dummy string to convert the unstructured free-text log entries into a sequence of log keys. Each event is a log key, which is also known as message type. The other traces were collected through Intel instrumentation tool (Pin [34]) on six real-world applications. Table II lists those applications and the sources. These applications are from various domains, from fluid dynamics to programming language interpreters and stochastic modeling. The regularity of the behaviors of those applications also varies significantly. Program `fluid`, for instance, is very regular; the core of it is structured linear algebra. Program `go`, on the other hand, as a stochastic tree search through Monte Carlo (random walk), is inherently random. Such a collection allows the experiments to check whether the compressed learning can discover and effectively leverage the repetitive patterns in a trace, and at the same time, avoid negative impact (slowdown, accuracy loss) if it is applied to trace the lack of such regular patterns.

Table I also shows the sequence statistics. Each sequence contains 500,000 events. The interval size is 50,000, so one sequence consists of a total of 10 intervals. For each sequence, we use the first five intervals for model training and the rest for model testing (e.g., online prediction). Continuous model refinement is disabled by default. If it is enabled, the learning on a subsequence happens after the prediction on that subsequence is done.

Counterparts for comparisons. Since CFG-guided compressed learning is generally applicable to domains whose sequences have repetitive patterns, we use standard RNN-based sequence modeling used in these domains as our baselines. Specifically, we compare our compressed learning (denoted as *ours*) with the following two default approaches.

(1) *Default learning with 1-event prediction (default-1).* This method trains the RNN using the *un-compressed sequence* and predicts only the next single event at one prediction. The number of predictions it has to make is the same as the number of events in a test sequence. All the prior works on applying RNNs to program trace analysis [3]–[5], [35] and system log analysis [7] use this strategy.

(2) *Default learning with k-event prediction (default-k).* This method also trains the RNN using the *un-compressed sequence* but has the same prediction scope as our *compressed learning* has. That is, after it predicts an event, it feeds the prediction to the RNN to make another prediction, and continues doing that until the next k events are predicted, where k is the average length of a prediction in our *compressed learning*. So unlike our compressed learning, *default-k* predicts the next k events by making k consecutive predictions rather than one prediction; it hence saves no prediction time.

Models. The RNN model used in the experiments of all the methods is the same. It consists of an embedding layer with an embedding dimension of 256, a GRU layer with 1024 units, and a fully-connected output layer. We train an RNN model for each sequence. For offline training, the RNN models are trained with ADAM [36] using an input length of 100 for all methods. If online training is enabled, the models are refined for one epoch on each interval (i.e., 50,000-length event sequence) with an input length of 100.

Hyperparameters. Compared to default RNN training, the only extra hyperparameter introduced by compressed learning is the frequency threshold ($FREQ$) used in the lowering step. We used binary search to determine the best $FREQ$ that meets a user-specified accuracy requirement while achieving good inference speedups. Specifically, we allow users to specify a tolerable accuracy drop (e.g., 1%) compared to *default-1*. We use the first 4 training intervals for RNN training and the remaining 1 training interval for validation. We increase $FREQ$ to lower the compression rate in exchange for better model quality or decrease it for a higher speedup. To reduce the overhead of binary search, the options of $FREQ$ are currently limited to 14 values: 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, and 50000. When $FREQ$ reaches 50,000 which is the interval size, the sequence is not compressed and compressed learning falls back to default-1 so that the same accuracy is guaranteed. If one changes the interval length, the list of options can be adjusted accordingly.

Metrics. Our evaluation uses the following three metrics. (i) The *speedup* over the inference time (i.e., averaged time spent on predicting the next event) taken by *default-1* when all runtime overhead (e.g., tokenization, rollback) is counted in. (ii) The *prediction scope*, which is the average length of a prediction. (iii) The *prediction accuracy*, which is the ratio between the # of correctly predicted events over the total number of events.

Platform and hardware. All the experiments are performed with TensorFlow 2.2 on computing nodes equipped with a 12-core 2.40GHz Intel Xeon E5-2620 v3 processor, 256GB of RAM, 256GB SSD and 4 NVIDIA TITAN X GPUs. For each experiment a memory limit of 60GB and a limit of 1 GPU usage was set. CUDA version is 10.1. Source code is available anonymously: shorturl.at/imuDR.

TABLE III
ONLINE PREDICTION RESULTS OF COMPRESSED LEARNING AND ITS COMPARISON WITH DEFAULT APPROACHES THAT USE UNCOMPRESSED SEQUENCES.

