EVENT SEQUENCE IDENTIFICATION AND DEEP LEARNING CLASSIFICATION FOR ANOMALY DETECTION AND PREDICATION ON HIGH-PERFORMANCE COMPUTING SYSTEMS

Zongze Li

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

December 2019

Li, Zongze. *Event Sequence Identification and Deep Learning Classification for Anomaly Detection and Predication on High-Performance Computing Systems.* Doctor of Philosophy (Computer Science and Engineering), December 2019, 95 pp., 27 tables, 15 figures, 95 numbered references.

High-performance computing (HPC) systems continue growing in both scale and complexity. These large-scale, heterogeneous systems generate tens of millions of log messages every day. Effective log analysis for understanding system behaviors and identifying system anomalies and failures is highly challenging. Existing log analysis approaches use line-by-line message processing. They are not effective for discovering subtle behavior patterns and their transitions, and thus may overlook some critical anomalies. In this dissertation research, I propose a system log event block detection (SLEBD) method which can extract the log messages that belong to a component or system event into an event block (EB) accurately and automatically. At the event level, we can discover new event patterns, the evolution of system behavior, and the interaction among different system components. To find critical event sequences, existing sequence mining methods are mostly based on the a priori algorithm which is compute-intensive and runs for a long time. I develop a novel, topology-aware sequence mining (TSM) algorithm which is efficient to generate sequence patterns from the extracted event block lists. I also train a long short-term memory (LSTM) model to cluster sequences before specific events. With the generated sequence pattern and trained LSTM model, we can predict whether an event is going to occur normally or not. To accelerate such predictions, I propose a design flow by which we can convert recurrent neural network (RNN) designs into register-transfer level (RTL) implementations which are deployed on FPGAs. Due to its high parallelism and low power, FPGA achieves a greater speedup and better energy efficiency compared to CPU and GPU according to our experimental results.

ACKNOWLEDGMENTS

I want to express my sincere gratitude to my advisor, Dr. Song Fu, for his patient and meticulous guidance in my 5-year Ph.D. study.

I also want to thank all the professors on my dissertation committee:

Dr. Yan Huang

Dr. Hui Zhao

Dr. Xiaohui Yuan

for their guidance and encouragement. Without their help, I cannot achieve that much in my research.

I want to particularly thank Dr. Hui Zhao for allowing me to use her VC707 FPGA board. Without that powerful platform, I cannot test and evaluate my RNN design on FPGA.

I want to particularly thank Dr. Robin Pottathuparambil for giving me suggestions on my reseach of implementing RNN on FPGA.

I want to apprecite my parents and my wife: Yi Li, Chunmei Sheng, Yuxuan Wu, for their support and encourage in my five years of study. I love you all.

I also want to thank my research mentor Sean Blanchard at Los Alamos National Laboratory (LANL) for giving me the opportunity of researching the syslog traces collected from DOE HPC system Mutrino. He also provided me internship opportunities in the summers of 2016 and 2017 and gave me a lot of suggestions while I was working at LANL.

I want to thank Dr. Xiaoguang Tian, who is currently an Assistant Professor at Purdue University. Dr. Tian graduated from the UNT College of Business. He provided me real-world marketing data and cooperated with me on research that uses deep learning techniques to cluster marketing data, and we presented our results [83] on the SWDSI conference.

During my 5 years of research, I have the following publications: [63], [83], [64], [25], [65], [37], [44], [43].

Last but not least, I want to say thank you to all of my friends: Dr. Qiang Guan, Dr. Ziming, Zhang, Dr. Song Huang, Shuwen Liang, Zhi Qiao, Jacob Hochstetler, Shuo Sun, Dr. Islam Rezbaul, Zhaochen Gu, Xu Ma, Sihai Tang, Yuxin Mei.

Thank you for all your help, accompanying, and encouragement which enables me to persist in my dissertation research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

High-performance computing (HPC) systems continue growing in both scale and complexity. For example, the Summit supercomputer at Oak Ridge National Laboratory has 2,414,592 compute cores [16]. The Trinity supercomputer at Los Alamos National Laboratory has more than 19,000 heterogeneous (i.e., Intel Haswell and Intel Knights Landing) compute nodes [36]. Titan at Oak Ridge National Laboratory has 18,688 compute nodes equipped with AMD Opteron processors and NVIDIA Tesla GPUs [14]. These large-scale, heterogeneous systems generate tens of millions of log messages every day. In addition to the sheer volume, both the format and the content of these log messages vary dramatically, depending on system architecture, hardware configuration, management software, and type of applications. Effective log analysis for understanding system behavior and identifying system anomalies and failures is highly challenging.

Log analysis for system behavior characterization has continuously been an important research topic. Existing approaches use line-by-line log analysis. Although they can discover distribution and precedence relation among log messages, they are not effective for discovering subtle behavior patterns and their transitions, and thus may overlook some critical anomalies. However, log messages are not isolated from each other. An event of a component or an event of the system may produce multiple messages. Analysis at the event level can provide a richer semantics of system behaviors and thus enables to detect more subtle anomalies that the traditional line-by-line analysis methods cannot find.

In this dissertation research, I use the event block (EB) to represent the log messages that belong to a component or system event. Event block-based analysis is not trivial. A fundamental challenge is how to identify log messages that are related to the same event and thus group them into an EB. The high concurrency in HPC systems causes messages from different events and even from different nodes to overlap with each other. It is common to see messages of an event are scattered into multiple pieces by messages from several other events. Moreover, the overlap does not follow a fixed pattern.

Additionally, some messages may appear, disappear, or have variable contents in different instances of an event. Without detailed execution contexts, such as information of application workload, system processes, scheduling method, and device status, it is difficult to identify event blocks accurately. This context information itself, however, is not easy to obtain in large-scale production HPC systems.

Despite these challenges, the advantages of Event Block-based log analysis are intriguing. By converting the original, lengthy, and unstructured messages in syslogs into a compact and structured list of EBs, the complexity of log analysis can be significantly reduced, and the results are more interpretable and easier to understand. By working at the level of EBs, we can find patterns of events, the evolution of system behavior, and the interaction among different system components, which is very difficult to achieve by using the traditional message-level analysis. Variation among instances of an event is also an indicator of possible anomalies. Thus, a larger set of anomalies can be detected.

In my research, I propose a System Log Event Block Detection framework, called SLEBD, that can extract event blocks accurately and automatically. SLEBD explores the probability of messages occurring together in a flexible period and leverages the law of total probability to consolidate messages that occur together even with variations into EB patterns from syslogs.

Then, I explore SLEBD for anomaly detection and problem diagnosis. We found several new types of system-level anomalies in our experiments. Based on the identified event blocks, we define an event sequence classification problem and leverage deep learning methods, more specifically, Recurrent Neural Network (RNN), to analyze event sequences and characterize system behavior. RNN model can help us predict if an event will successfully happen. However, existing RNN models work on sequences with boundaries. As system logs are generated in real-time, existing RNN models are not suitable for predicting real-time events from system logs. To solve this problem, we use a sequence mining method to detect sequence patterns from existing event lists and use these sequence patterns to characterize

HPC's behavior from the system log. We further predict if some events are going to happen and use the trained RNN models to determine if an event is normal or not.

Sequence mining aims to discover important patterns among a set of objects. It can help us discover regularity among events, detect anomalies, and predict events in HPC environments. Existing sequence mining methods are mostly based on the Apriori algorithm [23]. Apriori-like algorithms perform multiple rounds of scanning of objects to detect all possible pattern candidates, which is compute-intensive and runs for a long time. Moreover, this type of algorithms generates a large number of candidate sequences which need a large database to store and analyze. The time and space complexity of Apriori-like algorithms makes them inefficient for processing objects with a large population and complex relation, such as huge logs and various events in production HPC systems.

We aim to detect anomalous system and component behaviors from a large number of events identified in HPC systems by developing a novel, topology-aware sequence mining (named TSM) method. Our previous studies of HPC systems have revealed that certain events commonly happen on compute nodes. Some events may appear multiple times in a period. However, a new execution sequence does not start until an old sequence of the same type is completed. As a result, the event sequence can be represented by a Directed Acyclic Graph [60]. This finding inspires us to design a sequence mining algorithm that leverages the topology [49] information. TSM identifies all possible long sequences among system and component events after generalizing their positions from massive events in a cost-effective manner. TSM is highly efficient as it scans events only once to collect the temporal and spatial information of events, and it does not require a large number of comparison and merge operations to generate sequences. In addition to the low time complexity, TSM is space-efficient as it needs small memory and storage space to store the scanning results and temporary data. In our experiments, we use the produced event patterns to further detect system anomalies by leveraging recurrent neural network [33] models.

To cluster Event Block sequences, we use Python-based Deep Learning library Keras[9] to create and train our RNN models. We find that the software-based RNN models run on general-purpose CPU or GPU cannot achieve high efficiency.

Field Programmable Gate Array (FPGA) is a good platform for accelerating the performance of massive floating-point data computations. With the unique features of both high parallelism and low power, FPGA can achieve a significantly higher speedup and better energy efficiency compared to CPU and GPU. However, implementing Python-based RNN algorithms at the Register Transistor Level (RTL) is very time-consuming. The existing approaches generate RTL by designing RNN algorithms in the C program and use High Level Synthesis (HLS) tools [94] to convert the C program into RTL. The HLS tool generated RTL design is not optimized and cannot achieve high efficiency and cannot fully show FPGA's advantage.

To address this problem, I propose a design flow which can automatically convert Python-based RNN designs into RTL code. Our experimental results show the generated RTL run on FPGA code is about seven times faster than the Python code run on CPU, and their output results are the same.

The four research tasks, i.e., event extraction, event sequence identification, anomaly detection, and deep learning-based anomaly prediction, are connected and integrated. By grouping related system messages into event blocks, we convert unstructured system logs into structured event block lists and obtain event semantics, which enables us to detect event-level anomalies that have not been discovered before. In order to detect event anomalies, we need to understand system behavior which can be characterized by event sequences. Deviations from the normal, critical event sequences could be an anomaly. Deep learning methods, such as long short term memory (LSTM), can efficiently classify event sequences and predict future anomalies. However, deep convolutional neural networks involve intensive floating-point computations, which makes them slow on CPU and even GPU (power-hungry). We propose to explore low-power and highly-parallel FPGA to accelerate RNN computation and thus make event-level anomaly detection and prediction faster.

CHAPTER 2

RELATED WORK

2.1. Existing Log Analysis Tools and Existing System Log Anomaly Detection Methods

Several methods have been proposed and developed to analyze the system log. For example, in a study of 200,000 Splunk queries, queries are clustered into several categories, and transformation sequences are detected to analyze connection among queries [56]. In [45], console, netwatch, consumer and apsched logs are stored in a MySQL database and accessed through a web interface for application profiling. A three-layer filtering method is used to compress log data without losing important information [62]. System logs from Blue Gene/P and five supercomputers are analyzed in [27] [82], and several important observations about failure characteristics are reported. Time coalescence techniques are also used to analyze supercomputer logs [29]. An apriori-like algorithm and correlation graphs quantify correlation among log messages are used to perform event prediction [70]. Kim et al. [57] combined the misuse detection method and anomaly detection method and developed a hybrid intrusion detection system to detect attacks from the network. Hong et al. [47] presented an integrated Anomaly Detection System, which consists of host- and network-based anomaly detection systems to detect intrusions to a target system. He et al. [50] evaluated three supervised anomaly detection methods (i.e., Logistic Regression[87], Decision Tree[54], and SVM[88]) and three unsupervised methods (i.e., Log Clustering[95], PCA[74], and Invariant Mining[89]) and developed a toolkit for detecting anomalies from system logs. Du et al. [35] presented a deep neural network model using LSTM to process a system log as natural language sequences, which can learn log patterns from normal executions, and detect anomalies when log patterns deviate from the model trained for normal execution. Pandeeswari et al. [73] developed a hybrid algorithm, which is a mixture of the Fuzzy C-Means clustering algorithm and the Artificial Neural Network (FCM-ANN) to build an anomaly detection system in the cloud.