Sequences	spec. acc. drop	$FREQ$	avg. pred length	#predictions	#rollbacks	tokenization overhead (%)	avg. latency* (ms)		prediction speedup (×)	event accuracy (%)		
							ours	default-1		ours	default-k	default-1
fluid-calls	0%	2	7830	35	0	0.20680	0.036	4.340	120.9	99.9984	99.9984	99.9984
	1%	2	7830	35	0	0.20680	0.036		120.9	99.9984	99.9984	
perl-calls	0%	50	31	8916	522	0.00017	0.132	3.701	28.1	99.74	98.54	99.89
	1%	5	133	7372	1180	0.00018	0.082		45.3	99.57	98.59	
molecule-calls	0%	50	30	8572	722	0.00044	0.179	3.582	20.0	99.67	95.86	99.6
	1%	5	91	4393	1099	0.00287	0.071		50.5	99.5	77.37	
ocean-calls	0%	500	17	14677	191	0.00007	0.209	3.726	17.9	99.89	99.51	99.95
	1%	5	200	5136	725	0.00282	0.084		44.5	98.94	44.59	
wave-calls	0%	20	4798	4070	21	0.00616	0.246	3.707	15.1	83.74	83.76	83.76
	1%	20	4798	4070	21	0.00616	0.246		15.1	83.74	83.76	
go-calls	0%	20000	1	248732	2132	0.00001	3.608	3.608	1	87.4	87.59	87.59
	1%	5000	2	229512	3767	0.00001	3.280		1.1	86.68	70.47	
fluid-mem	0%	5	2500	100	0	0.05676	0.002	3.551	1762	99.96	89.53	99.97
	1%	5	2500	100	0	0.05676	0.002		1762	99.96	89.53	
go-mem	0%	5	78	3336	55	0.00001	0.054	3.611	66.4	98.82	91.49	99.04
	1%	5	78	3336	55	0.00002	0.054		66.4	98.82	91.49	
perl-mem	0%	20	81	6273	105	0.00014	0.086	3.630	42.2	99.48	98.52	98.74
	1%	20	81	6273	105	0.00014	0.086		42.2	99.48	98.52	
ocean-mem	0%	20	4	83383	3640	0.00002	1.478	3.954	2.7	81.63	75.93	81.3
	1%	5	6	78782	1166	0.00004	1.236		3.2	80.69	72.37	
wave-mem	0%	500	2	193050	1902	0.00001	3.084	3.700	1.2	79.32	64.71	79.56
	1%	5	5	85957	1416	0.00001	1.233		3	79.1	50.71	
molecule-mem	0%	2000	1	250000	0	0.00001	3.625	3.625	1	93.3	93.36	93.36
	1%	1000	2	210869	1495	0.00000	3.296		1.1	92.05	73.69	
windows-log1	0%	200	11	40930	3768	0.00004	0.605	3.750	6.2	95.13	88.91	95.91
	1%	20	31	24391	6996	0.00017	0.364		10.3	93.25	79.54	
windows-log2	0%	100	14	38781	4636	0.00005	0.716	4.009	5.6	95.37	90.62	96.49
	1%	20	23	29882	9836	0.00012	0.617		6.5	95.08	91.81	
thunderbird-log1	0%	1000	3	109977	4580	0.00011	1.521	4.014	2.6	93.91	94.02	94.02
	1%	200	10	40933	2445	0.00026	0.628		6.4	92.37	84.93	
thunderbird-log2	0%	20000	1	250000	0	0.00001	3.531	3.531	1	92.45	92.52	92.52
	1%	1000	2	135229	11498	0.00007	1.962		1.8	91.04	83.53	

*avg. latency: averaged time spent on predicting the next event. *default-1* and *default-k* have the same avg. latency.

B. Results

Table III reports the online prediction results of compressed learning and its comparison with the two default approaches. The user-specified tolerable accuracy drops are 0% and 1%, with respect to *default-1*. The results are averaged over five runs with different random seeds. Standard deviation of event accuracy varies from zero to 1.9%.

Table III shows the clear benefits from our compressed learning on both prediction scope and speed. The prediction scope increases from one in *default-1* to hundreds or even thousands of events (as the "avg. pred length" column shows), and the inference time decreases by up to three orders of magnitude (as the "prediction speedup" column shows). Getting benefits on both aspects at the same time shall be no surprise. The larger prediction scopes entail the need for fewer predictions, and hence the much-reduced prediction time. As Figure 4 illustrates, for the part of the call sequence of `perl` with regular repetitive patterns, the RNN in compressed learning can recognize the patterns and makes predictions much less frequently.