Several tools are available for analyzing log messages. For example, Baler [34] uses a list of keywords to scan log messages and counts the number of times that each keyword appears. SLCT [85] clusters log lines based on the frequency of keyword-position occurrences. HELO [58] splits log messages in a training file into clusters. The splitting position in a message line is determined by those words that appear most frequently. To pre-process system logs, Zheng et al. [46] used the occurrence probability to model the causal relation among single-line messages. Their method, however, cannot be used to determine causally related log messages that belong to an event with a high confidence. There are also studies that explore system logs for anomaly detection. For example, Lou et al. [66] calculated the occurrence probability among log lines in a time window to analyze their dependency. Fu et al. [67] used a finite state automaton (FSA) to model execution paths of a system and applied these FSAs to detect anomalies from log messages. Xu et al. [55] grouped single-line patterns based on alphabetic words in line groups. They applied the principal component analysis method to the extracted feature vectors to detect anomalies. Baseman et al. presented a textual-numeric data ground graph [37] to analyze log messages and used Info-map [24] to extract clusters from subgraphs of the ground graph. These approaches focus on individual log messages and analyzing distributions or precedence relation among log lines. By exploring event blocks, our proposed approach transforms unstructured log messages into structured event sequences, which enables us to identify event patterns and more subtle system and component anomalies.

2.2. Existing Sequence Mining Methods

Existing sequence mining methods are mostly Apriori-like [23]. An Apriori-like method, such as GSP [81], AprioriAll [48] [22], and SPADE [93], uses multiple candidate generation-and-mining scans and tests to produce all possible sequences. This process is both time-consuming and space-inefficient. To address this issue, Han et al. proposed FreeSpan [32]. FreeSpan scans target objects to generate length-2 subsequences. It then projects length-2 subsequences to length-3 subsequences and continues until no more sequences of longer length can be projected from shorter subsequences, e.g.,

$$[A, B], [B, C], [A, C] \rightarrow [A, B, C]$$

Example 1. How existing sequence mining methods generate longer sequence pattern from a shorter pattern

A length-N sequence is projected from N length-(N-1) subsequences, which requires N to $\sum_{i=1}^{N} i$ times of subsequence comparisons and merges. As an example, Table 2.1 shows a simple set of events.

Sequence ID	*Sequence*
Seq_1	$[A, B, C, A, B, C, A, B, C]$
Seq_2	$[A, B, C, A, B, C, A, B, C]$

Table 2.1. EXAMPLE OF EVENTS

FreeSpan can produce 325 length-1 to length-9 subsequences. Most of the sequences are subsequences of other sequences. However, there is only one useful length-3 sequence, i.e., [A, B, C] in this example. In our experiments on an HPC system, one event sequence contains more than 80 events. In the worst case, FreeSpan needs to generate 3,160 length-2 subsequences and 82,160 length-3 subsequences for a length-80 sequence.

To reduce the large number of subsequences generated by FreeSpan, Han et al. developed an improved algorithm, called PrefixSpan [84], which treats some subsequences as prefixes and only projects subsequences of longer length based on the prefix subsequences. The PrefixSpan algorithm has been used in many areas, for example, analysis of supermarket records [31].

A critical problem with Apriori clustering is that objects clustered into one class must have the same number of occurrences. In the HPC system, however, events do not happen for the same number of times. One event could be followed by repeated events and vice versa. Thus, Apriori-like algorithms are not suitable for event analysis in HPC environments.

Han et al. developed a frequent pattern mining method called FP growth [76]. It mines frequent patterns by generating an FP-tree and avoids using the Apriori algorithm, which can reduce the execution time. However, the frequent patterns discovered by the FP growth do not include sequential information, which makes it unsuitable for finding the event execution sequence patterns in our research.

The preceding issues require us to develop a new sequence mining method which can discover execution sequences from various events on HPC systems. In this paper, we propose a topology-aware sequence mining (TSM) algorithm to address the following major problems in the existing sequence mining methods.

1. They generate a huge number of temporary, short subsequences – high space complexity.

2. They perform many comparison operations and merging short subsequences into longer sequences – high time complexity.

3. They produce many misleading subsequences which are not useful for behavior analysis and anomaly detection for HPC systems.

2.3. Existing Methods of Implementing Deep Learning Algorithm on FPGA

Implementing RNN/LSTM on FPGA is a good research topic in recent years. For example, Guan et al.[51] designed an LSTM forward propagation section feature in C code and used the High-Level Synthesis (HLS)[94] tool to convert their design into RTL. Chang et al.[30] and Ferreira et al. [53] have designed the LSTM forward propagation section in RTL and implement them on Xilinx Zynq platforms, these are a good breakthrough, but they can only partially accelerate prediction performance because their main program is running as SOC software on Zynq chip. The above three research's LSTM model is trained on the desktop computer but not on FPGA. Li et al. [86] designed an RNN model in SystemC[12], which includes both Forward propagation and Backward propagation sections, and they also use the HLS tool to convert their design into RTL. Their design can do model learning on-chip. Both four works are describing their values as fixed-point data, which can save calculation time but could lose accuracy.

Using HLS tools to help us generate our RTL design code is not our propose. As we can find, using HLS tools cannot achieve fully optimized performance. We will discuss this finding in Chapter 5.

Use off-chip trained RNN/LSTM model and partially accelerate predict section is not our research's purpose. We wish to do both forward and backward propagation sections on-chip. And our research will use floating-point type data, not fixed-point to maintain accuracy.

CHAPTER 3

SYSTEM LOG EVENT BLOCK DETECTION FRAMEWORK

3.1. Introduction

Existing HPC system log analyze tools analyze logs in a line-by-line fashion. However, as we find, an event of a component or the system may produce multiple messages. Analysis at the event level can provide a richer semantics of system behaviors and thus enables to detect more subtle anomalies.

In our research, we first model original system logs into single line pattern lists. Then we leverage an extended form of the Bayes' theorem[75], which is called the Law of Total Probability [72] to calculate the probability of each line patterns happening together with other patterns and group the happening together patterns into Event Block pattern. Such method can tolerate noises in pattern learning files.

SLEBD extracts Event Blocks from original log files to generate Event Block lists. This feature can handle real time streaming system log messages. If one log message which is supposed to be included an Event Block pattern but is treated individually, SLEBD reports an anomaly.

3.2. System Log Preprocessing

The format of system logs varies as system architecture, operation system, runtime, management tools, and applications can be different. For example, Mutrino, which is a Cray XC40 system at Sandia National Laboratories, has its syslogs in the following format:

2015-02-13T13:16:11.865060-06:00 c0-0c0s0n1 trying chooser simple

Example 2. Syslog format on the Mutrino supercomputer (SNL)

where "2015-02-13T13:16:11.865060-06:00" is a timestamp, "c0-0c0s0n1" is a node ID, and "trying chooser simple" is the message body.

We aim to design our event block analysis framework to be generic, being capable of processing and analyzing system logs in different formats. Our tool requires limited user involvement. Users only provide message syntax information, indicating the structure of log messages. Then, SLEBD parses messages and extracts message elements by using the syntax information.

Log Preprocessing: System logs collected from production HPC systems are complex. Messages from all compute nodes and service nodes are mixed, which makes it difficult to identify event blocks accurately.

The log preprocessing process filters and separates those mixed messages. We separate streaming messages or messages from large mixed logs into multiple files based on node IDs, that is one file for each node. These node-wise log files are first analyzed individually to extract line patterns and event blocks on each node and then combined to identify event blocks across multiple or all nodes. Additionally, the preprocessing process formats messages and removes incomplete messages, which facilitates the learning of line patterns for event block extraction.

Event blocks contain multiple lines of messages. However, they may only appear once in a period on a single node. They cannot be captured if only one node's log file is analyzed. To solve this problem, we define an inspection time window and produce a directory to store log messages from all nodes in a window. Figure 3.1 shows the structure of the preprocessed log sets.

3.3. Event Block Database and Event Block Extraction

Using the preprocessed log files as input, our System Log Event Block Detection (SLEBD) framework performs 1) event block database (EBD) generation, 2) event block extraction, and 3) anomaly detection. SLEBD uses a Line Pattern Hash Table (LPHT) and the EBD to conduct event block extraction. Figure 3.2 shows the major components and workflow of SLEBD.

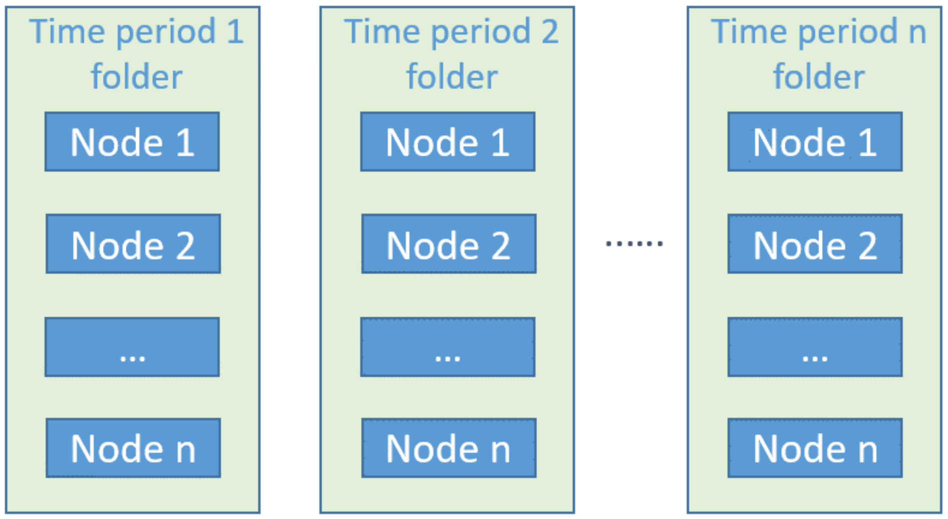


FIGURE 3.1. Structure of the preprocessed log sets

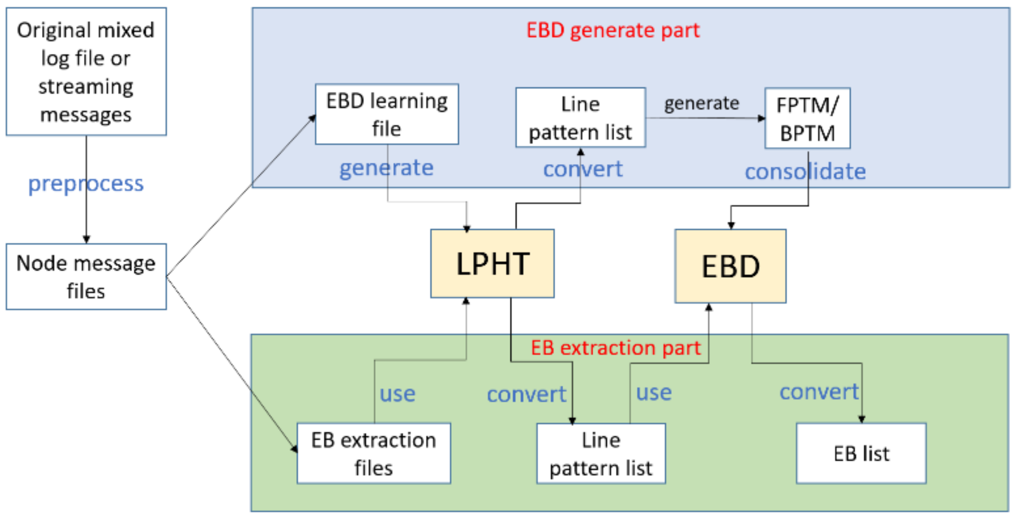


FIGURE 3.2. Major components and workflow of SLEBD

3.3.1. Event Block Database (EBD) Generation

Single line patterns

Syslog listens for messages on /dev/log. Multiple threads or devices may generate log messages of an event. They have certain line patterns in syslog but may contain variations. The following are two examples of messages taken from Mutrino's syslog.

2015-02-13T13:19:42.494097-06:00 c0-0c2s1n1 ACPI: PCI Root Bridge [PCI0]

(domain 0000 [bus 00-fe])

2015-02-13T13:16:45.462890-06:00 c0-0c0s0n1 ACPI: PCI Root Bridge [UNC0] (domain 0000 [bus ff])

Example 3. Two Mutrino syslog message examples

Even though the preceding messages are produced from two different nodes, they are generated by the same process and thus have similar message structure.

SLEBD processes messages in syslogs and generates a line pattern for each message. Each line pattern has a unique identifier in the form of "[LinePattern_$num]". Alphabetic words to create one line pattern in the message and their corresponding positions. Numbers are treated as variables, and not included in line patterns. The line pattern of the preceding messages is as follows. Although the two messages are different, their line patterns are the same. Their line patterns are the same:

[[0, ACPI:], [1, PCI], [2, Root], [3, Bridge], [5, (domain), [6, [bus]]

Example 4. A single line pattern

Two messages are called k% similar if at least k% of words and their positions in their line patterns are the same. Two messages have the same pattern if they are k% similar and k is greater than a defined threshold. Otherwise, we say their line patterns are different. Variables in a message have numbers and may contain letters, such as username and node ID. It is not necessary to have an exact match for two variables as they are less critical than alphabetic words which are the skeleton of a message. For example, in the preceding log messages, the bus channel on Node "c0-0c0s0n1" is named "ff," while the bus channel on Node "c0-0c2s1n1" is named "00-fe". Both are variables, and their mismatch does not affect the similarity of their corresponding line patterns. Variables are analyzed later for anomaly detection.

These generated line patterns are stored in a Line Pattern Hash Table (LPHT). For a new log message, SLEBD searches for the most similar (i.e., k-similar with high k-value) pattern(s) from LPHT. If we found no similar pattern, we will add a new line pattern to LPHT.

Line Pattern Forward/Backward Transition Matrix (FPTM/BPTM)

By using single-line patterns stored in LPHT, we convert the original, unstructured log messages into a line pattern list. SLEBD produces a line pattern list for each node in a system. These line pattern lists for different nodes are then co-analyzed to determine how often every pattern occurs and what adjacent line patterns happen before (called backward patterns) and after (called forward patterns) the line pattern in question. We will generate A Forward Probability Transition Matrix (FPTM), and a Backward Probability Transition Matrix (BPTM). For example, assume the line pattern lists of a two-node system are as follows.

Node ID	*Line pattern list*
Node 1	*A, D, E, F*
Node 2	*A, E, F*

Table 3.1. AN EXAMPLE OF LINE PATTERNS IN A TWO-NODE SYSTEM

	A	*D*	*E*	*F*	*Finish*
A		0.5	0.5		
D			1		
E				1	
F					1

Table 3.2. FPTM GENERATED FROM LINE PATTERN LISTS

Table 3.1's corresponding FPTM and BPTM are presented in Table 3.2 and 3.3.

In the example of Table 3.1, we can see LinePattern_A occurred twice and have two forward patterns, LinePattern D & E. Thus, in the FPTM, PF $(A \rightarrow B) = 50\%$ and

	A	D	E	F	$Start$
A					1
D	1				
E	0.5	0.5			
F			1		

TABLE 3.3. BPTM GENERATED FROM LINE PATTERN LISTS

PF ($A \rightarrow D$) are equal to 50%. And LinePattern_D have only one backward pattern, LinePattern_A. In the BPTM, PB ($A \rightarrow D$) = 1. From this view, we can find that BPTM is not a transpose of FPTM.

Event Block Generation and Consolidation

Based on the Bayes' theorem [75], the occurrence probability of an event depends on that of events that are related to this event. The two matrices, FPTM and BPTM, described in the preceding section, contain information of forwarding and backward line patterns, respectively. We apply the Bayes' theorem by using the two matrices to find the most relevant line patterns in the forward and backward ranges of a line pattern to identify an event block.

To this end, we leverage an extended form of the Bayes' theorem, which is called the Law of Total Probability [72], expressed as:

$$P(E|A) = \sum_{i=1}^{n} P(E|A \cap B_i) * P(B_i|A) \tag{3.1}$$

We use it to calculate the probability of a line pattern E happening in the forward range of a line pattern A. We use "PF($A \rightarrow E$)" to denote this forward probability and "PB($A \rightarrow E$)" to denote the backward probability. Thus, we have

$$PF(A \rightarrow E) = \sum_{i=1}^{n} PF(A \rightarrow B_i) * PF(B_i \rightarrow E) \tag{3.2}$$

where Bi represents line patterns that immediately follow the line pattern A. We use the FPTM matrix to find those line patterns.

If both PF($A \rightarrow E$) and PB($A \leftarrow E$) are greater than a Threshold of Occurring Together (TOT), we say line patterns A and E occur together with high confidence. The threshold TOT can be set to its initial value provided by system administrators or dynamically updates its value based on the occurrence frequency of the line pattern A.

Those line pattern pairs that are identified to occur together are stored in an Occurring Together pattern Pair List (OTPL). In OTPL, each line pattern p has two lists, one of which contains the line patterns that happen before (i.e., backward) p. The other list stores those line patterns that appear after (i.e., forward) p. Table 3.4 shows an OTPL generated based on FPTM and BPTM.

Pattern ID	*Backward Pattern List*	*Forward Pattern List*
A	{}	*E*
D	{}	{}
E	*A*	*F*
F	*E*	{}

Table 3.4. OCCURRING TOGETHER PATTERN PAIR LIST (OTPL) GENERATED FROM FPTM AND BPTM

Event block consolidation: SLEBD starts with those line patterns whose backward pattern lists are empty but forward pattern lists are not empty in OTPL in the EB consolidation process. A first-in-first-out (FIFO) list and one Temporary EB pattern List (TEBL) are used. First, SLEBD puts a line pattern to the end of the FIFO list. Then,

1. SLEBD selects the first line pattern P_{top} in the FIFO list and stores it in TEBL.

2. If the forward pattern list of P_{top} is not empty, puts those line patterns which are in the forward pattern list of P_{top} but not in the FIFO list yet, to the end of the list.

3. Adds P_{top} to TEBL if P_{top} does not exist in it.

This process continues until the FIFO list becomes empty.

When the EB consolidation process stops, SLEBD considers the line patterns in TEBL as one event block and assigns an EB ID in the form of "[Block_$num]" to the new EB pattern.

All EB patterns are stored in the Event Block Database (EBD). The line pattern whose backward list is empty but forward list is not empty, e.g., A in the example is marked as the start line pattern. The line pattern whose forward list is empty but backward list is not empty, e.g., F is marked as the end line pattern. SLEBD updates the information of those line patterns in the Line Pattern Hash Table (LPHT) to indicate to which EB patterns they belong.

The Threshold of Occurring Together (TOT) and Confidence Interval

We use the Threshold of Occurring Together (TOT) in Section 3.3.1. SLEBD can dynamically update TOT at runtime.

The reason that TOT is used is that when consolidating EBs, SLEBD should tolerate noise when building EBD from syslogs. In SLEBD, two-line patterns do not have to occur together all the time to decide they are one pair of occurring-together patterns. More often a line pattern appears the higher accuracy in which the occurring-together patterns can be identified. This feature mitigates the influence of noise or incomplete information in the learning process. As a result, if a line pattern occurs less frequently, its TOT is lower. The TOT increases as a line pattern occur more often, and the conditions for identifying its occurring-together patterns are more accurately identified.

To realize this design, we use a confidence interval generated by the square root of the occurrence count to update TOT dynamically. The less frequently that SLEBD sees a line pattern, the wider the confidence interval for TOT. Table 3.5 provides some sample values of TOT for different occurrence counts.

Occurrence count	*Square root*	*Occurrence count confidence interval*	*TOT*
50	7	43 – 57	86%
100	10	90 – 110	90%
200	14	186 – 214	93%

Table 3.5. THRESHOLD OF OCCURRING TOGETHER (TOT)

This confidence interval enables SLEBD to identify line patterns that should not be considered as occurring together. For example, if LinePattern_2's occurrence count does not satisfy LinePattern_1's confidence interval, SLEBD does not calculate the occurring-together probability of LinePattern_1 and LinePattern_2. As the occurrence count and occurrence count confidence interval corresponding feature can avoid computing unrelated line patterns' relationship, it also helps reduce computation workload.

The value of TOT should not be manually assigned very small, as this may cause unrelated line patterns to be considered as occurring-together.

Enhancing EBD from using multiple time windows of system logs: The number of log messages affects the accuracy of event block identification. The longer period a log covers, the higher the probability that more event blocks and their complete patterns can be included. In our study, we found the following problems in the logs.

1. Some unrelated line patterns always occur together in a short period.
2. A log contains incomplete, mingled, or noise messages.
3. New types of messages appear due to hardware upgrades, runtime updates, installation of new monitoring tools, change of configuration, etc.

SLEBD selects log files of a period to generate the EBD. However, even a carefully selected log set may still suffer from the preceding problems. To address this issue, SLEBD keeps updating EBD as more messages are processed and analyzed. After system log files in a new time window are analyzed, add the line patterns which fulfill these conditions into the Reconsidered Pattern List (RPL):

1. LP_1 have one previous occurring-together LP_2 which occurrence count cannot fit in LP_1's recent confidence interval.

2. LP_2's previous occurrence count cannot fit in LP_1's previous confidence interval, but now they can be fit. SLEBD reconsiders LP_1's relationship with LP_2.

3. New detected line patterns.

SLEBD uses the latest FPTM/BPTM to calculate the occurring-together probability of the line patterns in RPL. If one line pattern's relationship with other patterns changes, SLEBD updates its forwardbackward pattern lists in OTPL.

After the line patterns in RPL are re-analyzed, SLEBD performs EBD consolidation using the updated OTPL. As a result,

1. Correctly split previously consolidated unrelated line patterns.

2. Consolidate previously separated but related line patterns.

3. Add newly found line patterns to LPHT, and add updated EB patterns to EBD.

Cutoff for reconsideration: SLEBD terminates the reconsidering update process according to certain configurations, which is called the reconsideration cutoff. Such cutoff conditions can be:

1. The relation of line patterns is stable for a specified period, for example, 30 days;

2. A-line pattern's occurrence count reaches a certain number, for example, 500 times.

3.3.2. Event Block Extraction

SLEBD uses EBD to extract event blocks from system logs. The EB extraction process uses the line pattern list as input. It explores LPHT and EBD. For each line pattern, it searches for a possible match pattern in LPHT. Then SLEBD identifies Event Blocks from the line pattern list by matching with the patterns stored in EBD.

SLEBD uses a stack of line patterns to record the number of log lines that are received and processed for each node. If a line pattern is the start line of an EB pattern, then the block ID and log line number are pushed into the stack. If the line pattern is the end of a block pattern, then the block ID stored on the top of the stack is popped out and compared with the line pattern's block ID. If they match, write the block ID and the log line numbers from the start to the end to an extracted EB list.

The reason for using a stack to identify event blocks is that EBs can be nested, that is one EB happens inside another EB. If such nesting is detected, keep the outer block, and remove the inner block from the output EB list.

When finding that the end line pattern does not match with the block at the top of the stack, or processed all line patterns have but some start line patterns remain in the stack, SLEBD detects an anomaly.

3.4. Performance Evaluation

We have implemented a prototype software of SLEBD and conducted experiments using Mutrino's system logs. The Mutrino system [61], situated at Sandia National Laboratories, is a Cray XC40 system with 118 nodes. It is a test environment of the Trinity production supercomputer [36], the 7th most powerful supercomputer in the world. The dataset that we use contains all syslogs collected from 2/11/2015 to 6/13/2016 on Mutrino data[3].

We use console logs in our experiments. In total, there are 553 console logs in the dataset. Because each console log is composed of mixed messages from all 118 nodes, we select the first 300 days as our test set and assign two days as the size of the inspection time window. After log preprocessing, SLEBD has generated 150 directories, and each one contains 118 node-wise log files.

We conduct our experiments in two steps:

1. Performing EBD generation on the part of the preprocessed logs;

2. Performing EB extraction on the rest of the logs using the generated EBD. After extract an EB list, SLEBD detects anomalies where system behavior is different from the normal.

3.4.1. Experiment Settings and SLEBD Configuration

Table 3.6 lists the configuration of the servers where we run SLEBD.

CPU model	*Intel Xeon X*3460 2.8*GHz*
Core count	8
OS	*Linux CentOS* 7.3.1611
Memory	32*GB*

Table 3.6. EXPERIMENT SETTINGS

In our experiment, a "valid line" refers to a log message which has at least one alphabetic word. Thus, create a line pattern for a valid line.

Two-line patterns are considered to be similar if their similarity ratio is above the threshold. In our experiment we set the threshold k as 67%, which means that if two line patterns, for example, each has three words, have at least two words and their positions in the patterns the same, then we can consider these two line patterns have the same line pattern.

We describe the forward block detection range in Section 3.3.1. In our experiment, we set this range to 4. We will explain the reason for this setting in Section 3.4.4.

In our experiments, when SLEBD analyzes one-period directory, all line patterns whose transition probabilities in FPTM and BPTM have a dramatic change compared with previous records are added to the Reconsidered Pattern List (RPL).