In comparison, when the default method extends its prediction scope to the same as the compressed learning has, significant accuracy loss appears (e.g., 54% accuracy loss on `ocean-calls`), as the "default-k" column shows. Moreover, as Section IV-A has mentioned, to predict k events, *default-k* still needs to make k predictions; so it saves no prediction time at all.

The exact amount of speedups by compressed learning varies from sequence to sequence, depending on how often repetitive patterns show up in the sequence, which is intuitive. What is satisfying is that for traces with regular patterns, compressed learning can indeed tap into the potential, effectively recognizing the patterns and translating them into dramatic speedups, as typified by the results on the traces of `fluid`. On the other hand, on irregular traces, the method can still achieve the target accuracy while causing no slowdowns, as shown by the function call sequence of `go`, the random tree search application.

The effectiveness of the technique holds across domains and sequence types. The benefits are more pronounced on function call and memory traces than on system logs, due to the less regularity in the system logs. But it is worth noting that even on system logs, the benefits are still significant, 1–10.3× speedups of inference and up to 31× larger prediction scopes. To achieve the same prediction scopes, *default-k* suffers up to 16% accuracy drops while giving no speedups.

Runtime overhead of online tokenization. The myopic nature of online tokenization incurs a number of rollbacks in compressed learning for most sequences, as the "#rollbacks" column in Table III shows. But as Section III-D has proved,

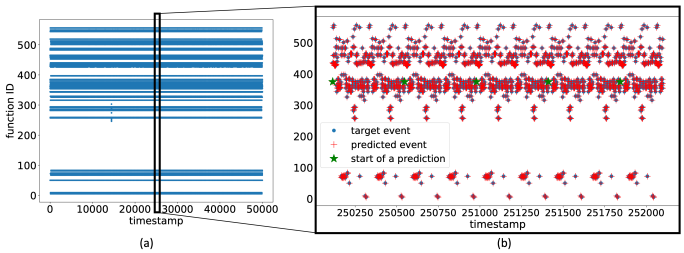


Fig. 4. (a) The function call sequence from `perl` and (b) predictions made by the RNN trained with compressed learning.

rollbacks do not cause extra invocations of predictions. The time overhead of a rollback consists of only the switch of one single reference (to point to an earlier data block that holds the recent hidden state of the RNN), which is negligible. That explains the significant speedups despite the many rollbacks in compressed learning.

The other source of runtime overhead is the time spent on tokenization for online prediction. The results are listed in column "tokenization overhead (%)" of Table III. Overall, this tokenization overhead is negligible, less than 0.2% compared to the total amount of prediction time (the time spent on the RNN model plus online tokenization) for all sequences.

Runtime overhead of partial compression. In another experiment, we turned on continual model refinement, for all the three methods in comparison. We did not see noticeable changes in the prediction results (accuracy, speedup, scope). The reason is that the one-epoch retraining is not enough to create some notable improvement of the model quality on these traces. The refinement incurs extra runtime overhead for compressed learning, as it needs to do the partial compression.

Table IV reports the time overhead of partial compression and also the speedups of our compressed learning on the one-epoch training compared to default approaches (*default*). For compressed learning, we used the FREQ that corresponds to a specified accuracy drop of 1%. *default* refers to both *default-1* and *default-k* because they use the same training strategy (e.g., train on uncompressed sequences) and thus require the same amount of training time. Overall, the time overhead of partial compression is minimal, taking up to 6.08% of the total model refinement time (including the time spent on partial compression). Increasing the number of epochs for model refinement will further reduce the percentage of compression overhead. As a side effect of compression, continual model refinement on compressed sequences for one epoch is up to 39.4× faster than default approaches.

TABLE IV
RUNTIME OVERHEAD OF CONTINUAL MODEL REFINEMENT.

Sequences		compression	refinement
No.	Name	overhead (%)	speedup (×)
1	fluid-calls	0.03	22.1
2	go-calls	4.35	1.2
3	molecule-calls	0.06	5.4
4	perl-calls	0.02	11
5	ocean-calls	0.03	7.8
6	waves-calls	2.04	21.5
7	fluid-mem	0.1	39.4
8	go-mem	0.16	7.6
9	molecule-mem	6.08	1.8
10	ocean-mem	0.96	4.4
11	perl-mem	1.29	5.9
12	waves-mem	2.12	5
13	windows-log1	0.01	5.6
14	windows-log2	0.01	5.9
15	thunderbird-log1	0.02	3.4
16	thunderbird-log2	0.03	1.7

* Each RNN model is refined for one epoch for both *default-1* and compressed learning.