3.4.2. Generated EBD from Mutrino Logs

We present the experimental results from the Mutrino logs in Table 3.7

From Table 3.7, we can see that SLEBD detects in total 2,884 types of line patterns from the Mutrino log set. 608 of them are included in the EB patterns. 409 of them occur only once in all test files, and they are not consolidated into EB patterns. 949 of them appear less than five times.

Number of log messages	2,817,016
Number of invalid lines (removed by preprocessing)	67,478
Number of line patterns	2,884
Number of line patterns included in event blocks	608
Number of event block patterns	189
Number of line patterns occurring only once	409
Number of line patterns occurring less than five times	949

TABLE 3.7. MUTRINO EXPERIMENT RESULTS

SLEBD detects 189 EB patterns in total, converted 2,817,016 lines of the original logs into 1,690,391 events, including both event blocks and single-line messages.

3.4.3. Reconsidered Line Patterns and Relationship Changed Line Patterns

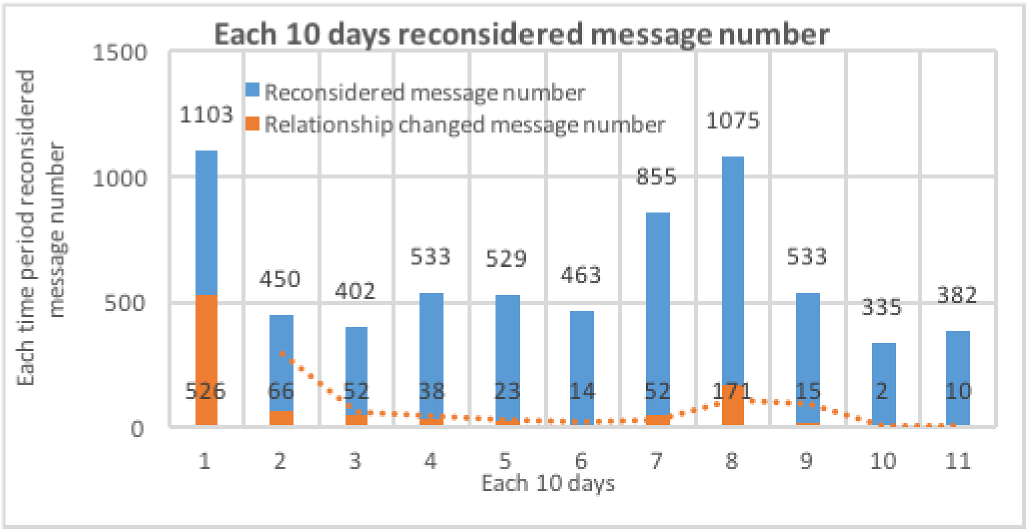


FIGURE 3.3. Reconsidered line patterns and updated pattern relation

Figure 3.3 shows the number of patterns which are reconsidered, and the relation is updated. We use all logs of the first 110 days in the experiment. The blue bars present the size of RPL, that is the number of line patterns that need to be reconsidered. The orange bars inside blue bars show the number of line patterns whose relationship with other patterns are updated.

From Figure 3.3, we can see:

1. For the first ten days, SLEBD produces 1103 line patterns and 526 patterns' relation. As the EBD is not established, SLEBD considers all newly detected line patterns.

2. Bars 2 to 6 show the size of RPL is around 500. The number of relationship updates becomes less. The reason is that the relation between line patterns becomes more stable.

3. For Bars 7 and 8, the number of reconsidered patterns and relationship updates suddenly increases. We check the original system logs and find that from Day 78 to Day 82, the system had a software update, which made the system produced many new line patterns which had not been seen before. SLEBD effectively reconsiders line patterns and updates relationship focusing on newly detected line patterns.

This result shows SLEBD is adaptive and it only reconsiders a subset of line patterns. For example, on Date 51 (i.e., the first day of Bar 6) SLEBD detects 1500 types of line patterns. As the RPL is generated, SLEBD only reconsiders 463 line patterns, which saves a considerable amount of computation. Meanwhile, we can see only 14 line patterns updated for Bar 6. If we carefully select a cutoff threshold, we can reduce the number of reconsidered patterns.

3.4.4. Distribution of EB Pattern Length

Figure 3.4 shows the distribution of the size of the multi-line EBs extracted from Mutrino syslogs. From the figure, we can see 75% of the EBs have a length of 4 or less. However, some EBs have more than 60 line patterns. The largest EB contains 269 line patterns.

After consulting with the system administrator, we find the largest event block that has 269 line patterns corresponds to a system boot sequence which is important for detecting possible system failures and shutdown.

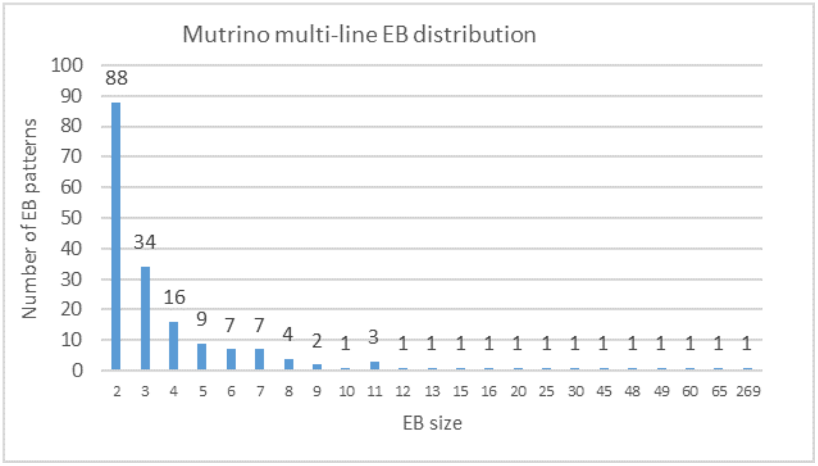


FIGURE 3.4. Mutrino System: Multi-line EB size Distribution

In our experiments, the forward block detecting range is set to 4. From Figure 3.4, we can see most EBs have a length less than 5. Means the 4-line forward range is enough for most line patterns to find other occurring-together line patterns.

3.5. Experimental Results of Even Block Extraction on Baler and SLEBD Line Pattern Lists

I mentioned another syslog message cluster tool Baler [34] in the related work section. Baler is part of Sandia National Laboratories' OVIS project [5]. It is designed to work on HPC's syslog port. Before collecting syslog messages, users need to create a process and designate Baler to listen to the syslog port. Baler also provides 'syslog2baler.pl' program to cluster messages from an existing log file. Baler is used to cluster single line syslog messages and has been deployed on several production HPC systems.

To process log messages, Baler creates a database to store message patterns. This database stores all message line patterns produced during log processing. Additionally, Baler uses a dictionary to identify alphabetic words. A pre-built dictionary is included in the OVIS package. Users can also add new words to the dictionary so that Baler will recognize those words in log messages. The default dictionary has 294,678 words. Baler's dictionary is case sensitive. Thus, each word is stored in three forms, i.e., upper-case, lower-case and the first letter capitalized ones

In this section, we present the experimental results from event block extraction using the line pattern lists generated by Baler and SLEBD on the Mutrino system logs. The objectives of this set of experiments are to evaluate the effectiveness of SLEBD for event block extraction and its sensitivity to different input line pattern lists. We also want to understand how the event block extraction function of SLEBD can enhance Baler for log mining and system behavior analysis.

Our experiments use the 22-day log files collected on Mutrino. The test set contains a total of 613,700 lines of log messages.

Number of single − line patterns	4,980
Number of single − line patterns which happen only once	1,643
Number of extracted event block patterns	226
Total number of events	302,983
Log compression rate	2.02
Number of detected anomalies	1,027
Processing time	3 *hr* 12 *mins*

Table 3.8. EVEN BLOCK EXTRACTION ON BALER'S LINE PATTERN LIST

Table 3.8 and Table 3.9 show the results from event block extraction on the line pattern lists produced by Baler and SLEBD. From the two tables, we can see that

1. SLEBD generates a less number of single-line patterns than Baler, which is 1,424 vs. 4,980.

2. SLEBD identifies a less number of line patterns that occur only once than Baler, which is 188 vs. 1,643. Those line patterns do not contribute to event block extraction as they cannot be grouped with other line patterns into event blocks.

3. By using the line pattern list produced by SLEBD, a less number of event blocks is extracted compared with that using the line pattern list by Baler, which is 141 vs. 226.

Number of single − line patterns	1,424
Number of single − line patterns which happen only once	188
Number of extracted event block patterns	141
Total number of events	206,694
Log compression rate	2.97
Number of detected anomalies	297
Processing time	26 *mins*

Table 3.9. EVEN BLOCK EXTRACTION ON SLEBD'S LINE PATTERN LIST

4. The log compression rate by using the SLEBD line pattern list is higher than that using the Baler line pattern list that is 2.97 vs. 2.02. The log compression rate is calculated as the ratio of the total number of raw log messages to the total number of events extracted.

5. With the line pattern list by SLEBD, the overall processing time is shorter, which is 26 mins using the SLEBD line pattern list and 84 mins using Baler pattern list.

In addition to the preceding results, we observe that 56 event blocks extracted from Baler's line pattern list and SLEBD's line pattern list the same, which accounts for 24.8% and 39.7% of the event blocks identified by Baler and SLEBD respectively.

As Baler generates more single-line patterns than SLEBD, more combinations of line patterns are considered in event block extraction. Some event blocks identified by SLEBD are split into multiple smaller event blocks in Baler's output. That is why more event blocks are produced by using Baler's line pattern list.

By using the event blocks identified by Baler and SLEBD line pattern lists, we detect more anomalies from the former than the latter. After a further study of those anomalies, we find that 197 anomalies from both sets are the same. More interestingly, one type of anomaly in Baler's results appears 450 times, which is not detected in SLEBD's results. That is because more event blocks are found from Baler's line pattern list and some anomalies associated with one type of event block in SLEBD become associated with multiple types of event blocks in Baler's results, which leads to more anomalies for Baler. Thus, a large number of line patterns and event blocks detected from Baler makes anomaly detection more complicated and less effective.

Experimental results on the Mutrino system logs show that we can utilize Baler to preprocess mixed system logs, and Baler causes extra overhead from message grouping and makes event extraction more complicated.

3.6. Summary

SLEBD has the following attractive features.

1. It converts millions of unstructured log messages into concise and structured event block lists, which facilitates system monitoring and behavior analysis.

2. It generates an event block database (EBD) from the original system logs. System administrators can use EBD to process message streams in real-time.

3. It updates EBD by continuously analyzing multiple time-periods of log files. Thus, EBD evolves as the monitored system changes.

4. It analyzes system and component events to identify anomalies and enable fault diagnosis.

The generated event blocks capture the major behavior of an HPC system and facilitate anomaly detection. By transforming the original mixed messages into clear event lists, system administrators can save their time and efforts to focus on analyzing high-level event distribution and relation with richer semantic information.

To further improve SLEBD, we are currently developing a machine learning based approach to study system behaviors using EB lists. We plan to make the EBD generation process more efficient and automatic. We also consider using FPGAs and ASICs to accelerate log analysis and event block extraction. We also plan to integrate SLEBD with our previous works and tools for performance anomaly detection [39] [40], failure prediction [90], power management [41] [42], and soft error analysis [38] to build a system monitoring and analytics framework, where resilience, performance, and power are managed in an integral way.

CHAPTER 4

TOPOLOGY-BASED SEQUENCE MINING METHOD

4.1. Introduction

Sequence mining is a method of discovering frequent sequential patterns from a set of objects. SLEBD can group log messages that happen together into event blocks by using probability theories. In this way, the sequence between all happening together single line messages which belong to the same atomic operation is captured. This feature provides us an opportunity to perform sequence mining to analyze the System Event List and find connections between different types of System Events.

As an HPC system may have hundreds (and even more) of types of System Events and could generate tens of thousands of System Events every day, the number of the target objects can be huge. The occurrence time of each type of System Event in one System Event List can be highly different. The traditional Apriori-like methods cannot meet our request. Even the most recently developed and most efficient PrefixSpan [84] algorithm requires a large number of compare and merge operations. Besides this, PrefixSpan generates all happened subsequences and marks how many times these subsequences occurred. The number of all the subsequences generated from even one very simple System Event List could be larger than the System Event List itself. If we want to use F/P algorithms to find HPC system event execution sequences, we need to generate all subsequences and do additional work to filter useless results. That is unnecessary for us and will make the result analysis difficult.

As our study on HPC system found, most HPC system operations always have multiple steps, and these operation steps are following a principle. So, in a short time window, such sequences will become Directed Acyclic Graphs [60]. This finding inspired us to design a Topology [49] based Sequence Mining algorithm (TSM). It can detect all possible longest sequences from sequence Mining Object in just seconds. Our algorithm only needs to scan the sequence Mining Object once to collect temporary data and do not need a large amount of compare and merge operation to grow a sequence. And our algorithm only requires a very small space to store our result and temporary data.

4.2. Mining Event Patterns Based on Topology

4.2.1. Topology Position between Events in Sequences

As we find, there are five types of position status between items in sequences:

1. **Prior:** Item_A always occurs prior to Item_B.

2. **After:** Item_A always occurs after Item_B.

3. **Wrapping:** Item_A always occurs both prior to and after Item_B. It looks like Item_A wrap Item_B.

4. **Wrapped:** Item_A always occur between two Item_B. It looks like Item_B wraps Item_A.

5. **Tangling:** sometimes Event_A occurs prior to Event_B, and sometimes Event_A occurs after Event_B.

By analyzing the target object, we find sometimes Item_A occurs prior to Item_B, but sometimes Item_A occurs after Item_B. We cannot make our decision of Item_A's position status with Item_B. At this time, we can only say Item_A is tangling with Item_B. As Item_A and Item_B are tangling, they cannot exist in the same sequence pattern.

For example:

Sequence ID	*Sequence*
Seq_1	$[A, B, C, D, E, C]$
Seq_2	$[B, A, C, D, E, C]$
Seq_3	$[B, A, D, E]$
Seq_4	$[B, C, D, E, C]$

TABLE 4.1. MINING OBJECT EXAMPLE 4.1

From these sequences, we can find the following position status:

1. Items A & B are prior to items C, D & E.
2. Items C, D & E are after items A & B.
3. Item C is wrapping items D & E.
4. Item C wraps Items D & E.
5. Item A and item B are tangling.

4.2.2. Generate Prior Transition Matrix and Position Status Matrix From Object Sequences

TSM scans sequences once and produces a Prior Transition Matrix (PRTM). Then it uses PRTM to generate a position status matrix.

Prior Transition Matrix (PRTM)

An entry in PRTM, for example, PRTM[A][C], indicates the relationship of Event_A with Event_C. It has three elements: Prior Count (PC), Ready Switch (RS), and Occurs in Same sequence Count (OSC). Note PRTM[A][C] is not the same as PRTM[C][A]. TSM creates a PRTM based on a list of events of interest provided by users and initializes all events' PC and OSC value to 0 and RS to "Not ready."

When scanning a sequence, TSM performs the following operations for each event, denoted by event_n.

1. Set PRTM[n][k][RS] to "Ready". Here "k" refers to another event, event_k.

2. Look up another event_k in PRTM. If PRTM[k][n][RS] is "Ready", then add "1" to PRTM[k][n][PC] and change PRTM[k][n][RS] to "Not ready";

3. Use a counter to count how many times event_n appears in the sequence.

After scanning the sequence, TSM updates events' OSC in PRTM with events' occurrence counts. For example, if Event_A occurs once in Sequence_1 and Event_C occurs twice in Sequence_1, then after scanning sequence_1, TSM adds 1 to PRTM[A][C][OSC] and adds 2 to PRTM[C][A][OSC].

Generating the position status matrix(POSM) from PRTM

Let's look at the PRTM that we have generated from object Example 4.1:

Each entry in the PRTM contains [PC, OSC]. To analyze the position relation of two events (say event_n and event_k) position, we need to define a support count. A support

	A	B	C	D	E
A		$[1,3]$	$[2,2]$	$[3,3]$	$[3,3]$
B	$[2,3]$		$[3,3]$	$[4,4]$	$[4,4]$
C	$[0,4]$	$[0,6]$		$[3,6]$	$[3,6]$
D	$[0,3]$	$[0,4]$	$[3,3]$		$[4,4]$
E	$[0,3]$	$[0,4]$	$[3,3]$	$[0,4]$	

Table 4.2. PRTM GENERATED FROM MINING OBJECT EXAMPLE 4.1

count is assigned by the lower OSC of PRTM[n][k] and PRTM[k][n] and is used as a threshold to tolerate noise. Then we use PRTM[n][k][PC] and PRTM[k][n][PC] to compare the support counts to understand their positions.

1. If PRTM[i][j][PC] is greater or equal to support count but PRTM[j][i][PC] less than the support count, it means item_i is prior to item_j and item_j is after item_j.

2. If both PRTM[i][j][PC] and PRTM[j][i][PC] are all greater or equal to the support count, it means item_i and item_j are both prior to each other. Now which item's OSC is bigger, which one is wrapping the other one.

3. If both PRTM[i][j][PC] and PRTM[j][i][PC] are all greater or equal to the support count and PRTM[i][j][OSC] and PRTM[j][i][OSC] are equal, then the item whose PC is bigger is prior to the other one. That is because the support count ratio threshold is set too low. We will discuss this situation later.

4. If both item_i and item_j 's PCs cannot reach the support count, it means they are tangling.

Now consider the example in Table 4.2. For event_A and event_B, their support count is 3. However, PRTM[A][B][PC] = 1 and PRTM[B][A][PC] = 2, i.e., both of them are less than the support count. So event_A and event_B are tangling. For event_A and event_C, their support count is 2 (PRTM[A][C][OSC]), PRTM[A][C][PC] = 2, however, PRTM[C][A][PC] = 0. Then event_A is prior to event_C and event_C is after event_A. For event_C and event_D, their support count is 3 (PRTM[D][C][OSC]), and PRTM[C][D][PC] = PRTM[D][C][PC] = 3. Thus, wrapping is the position relation between event_C and event_D. As PRTM[C][D][OSC] = 6, which is greater than PRTM[D][C][OSC], event_C wraps event_D (or event_D is wrapped by event_C).

Users can assign two thresholds to tolerate noise:

1. Minimal occurrence count threshold for identifying rare events. If PRTM[n][k][OSC] is less than the minimal occurrence count threshold, it means the occurrence count of event_n and event_k in the same sequence is very small.

2. A support count ratio threshold. For instance, assume the support count of event_n and event_k is 100 and the threshold as 90%. If we observe event_n is prior to event_k for 90 times, then we say event_n is always prior to event_k.

By applying the preceding rules, we build a position status matrix (POSM), as shown in Table 4.3.

	Prior	*After*	*Wrapping*	*Wrapped*	*Tangle*
A	C, D, E				B
B	C, D, E				A
C		A, B	D, E		
D	E	A, B		C	
E		A, B, D		C	

Table 4.3. POSM GENERATED FROM TABLE 4.2

Growing sequence patterns from position status matrix by using Topology

Based on the position relation among events, we can build event patterns. Our topology-aware event pattern building process works as follows.

1. We use a Waiting Item List (WIL) to store all user-interested items. Use a Temporary Sequence List (TSL) to store all temporary sequence patterns.

2. In each round, we randomly select one Waiting Item (WI) from WIL and remove this WI from WIL.

3. If TSL is empty, append length-1 sequence [WI] into TSL. Then make all this WI's Tangling Items (WITI) from this WI's tangle list as length-1 sequence [WITI]. Append sequence [WITI] into TSL.

4. If TSL is not empty, we compare WI with all Temporary Sequences (TS) in TSL.

a) If WI is already in the TS or WI tangles with some event in the TS, then continue.

b) If WI can be inserted into the TS based on POSM, we add a new sequence [TS-WI] to TSL and add the WI's Tangling Events (WITI) to the TS. If any WITI is successfully added to the TS, we add a new sequence [TS-WITI] to the TSL.

c) WI cannot be added to any TS in TSL, add length-1 sequence [WI], and all length-1 sequences [WITI] to TSL.

5. Repeat Steps 2 to 4 until WIL is empty.

6. If any TS in TSL has events with Wrapping/Wrapped position, we add one more wrapping event to the TS. The insertion position is after all events which have "Wrapping" or "After" position with it.

The TSL that is generated from POSM contains all of the long event patterns which events of interest are included.

The topology-aware sequence mining algorithm scans events only once and does not generate a large number of temporary subsequences. Unlike FreeSpan or PrefixSpan, TSM uses tangling tags to determine if two events can be placed in the same sequence or not before generating any Temporary Sequence (TS). If a Waiting Event (WI) can be added to a Temporary Sequence, all Waiting Tangling Events (WITI) cannot be added to the sequence generated from TS and WI ([TS-WI]). We also note all WITIs may be added to TS. That is why we try to add WITI to TS and generate another sequence [TS-WITI]. In this way, TSM can reduce a large number of subsequence comparison and merging operations.

TSM performs some matrix search and comparison operations. It needs to search WI's position status with events in the TS from Position Status Matrix (POSM) to find the insertion position in the TS. When a new TS is added to TSL, TSM needs to compare this TS with existing TSes in TSL to see if this TS already exists. Such search and comparison operations do not happen very often, and they are not computing-intensive.

4.3. Generating Event Patterns – Case Study

In this section, we use an example to illustrate how TSM builds event patterns, which is described in Section 4.2.1. In the example shown in Table 4.1, there are five events: A, B, C, D, and E. The Wait Event List (WIL) includes these five events.

4.3.1. Using Event Order in WIL to Process WIs

Step 1: processing event A.

As TSL is {}, TSM adds pattern [A] and A's tangle event [B] to TSL. Thus, TSL = {[A], [B]}

Step 2: processing event B.

TSM tries to add B to each TS in TSL. As B tangles with A and pattern [B] already exists. TSM does not change TSL. Still TSL = {[A], [B]}

Step 3: processing event C.

TSM tries to add C to [A] and [B]. As C is after both A and B, C can be appended to [A] and [B] as [A, C] and [B, C]. TS [A] and [B] are then removed from TSL. So, TSL = {[A, C], [B, C]}

Step 4: processing event D.

D is after A and B, and D is wrapped by C. TSM discovers two new patterns [A, C, D], [B, C, D]. Then [A, C] and [B, C] are removed from TSL. Now TSL = {[A, C, D], [B, C, D]}

Step 5: processing event E.

E is after A, B, and D, and wrapped by C. TSM appends E to each TS in TSL. So, TSL = {[A, C, D, E], [B, C, D, E]}

4.3.2. Processing and Adding Wrapping Events to TS.

As patterns [A, C, D, E] and [B, C, D, E] have event C and its wrapping events D and E in them. TSM adds event C to the two patterns. The insertion position is after D and E. Thus, TSL becomes:

{[A, C, D, E, C], [B, C, D, E, C]}

4.3.3. Extracting Subsequences From TSL to Match Event Sequences

As we discuss in Section 4.2.2, the topology-aware event sequence building method, TSL contains all long event patterns. However, some patterns may not include events of interest. Thus, those patterns may not be able to match with event sequences.

In the preceding example, the set of events that appear in event sequences are different, as shown in Table 4.4.

$Sequence\_id$	$EventSequence$	$EventSet$
Seq_1	$[A, B, C, D, E, C]$	A, B, C, D, E
Seq_2	$[B, A, C, D, E, C]$	A, B, C, D, E
Seq_3	$[B, A, D, E]$	A, B, D, E
Seq_4	$[B, C, D, E, C]$	B, C, D, E

Table 4.4. EVENT SETS OF EVENT SEQUENCES

From Table 4.4, we can see two event patterns in TSL cannot match with Sequence_3. On the other hand, we can use Sequence_3's event set {A, B, D, E} to extract two subsequences, that is [A, D, E] and [B, D, E]. Thus, the event pattern list becomes:

{[A, C, D, E, C], [B, C, D, E, C], [A, D, E], [B, D, E]}

4.4. Event Pattern Verification

TSM can significantly improve the efficiency of event pattern mining and reduce computation overhead. All events are important. TSM randomly selects one WI from WIL to process each time, which makes some WIs processed early while other WIs are processed late.

To make sure the event patterns that TSM produces are correct. We verify the following three requirements.

1. All event patterns are fully generated. In other words, no more events can be added to any event patterns.

2. Event patterns are independent of the order in which WIs are processed.

3. Event patterns should not conflict with the sequences that are learned. If one sequence does not contain all events that an event pattern possesses, then a subsequence of the event pattern must be capable of matching the sequence.

Correspondingly, we develop three functions to verify the preceding requirements.

1. Try to add all events from WIL to all TS in TSL. If any WI from WIL can be added to any TS, then the TS is not fully generated.