V. RELATED WORK

Deep learning on compressed inputs. There are some studies on deep neural network (DNN) training and inference with compressed input data, but all on images and convolutional neural network (CNN) [37]–[41]. In Natural Language Processing (NLP), the representation of inputs sometimes uses some tokens to represent some common phrases. An example is Byte Pair Encoding (BPE) [42] used in subword tokenization. These representations are at the word or phrase level, offering no systematic ways to identify patterns in a long sequence of events and code them concisely. Moreover, as those studies work on separate sentences instead of continuous event streams, rather than online tokenizing inputs continuously, they use a preprocessing step to first tokenize the entire sentence before feeding it to the DNN. They are not applicable to streaming event sequences. To the best of our knowledge, this work gives the first proposal of compressed learning for RNNs on streaming event sequences.

DNNs for program traces. Some recent works have proposed applying DNNs on program traces for program behavior prediction. A study [4] uses an offline attention-based LSTM model to provide insights for designing a simple online hardware cache replacement policy. Perceptron-based Prefetch Filtering (PPF) [43] enhances the existing state-of-the-art prefetchers by observing the stream of candidate prefetches generated by a prefetcher, and then rejects those that are predicted by the online-trained neural model to be inaccurate. Another study [3] applies sequence learning to prefetching and proposes using LSTM to understand the semantic information of the underlying application given a memory access trace. A recent work [5] proposes an RNN-based page scheduler for programs that execute over hybrid memory systems. None of them have considered learning from the compressed traces.

DNNs for system logs. System logs record detailed software runtime information which allows software developers to track and analyze system behaviors. Recent years have seen a growing interest in applying Deep Learning models in analyzing system logs. One study [7] proposes Deeplog, which leverages LSTM for online anomaly detection. Another work [44] proposes to use RNN with the attention mechanism for anomaly detection. Some other work [6] builds an RNN-based content caching framework to predict the popularity of content objects on information-content networks. Wang and

others [45] used RNNs to predict the probability that a user will access a particular activity given their historical access logs. No prior work has proposed learning from compressed log sequences.

Sequitur [17] has been applied to various tasks, including program and data pattern analysis [46]–[48]. It has not been introduced into RNN learning before.

VI. Conclusion

This paper presents *CFG-guided compressed learning*, the first known approach to integrating sequence compression into RNN learning and inference for both expanded prediction scope and reduced inference latency. It builds on CFG and online tokenization, and addresses a series of complexities through the design of efficient rollback, accuracy-conscious lowering, partial compression, and other techniques. By discovering and leveraging patterns in a sequence effectively, it enables much faster inferences while achieving a substantially expanded prediction scope on 16 real-world sequences with repetitive patterns. Future work includes generalizing compressed learning to other autoregressive models and recent architectures such as Transformers.

References

[1] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[2] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4505–4515.

[3] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *International Conference on Machine Learning*, 2018.

[4] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[5] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A hybrid memory page scheduler with machine intelligence," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 37–48.

[6] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "Deepcache: A deep learning based framework for content caching," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018, pp. 48–53.

[7] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1285–1298.

[8] A. Beutel, P. Covington, S. Jain, C. Xu, J. Li, V. Gatto, and E. H. Chi, "Latent cross: Making use of context in recurrent recommender systems," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 2018, pp. 46–54.

[9] D. Neil, M. Pfeiffer, and S.-C. Liu, "Phased lstm: Accelerating recurrent network training for long or event-based sequences," in *Advances in neural information processing systems*, 2016, pp. 3882–3890.

[10] M. S. Zhang and B. Stadie, "One-shot pruning of recurrent neural networks by jacobian spectrum evaluation," *arXiv preprint arXiv:1912.00120*, 2019.

[11] M. Zhu, J. Clemons, J. Pool, M. Rhu, S. W. Keckler, and Y. Xie, "Structurally sparsified backward propagation for faster long short-term memory training," *arXiv preprint arXiv:1806.00512*, 2018.

[12] L. Liu, L. Deng, Z. Chen, Y. Wang, S. Li, J. Zhang, Y. Yang, Z. Gu, Y. Ding, and Y. Xie, "Boosting deep neural network efficiency with dual-module inference," 2020.

[13] S. H. Park, B. Kim, C. M. Kang, C. C. Chung, and J. W. Choi, "Sequence-to-sequence prediction of vehicle trajectory via lstm encoder-decoder architecture," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, pp. 1672–1678.