2. Manually assign different orders for processing WIs and apply these orders to build TSL. Then compare every two TSLs. If one TSL has some event patterns that the other TSL does not have, then event patterns are affected by the processing order.

3. Use event patterns in TSL to map event sequences. Scan an event pattern and subsequences extracted from it and record how many times one event pattern and its subsequence can be matched from the first event to the last one.

Each event pattern has one event which has a minimal occurrence count in sequences than other events in this pattern. If this event pattern's successful occurrence count is no less than the minimal event's occurrence count, we conclude this event pattern is correct.

4.5. Performance Evaluation

4.5.1. Experiment Setting and Test Cases

We have implemented a prototype software of TSM and conducted experiments using Mutrino's system logs. The Mutrino system [61], sited at Sandia National Laboratories, is a Cray XC40 system with 118 nodes. It is a test environment of the Trinity production supercomputer, the 6th most powerful supercomputer in the world. The dataset that we use contains all syslogs collected from 2/11/2015 to 6/13/2016 on Mutrino.

We apply our recently developed System Log Event Block Detection (SLEBD) tool [63] to extract event blocks from groups of log messages, and convert the raw log messages to event blocklists. We randomly select 18 nodes' event lists for the same period (two days). Among the 18 event list files, the shortest one contains 545 events, and the longest one has 970 events. Table 4.5 lists the number of events and events of interest in our test cases.

Number of system events	13156
Number of interested common event types	132
Number of interested event from test case	8981
Shortestfilesize	545
Longestfilesize	970

Table 4.5. EXPERIMENT TEST CASE INFORMATION

We run experiments on compute servers, each of which is equipped with Intel Xeon X3460 (8 cores, 2.8GHz) and 32 GB DRAM, and runs CentOS 7.3.1611.

4.5.2. Verification Results

We run TSM on the 18 event lists and analyze 8981 events of interest. TSM builds 364 event patterns. The longest pattern contains 84 events, and the shortest pattern includes 75 events.

We test the three requirements, as described in Section 4.4, to verify the correctness of our experimental results. We find that no event of interest can be added to any of the 364 event patterns, all of the 364 patterns match with our 18 event lists, and their matching counts are equal to or greater than the minimal-occurrence event's occurrence count. The event pattern that has the smallest matching count succeeds 33 times. We run TSM 10 times, and a random event processing order is used each time. The ten runs produce the same (364) event patterns.

4.5.3. Performance Results

Factors, such as event list size and the number of events of interest, may influence the performance of TSM, specifically, the building speed of event patterns. We conduct a set of experiments to evaluate the influence. We randomly select 100 events of interests out of the 132 common events. For comparison, we also two event list sets, one of which contains eight lists randomly selected from the 18 lists, and other set contain the remaining lists.

All TSL event patterns generated from the experiments are tested, and they all pass the verification test as described in Section 4.4. Table 4.6 & 4.7 shows the time that TSM uses to create PRTM, POSM and build event patterns.

Seq. size	*Event of interest*	*Event count*	*Seq. analysis and PRTM gen time(s)*
18	132	8981	1.11
18	100	7031	0.68
5	132	2127	0.34
10	132	4961	0.66

Table 4.6. EXECUTION TIME OF TSM I

POSM gen time(s)	*TSL build time(s)*	*TSL size*
0.165	2.28	364
0.08	0.29	33
0.16	1.85	378
0.16	1.67	296

Table 4.7. EXECUTION TIME OF TSM II

For ease of our discussion, we use "F_i_I_j" to denote the set of an experiment with a number of i event sequences and several j events of interest. For example, F_18_I_132 processes 18 event sequences and 132 events.

From Table 4.6 & 4.7, we find the POSM generation time of F_18_I_132, F_5_I_132, and F_10_I_132 is around 0.16 second. F_18_I_100 processes only 100 events of interest, but its POSM generation time is 0.08 second. Additionally, the TSL build time in the three experiments (F_18_I_132, F_5_I_132 and, F_10_I_132) are close to 2 seconds, and their sizes of TSL are close. The TSL build time of F_18_I_100 only takes 0.29 second. These findings show TSM's execution time is influenced more by the number of events of interest than the sequence length.

We also notice that F_18_I_100 only produces 33 event patterns. After further analysis, we find that there are 57 events among the 100 event types whose tangling lists are empty. Among the 132 event types, 67 events have empty tangling list. To further understand the relationship between event patterns and events' tangling lists, we conduct the following experiment.

Seq. size	*Event of interest*	*Event count*	*Seq. analysis and PRTM gen time(s)*
18	100	7543	1.13

Table 4.8. EXECUTION TIME FOR NEW 18 SEQUENCES AND 100 EVENTS I

POSM gen time(s)	*TSL build time(s)*	*TSL size*
0.160	0.92	364

Table 4.9. EXECUTION TIME FOR NEW 18 SEQUENCES AND 100 EVENTS II

After TSM generates POSM from the 132 events, we create a new event list which has 65 (i.e., 132 - 67) events whose tangling lists are not empty. We randomly select 35 events whose tangling lists are empty to create a 100-event list and build event patterns. Table 4.8 & 4.9 presents the results. In the table, we find the sequence analysis time, PRTM generation time, and POSM generation time are the same as those for F_18_I_132. That is because TSM uses the same event set to create POSM. Then the 100-event list is selected from POSM. We also find the size of TSL built from the 100-event list is the same as that in F_18_I_132. Each event pattern in F_18_I_100's TSL matches with a corresponding longer event pattern in F_18_I_132's TSL and the former pattern is a subsequence of the latter pattern.

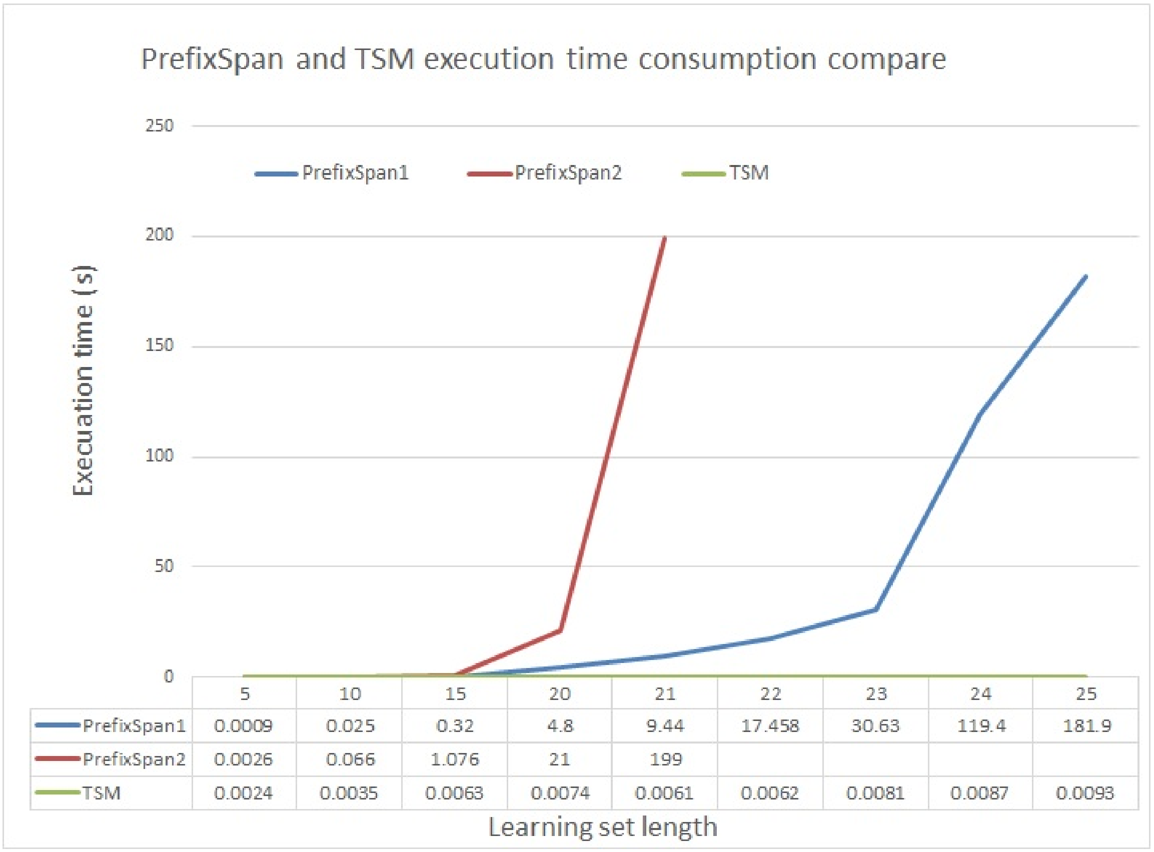
If TSM generates event patterns among events with an empty tangling list, only one pattern is produced. We call it common sequence. The difference between F_18_I_132 and F_18_I_100 is that F_18_I_132's common sequences are generated by 67 events with an empty tangling list, while F_18_I_100's common sequences are generated by 35 events. F_18_I_100's common sequences are subsequences of that of F_18_I_132. If TSM builds event patterns among the 65 events with non-empty tangling lists, 364 subsequences are generated, which is the case for F_18_I_132 and F_18_I_100. The TSLs generated from F_18_I_132, and the F_18_I_100 contain event patterns merged from the 364 subsequences and the common sequences. That explains why an event pattern from F_18_I_100 matches with a corresponding longer pattern from F_18_I_132. Thus the reason that only 33 event patterns are built-in F_18_I_100 is that the randomly selected 43 events with non-empty tangling lists can only generate 33 subsequences patterns.

We also note that even though F_18_I_132 and F_18_I_100 have the same set of events with non-empty tangling lists, the time for building their TSLs varies much. That is because the size of an event set influences the TSL build time. Adding events with an empty tangling list still takes time.

4.5.4. Performance Comparison with FreeSpan/PrefixSpan Algorithm

We download two PrefixSpan programs in Python from Github [6] [4]. We also implement TSM in Python. We run the three programs under F_18_I_132. The two downloaded programs crash before completion.

To get some useful results from the programs, we extract smaller test cases to evaluate the performance of PrefixSpan and TSM. We randomly select three-event sequences from the 18 sequences and extract 5, 10, 15, and 20-25 events from the beginning to create new test sets. Figure 4.1 shows the execution time of the three programs, i.e., two downloaded PrefixSpan programs and our TSM implementation.



	5	10	15	20	21	22	23	24	25
PrefixSpan1	0.0009	0.025	0.32	4.8	9.44	17.458	30.63	119.4	181.9
PrefixSpan2	0.0026	0.066	1.076	21	199				
TSM	0.0024	0.0035	0.0063	0.0074	0.0061	0.0062	0.0081	0.0087	0.0093

FIGURE 4.1. PrefixSpan vs. TSM: execution time comparison

From Figure 4.1, we can see PrefixSpan1 takes 200 seconds when the size of the test set is 21, and PrefixSpan2 takes 181 seconds for 25 events. In contrast, TSM's execution time is always less than 0.01 seconds. The figure shows the size of the event set increases, the execution time of PrefixSpan increases sharply, making PrefixSpan unsuitable for processing the overwhelming number of events on production HPC systems.

PrefixSpan1 and PrefixSpan2 extract 751,615 length-1 to length-21 subsequences.

Then we use a subsequence filtering program to remove subsequences that are too short or do not include untangling events of interest. Two long event patterns are produced, one of which is length-12, and another is length-13. TSM generates the same event patterns. Therefore, TSM is functionally correct but much more efficient than PrefixSpan.

As the two downloaded PrefixSpan programs crash even for a small event set, we implement PrefixSpan in Python by ourselves. We run our PrefixSpan code for F_18_I_132, and it takes 1.34 seconds to generate all length-2 subsequences and 245 seconds to generate all length-3 subsequences. We ran the program for 15 minutes hoping to get length-4 subsequences. However, the results did not show up, and so we terminated the execution.

4.5.5. Comparison with FP Growth

FP growth method [76] mines frequent pattern by using a tree structure which can avoid using the apriori method and reduce execution time.

We conduct an experiment to study the FP growth method. We use a Python-based FP growth library [1] and two test cases from event lists produced by SLEBD.

The first test case has three log files collected from three nodes, and each file has 50 events. We use the FP growth method to mine frequent patterns in this case by assigning the frequent threshold as 3. The execution time is 3.5 seconds, and it mines 15,365 frequent patterns in total.

The second test case also has three files, and each file contains 100 events. This time FP growth crashed.

The experimental results show FP growth is more efficient than the Apriori algorithm. However, we find FP growth has the following issues.

1. FP growth does not care about the ordering between learned items, which means the frequent patterns learned by FP growth do not have sequential/order information.

2. FP growth scans the mining object sequences first and creates an FP-tree. Each frequent item from the mined object appears in at least one node in the FP-tree. FP growth generates nodes in the FP-tree according to the occurring count of each item with one frequent prefix in the same mined object sequence. That means if the number of frequent

items and the number of mined objects is large, there could be a large number of branches in the FP-tree.

3. Each time FP growth generates a frequent pattern, it searches from the FP-tree's root node. If the number of frequent items is large, the depth of FP-tree is also deep, which makes the search time longer.

We cannot use FP growth in our research because the HPC system events are executed in approximate order. The sequence mining method used on the SLEBD converted event lists is to find such execution orders or sequences. FP growth could not produce sequential/order information.

4.6. Summary

As we find, the system events appear in approximate orders. Some events may repeat several times in a period, which makes sequence mining non-trivial.

The main difference between TSM and the apriori-based method is TSM's execution time depends on the number of system events, while the execution time of Apriori-based methods depends on the length of possible event sequences. The repeating events in the event lists make the Apriori-based algorithms' execution time become longer.

TSM uses topology information to grow longer sequence patterns where events of interest are included. TSM is computation and storage efficient and suitable for detecting sequence patterns with long chains from subsequences, which are generated by multiple steps of sequence mining. TSM scans events only once, which assures that its performance is not affected by the number of events. TSM can generate all sequence patterns much more quickly than other existing methods.

TSM achieves a better performance than FreeSpan and PrefixSpan. However, if users expect to count the total number of times that all subsequences occur, TSM may not be highly efficient. On the other hand, if the goal is to perform sequence mining on system events to discover the relationship among events and the execution sequences of a system, TSM is a good choice. With the generated sequence patterns discovered, detecting anomalies in the execution sequences become effective. Furthermore, event prediction and system behavior analysis becomes feasible.

CHAPTER 5

ANOMALY DETECTION AND EVENT PREDICTION

5.1. Introduction

SLEBD models and characterizes millions of messages from system logs by using only a limited number of event blocks. System operators can concentrate on analyzing the distribution and dynamics of EBs, which facilitates their understanding of system behavior and identifying anomalous behaviors. By analyzing the event block sequences, we can detect several kinds of anomaly events which have a relationship with one type of event pattern.

Recurrent Neural Network (RNN) [33] is an artificial neural network. The output of an RNN depends not only on the current input but also on the previous outputs. This feature makes RNN applicable to a variety of applications, such as speech recognition [79] [80], handwriting recognition [21] [69], and text recognition [68] [26]. Recurrent Neural Network (RNN) is capable of learning and predicting sequential data.

Long Short Term Memory (LSTM) [52] is an RNN in which some important input data, even far away from the current input, can affect the output. When analyzing long sequence such as long articles, RNN's performance on connecting previous information to the present task is not as good as that of LSTM. Because there are gaps between the relevant information and the locations where it is needed in log articles, RNN may not be able to learn the connections between the important information. In contrast, data that is close to the current input but is less important may not affect the output.

When we study LSTM, we explore a case study on using LSTM to cluster commercial records of customer reviews [83]. Each review is a sentence which has around 8 to 1000 words, and in the original records, there are 11 clusters. We selected the most seven frequent occurs clusters and select the top 4000 frequent words as interesting words and divide the whole commercial review set into two sets, learning set and testing set. And we generated an LSTM model from the learning set. Then use this LSTM model to predict the cluster of each review in the testing set. Their prediction accuracy is always more than 88%. From this experiment, we've found LSTM can cluster long sequences by analyzing items in a designated range from these sequences and able to predict a future event belongs to which event.

This LSTM case inspires us. From SLEBD experiment we can find, all the anomalies which SLEBD has detected are related to one specific Event Block. This feature provides us an opportunity to cluster event sequences before all such kind of Event Block/anomaly's occur position.

In our experiment of learning and testing LSTM models, we encounter a problem. All the sequences we have collected are constituted by system events before success or failed event which position in one Event Block list is already detected. However, HPC system events are real timely generating like a stream. Even though SLEBD can collect single line event log and extract Event Block lists from streaming system logs, it cannot predict if one system event is going to happen in the future. So even we can learn LSTM model for one specific system event, we are still unable to predict if this system event is going to happen in the future, and we also don't know which system event sequence we should use to map with this system event's LSTM model.

To solve this problem, we create a testing flow combining SLEBD Event Block patterns, TSM sequence mining results, and LSTM models. This testing flow can help us to predict one specific event's occurrence in the future and use the LSTM model to predict its execution success. In this chapter, we will discuss our method.

I use Python deep learning library Keras[9] to create our LSTM model from HPC system event sequences. As I have found, deep learning algorithms such as RNN require a large number of floating-point computations and require extensive computing resources. Currently, software-based RNN models created by Keras or Tensorflow [15] running on general-purpose processors like CPU or GPU cannot achieve high efficiency. In my experiments, one simple model training which training set have only 779 sequences will cost more than 2 minutes to complete.

The RNN algorithm is complicated, involving multiple types of floating-point computations. With its wide application, especially in real-time and embedded environments, it is imperative to accelerate its execution while limiting energy consumption. There have been many studies that apply GPUs to accelerate deep learning computation [91] [77]. However, the high energy consumption makes them not cost-effective.

Different from CPU and GPU, Field Programmable Gate Array (FPGA) provides a low-power computing environment with massive reconfigurable logics. One FPGA may contain hundreds of thousands of Look Up Table (LUT) logic slices, thousands of Flip-flop units, hundreds of DSP units, and multiple megabytes of on-chip Block RAM (BRAM). LUTs can be configured to implement any combinational logic functions and logic-and-arithmetic operations. DSP units on an FPGA are suitable for floating-point computation. Moreover, multiple tasks can be executed on an FPGA, and they receive input data via the dataflow pipeline[7] [71]. These distinct features make FPGA well suitable for accelerating deep learning applications.

5.2. Anomaly Detection Using Event Blocks

Here we present the benefit of using event blocks for anomaly detection.

5.2.1. Incomplete Event Block Anomaly

Every line pattern that is included in an EB pattern has a tag to identify it belongs to which EB pattern. When extracting the EB list, if SLEBD detects any line pattern happens individually instead of in an EB, this EB is not completely generated. Then SLEBD considers this case as an anomaly.

The cause of such type of anomaly can be that procedure is not executed completely, or there exist errors in the generation of log messages, or some other system or kernel level error happens.

We analyze the anomalies detected by SLEBD from the Mutrino logs. Here is an example.

Block_10 is consisted of two line patterns: LinePattern_255 and LinePattern_256:

[LP_255]: hub 1-0:1.0: USB hub found

[LP_256]: hub 1-0:1.0: 2 ports detected

Example 5. Block_10 Example

In the 300-day Mutrino logs, SLEBD finds Block_10 appears 3,665 times and detects 39 times of anomalies are related with Block_10. A typical anomaly is as follows.

[LP_735]: /scsi/) failed: hub 1-0:1.0: USB hub found

[LP_736]: Not a directory

[LP_256]: hub 1-0:1.0: 2 ports detected

Example 6. Sequence [LP_735, LP_736, LP_256] Example

This sequence occurs 5 times. LinePattern_256 supposed to belong to Block_10 but appears individually. SLEBD reports an anomaly. By searching the event extraction record, we have detected that LinePattern_735 occurs 6 times and LinePattern_736 occurs 5 times. Thus, part of LinePattern_735 is not generated completely. LinePattern_735 contains part of LinePattern_255. We then check the related line patterns, and find LinePattern_502 is as follows.

[LP_502]: udevd-work[5561]: rename(/dev/scsi/.tmp-b8:128, /dev/scsi/) failed: Not a directory

Example 7. LinePattern_502 Example

LinePattern_502 occurs 587 times and it always happens close to Block_10. Here is the symptom of the anomaly.

1. LinePattern_736 contains part of LinePattern_502 (Not a directory)

2. LinePattern_735 contains part of LinePattern_502 **(”/scsi/) failed:”)** and LinePattern_255 **(”hub 1-0:1.0: USB hub found”)**

Compared to LinePattern_502 and Block_10, LinePattern_735 and LinePattern_736 occur less often and their appearance is erroneous. SLEBD detects the anomaly.

This type of mixed-message anomaly happens on many nodes in the 300-day system log that we test. Eighteen event block patterns have the mixed-message anomaly, and SLEBD detects 197 such anomalies. One possible explanation of these anomalies is that messages coming from different components on a node causes a race condition for syslog to record them correctly.

Additionally, SLEBD detects system behavior that is different from the normal. For example:

[LP_558]: waiting for MySQL to accept connections

[LP_559]: Checking SDB version

[LP_560]: SDB version up-to-date (4.10.14)

[LP_561]: Initializing SDB Table

Example 8. Block_95 Example

[LP_558]: waiting for MySQL to accept connections

[LP_1155]: Not initializing SDB on backup sdbnode (65)

Example 9. Sequence [LP_558, LP_1155] Example

In this case, messages in Block_95 do not have the mixed- message anomaly. Block_95 occurs 61 times, and the sequence [LP_558, LP_1155] appears five times. SLEBD detects that LinePattern_558 appears individually instead of being part of Block_95. Block_95 indicates SDB was successfully initialized. However, the sequence [LP_558, LP_1155] indicates the SDB initialization failed.

5.2.2. Anomalous EB Length

SLEBD also uses the length of event blocks to identify anomaly detection. For example, Block_72 contains two messages. In the dataset that we test, Block_72 occurs 167 times, and it contains two messages in 166 times. In one case, however, one time one Block_72 has 852 messages, i.e., from Line #1514 to Line #2365. This situation is an anomaly as the length of this event far exceeds the average length. Certain error happened during that event causing more messages to be included in the anomalous event block.

5.3. Clustering Sequences by Using LSTM

5.3.1. Collecting Event Sequences and Creating Training/Test Sets

As we have introduced in Chapter 3, after SLEBD extracts Event Block patterns and stores these patterns in the Event Block Database (EBD), it can extract Event Block lists from original system logs. When SLEBD is extracting Event Block lists, it also generates a list indicating all anomalies occurring in all Event Block lists and indicate this Event Block pattern related to which anomaly event.

First, we need to assign an Event Block — for example, Block_10 to be our research object. Then we will find all Block_10's success positions and anomalies which are related to Block_10's occurrence positions from Event Block lists and anomalies list. We extract all Event Blocks in a designated range before these positions. In our experiment, the prior range is 500. These event sequences are grouped into two clusters: successful and failed.

The two clusters are not equally sized. For example, Block_10's successful cluster has 939 sequences but failed cluster has only 39 sequences. To solve this bias, we produce a dataset by using the SMOTE[28] method to balance the two sets and divide the produced dataset into a training set (which contains 2/3 of the event block sequences) and a test set (which contains the rest 1/3 of event block sequences).

In our experiment on Block_10, the training set have 624 successful sequences and 31 failed sequences. We repeat failed sequences for five times in training set. The training set has 779 sequences.