[14] S. Wiseman and A. M. Rush, "Sequence-to-sequence learning as beam-search optimization," *arXiv preprint arXiv:1606.02960*, 2016.

[15] H. Cheng, P.-N. Tan, J. Gao, and J. Scripps, "Multistep-ahead time series prediction," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2006, pp. 765–774.

[16] A. Ketterlin and P. Clauss, "Prediction and trace compression of data access addresses through nested loop recognition," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 94–103.

[17] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: a linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.

[18] P. Bille, A. R. Christiansen, P. H. Cording, and I. L. Gørtz, "Finger search in grammar-compressed strings," *arXiv preprint arXiv:1507.02853*, 2015.

[19] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann, "Random access to grammar-compressed strings and trees," *SIAM Journal on Computing*, vol. 44, no. 3, pp. 513–539, 2015.

[20] N. R. Brisaboa, A. Gómez-Brandón, G. Navarro, and J. R. Paramá, "Gract: a grammar-based compressed index for trajectory data," *Information Sciences*, vol. 483, pp. 106–135, 2019.

[21] Y.-H. Qian, D. d'Humières, and P. Lallemand, "Lattice bgk models for navier-stokes equation," *EPL (Europhysics Letters)*, vol. 17, no. 6, p. 479, 1992.

[22] R. Coulom, "Computing "elo ratings" of move patterns in the game of go," *ICGA journal*, vol. 30, no. 4, pp. 198–208, 2007.

[23] "Molecular modeling," http://ambermd.org/, 2020, accessed: 2020-09-30.

[24] "Roms: Regional ocean modeling system," http://www.myroms.org/, 2020, accessed: 2020-09-30.

[25] "Perl interpreter," http://www.perl.org/, 2020, accessed: 2020-09-30.

[26] A. Vaziri and M. Kremenetsky, "Visualization and tracking of parallel cfd simulations," 1995.

[27] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: A large collection of system log datasets towards automated log analytics," *arXiv preprint arXiv:2008.06448*, 2020.

[28] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 2007, pp. 575–584.

[29] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y. N. Patt, "Improving the performance of object-oriented languages with dynamic predication of indirect jumps," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 80–90.

[30] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 513–526.

[31] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 2001, pp. 14–25.

[32] Y. Solihin, J. Lee, and J. Torrellas, "Using a user-level memory thread for correlation prefetching," in *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 2002, pp. 171–182.

[33] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[35] Y. Zeng and X. Guo, "Long short term memory based hardware prefetcher: a case study," in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 305–311.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization. international conference on learning representations (2015)," 2015.

[37] D. Fu and G. Guimaraes, "Using compression to speed up image classification in artificial neural networks," 2016.

[38] R. Torfason, F. Mentzer, E. Ágústsson, M. Tschannen, R. Timofte, and L. V. Gool, "Towards image understanding from deep compression without decoding," in *International Conference on Learning Representations*, 2018.

[39] L. Gueguen, A. Sergeev, B. Kadlec, R. Liu, and J. Yosinski, "Faster neural networks straight from jpeg," in *Advances in Neural Information Processing Systems*, 2018, pp. 3933–3944.

[40] Z. Liu, T. Liu, W. Wen, L. Jiang, J. Xu, Y. Wang, and G. Quan, "Deepn-jpeg: A deep neural network favorable jpeg-based image compression framework," in *Proceedings of the 55th Annual Design Automation Conference*, 2018.

[41] X. Xie and K.-H. Kim, "Source compression with bounded dnn perception loss for iot edge computer vision," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019.

[42] P. Gage, "A new algorithm for data compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, 1994.

[43] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[44] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols, "Recurrent neural network attention mechanisms for interpretable system log anomaly detection," in *Proceedings of the First Workshop on Machine Learning for Computing Systems*, 2018, pp. 1–8.

[45] H. Wang, Z. Wang, and Y. Ma, "Predictive precompute with recurrent neural networks," *arXiv preprint arXiv:1912.06779*, 2019.

[46] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for variable length intervals and hierarchical phase behavior," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE, 2005, pp. 135–146.

[47] T. M. Chilimbi, "Efficient representations and abstractions for quantifying and exploiting data reference locality," in *ACM SIGPLAN Notices*, vol. 36, no. 5. ACM, 2001, pp. 191–202.

[48] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Using formal grammars to predict i/o behaviors in hpc: The omnisc'io approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2435–2449, 2015.