5.3.2. Generating the LSTM Model

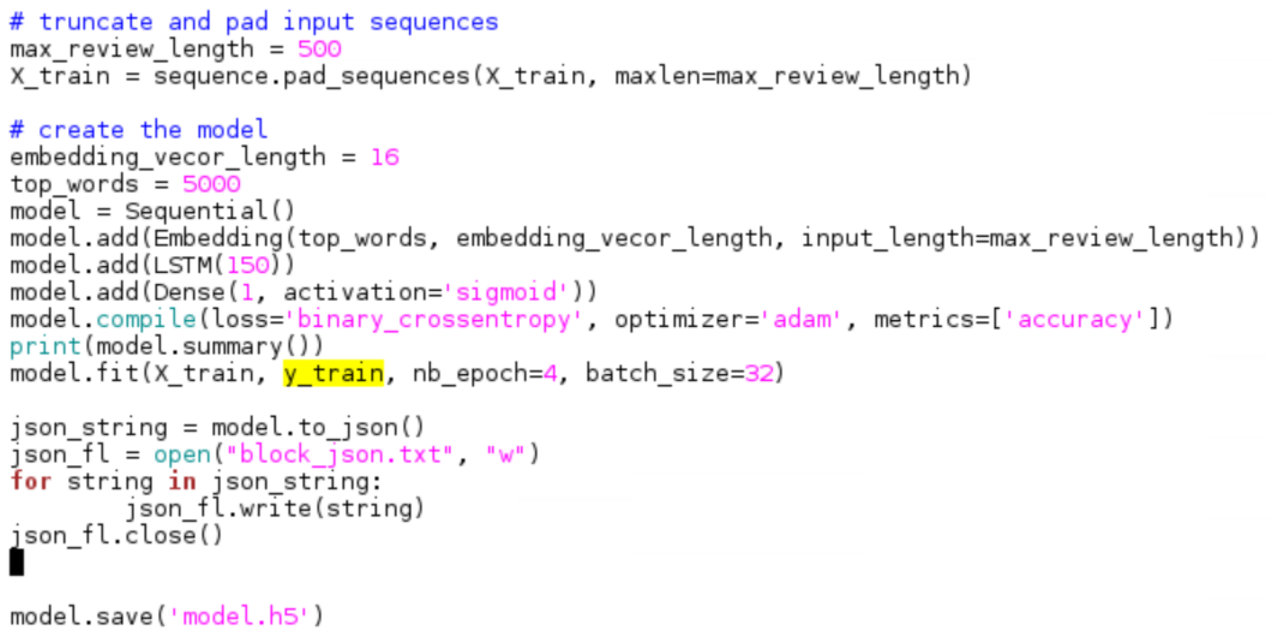
We use python deep learning library Keras[9] to create our LSTM model. The following is our configuration code:

Some event sequences are shorter than 500; we need to pad them into sequences which have length 500.

We run our code on the learning set and store the trained LSTM model into a .h5[2] file.

5.3.3. Testing Results of the LSTM Model

We test the LSTM model stored in the .h5 file with our testing set to test how accurate the LSTM model is. In our experiment on Block_10, the testing set have 312 successful events and eight failed events. The testing result is like this:



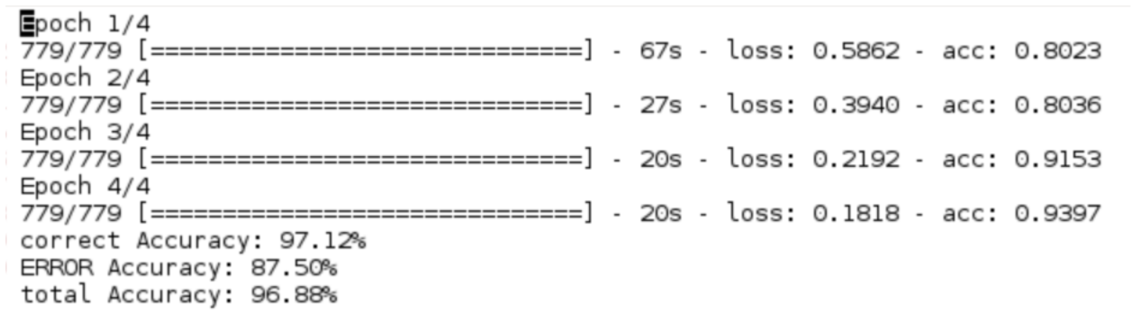
```
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)

# create the model
embedding_vecor_length = 16
top_words = 5000
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(LSTM(150))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, nb_epoch=4, batch_size=32)

json_string = model.to_json()
json_fl = open("block_json.txt", "w")
for string in json_string:
        json_fl.write(string)
json_fl.close()


model.save('model.h5')
```

Figure 5.1. Keras configuration code



```
Epoch 1/4
779/779 [==============================] - 67s - loss: 0.5862 - acc: 0.8023
Epoch 2/4
779/779 [==============================] - 27s - loss: 0.3940 - acc: 0.8036
Epoch 3/4
779/779 [==============================] - 20s - loss: 0.2192 - acc: 0.9153
Epoch 4/4
779/779 [==============================] - 20s - loss: 0.1818 - acc: 0.9397
correct Accuracy: 97.12%
ERROR Accuracy: 87.50%
total Accuracy: 96.88%
```

Figure 5.2. Keras LSTM experiment results in Block_10 sequences

Seven out of eight failed sequences have been successfully predicted. And 303 out of 312 success events have been successfully predicted. Our total accuracy has achieved 96.88%.

We also conduct the clustering and testing experiment on Block_181 and Block_130, both of which experiments can reach more than 97% accuracy.

5.4. Predicting Anomalies from Streaming System Logs

5.4.1. Extracting Critical Flow from Event Block Sequences by Using TSM

In Section 5.3.1 we describe how to collect system event sequences in a range before one specific system event's succeed or failed position from Event Block lists. We extract some sequences to create an LSTM model learning set. As our study on HPC system has found, most HPC system operations always have multiple steps, and these operation steps are following a principle. That means even the sequences in learning set are not the same; there are still some critical events which are occurring followed by a step-by-step sequence. For example:

	LSTM Learning set sequences :
Sequence 1	M_0, M_1, M_2...... M_100...... M_200...... M_300...... M_400...... M_498
Sequence 2	M_1, ... M_99, M_100...... M_200...... M_300, M_301...... M_400, M_401...... M_498
......	
Sequence n	M_0, M_1, ... M_100...... M_199, M_200...... M_300...... M_400, M_450...... M_498

TABLE 5.1. ONE SPECIFIC EVENT'S LSTM LEARNING SEQUENCES

Table 5.1 is an example of one specific event's LSTM learning set sequences. The sequences are not the same. But we can still use TSM to learn a commonly occurring sequence pattern:

Critical workflow	M_1, M_100, M_200, M_300, M_400, M_498

TABLE 5.2. CRITICAL WORKFLOW LEARNED FROM TABLE 5.1

We call such a step-by-step sequence which have these critical events as this specific system event's **"critical workflow."**

To extract critical workflow by implementing TSM on the LSTM learning sequence set, we should get the first summary which events are occurring commonly in all or most of the sequences. And we can make TSM grow sequence patterns on these common events. With very simple post-cleaning, we can easily find a critical workflow from these TSM-grown sequence patterns.

5.4.2. Realtime Monitoring and Event Occurrence Prediction

One specific system event will have at least one critical workflow; we will load these critical workflows to memory. When SLEBD is analyzing streaming system log lines, we run a system event supervisor. If one specific system event's critical workflow's combination fully occurred, we can predict such a specific system event is going to occur in the future, no matter success or fail.

5.4.3. Predicting Successful Execution of Events

When we can predict one specific system event is going to occur in the future, we can collect the event sequence, which mapping the first event to the last event of this specific event's critical workflow. Then we can use this specific event's LSTM model to test this event sequence and predict if this specific event is going to be successfully executed before it occurs.

5.4.4. Preliminary Experiment Results

As we only learned LSTM for 3 Event Blocks: Block_10, Block_181, and Block_130 from their 300 days of system logs. We were just done simple testing on these three LSTM models to verify if our idea works. The logs from day 301 to 310 are our testing objects. We use the EB patterns which learned from the first 300 days to extract EB lists from these ten days of syslog files, then do our research on the 3 Event Block we mentioned above.

In this period of 10 days, the three Event Blocks' occurring count is like this:

As the log messages are already included in the system log files, we can very easily extract Event Block lists from this period's log files and summarize each Event Block's

Block_name	*Success*	*Fail*
*Block_*10	16	1
*Block_*181	10	0
*Block_*130	8	0

Table 5.3. THREE EVENT BLOCKS' OCCURRING COUNT

occurring count. We use a script to simulate generating these log messages real-time and use these three Event Blocks' critical workflow to predict if these three Event Blocks are going to happen shortly. If our test flow predicts one Event Block is going to happen, our program will use that Event Block's corresponding LSTM model on the event sequence, which mapped by the critical workflow to predict if that Event Block will successfully execute. Our experiment result shows the prediction accuracy on both success and failure events are all 100%.

5.5. Accelerating RNN on FPGA with Efficient Conversion of High-Level Designs to RTL

After completed the research of detecting anomaly events from HPC system log, I thought about using FPGA to accelerate the RNN algorithms calculation performance. As I have found, manually implementing Python-based RNN algorithms into Register Transistor Level (RTL) design is very time costing. Even a hardware engineer has designed an RTL code for one specified RNN algorithm, his manually implemented design is not easy to change, and he may need to rewrite a new RTL design if his original RNN algorithm has very slightly changed.

Meanwhile, FPGA programming is different from CPU and GPU program. Writing Register Transistor Level (RTL) codes requires a knowledge of how the sequential logic circuit works and be familiar with the resources on the FPGA platform. So, RTL programming for implementing RNN algorithms is very time consuming and error-prone. To make things worse, even a slight change in an RNN algorithm may lead to rewriting a large portion of the RTL design. High-level programming languages, on the other hand, have been widely used to write machine learning (including deep learning) applications. For example, deep learning libraries in Python, e.g., Tensorflow [15], PyTorch [11], Keras [9], and Numpy[19], make application development highly efficient.

In this step of research, we present a design flow by which high-level RNN implementations can be converted to RTL designs efficiently and automatically. We leverage a popular RNN development approach in which an RNN gate is declared as a Numpy array type for a two-dimensional floating-point matrix. A major benefit of this approach is that it can overload some common operators and enable developers to conduct efficient matrix and vector operations instead of handling single elements. We follow a similar development paradigm and propose a new design flow in which developers can design and verify their RNN implementations in Python and convert their implementations to RTL designs automatically. Changes can be made to high-level programs, and RTL designs are regenerated quickly. The produced RTL design describes an RNN algorithm in-circuit logic without the need of assistance from SOC run on ARM cores or Python program run on CPU. Most importantly, the RTL design can realize the dataflow pipeline and parallel computing at low power.

The experiment shows the automatically generated RTL deployed on FPGA can achieve seven times faster than Python code running on CPU, and their calculation result is perfectly matched. Which proved using a script to automatically generate RTL based RNN design can help to speed up RNN calculation performance.

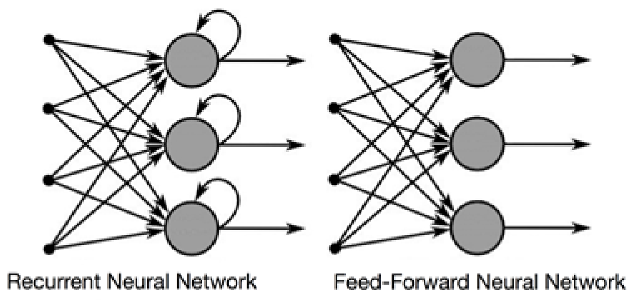


FIGURE 5.3. RNN and Feed-Forward Neural Network

5.6. RNN and LSTM Background

5.6.1. RNN

Recurrent Neural Networks (RNNs) were the first Neural Networks which designed to deal with sequential data. Unlike the Feed-Forward Neural Networks [78], RNN passes the previous outputs of hidden layers back to the current input. Figure 5.3 illustrates the structure of RNN and Feed-Forward Neural Networks.

For an input sequence X = (x_1, ..., x_t), the hidden layer H= (h_1, ..., h_t) outputs a vector sequence Y = (y_1, ..., y_t) which can be calculated as:

$$h_t = \delta(w_{xh}x_t + w_{hh}h_{t-1} + b_h)$$
$$y_t = w_{hy}h_t + b_y \tag{5.1}$$

Where w_{xh} is the weight matrix connecting the input layer and the hidden layer, w_{hy} is the weight matrix connecting the hidden layer and the output layer. w_{hh} is the recurrent matrix connecting the hidden states in two consecutive time steps' hidden states, e.g. $h_{t-1}$and h_t. b_h and b_y are the biases vector of the hidden and output layers. δ is the hidden layer activation function, e.g., Sigmoid or Tanh.

5.6.2. RNN Training

RNN training procedure has three main sections:

1. Forward propagation;
2. Backward propagation through time (BPTT);
3. Weight matrix update.

To train an RNN model, we need to provide a golden result sequence that contains the result that the RNN model is expected to predict.

Forward propagation

The forward propagation step generates an output sequence Y = (y_1, ... , y_t) using the input sequence X = (x_1, ... , x_t). The main idea of forwarding propagation is introduced in Section 5.6.1.

Backward propagation through time

In Forward propagation, the prediction result may deviate from the expected result. After the sequence is processed, it is helpful to go back through the neural network to find the partial derivative of error between the predicted result and the expected one. The partial derivative is then removed from its corresponding weight matrix to reduce deviation for processing future sequences.

At time step t, the derivate d_t of the predicted result y_t from the expect result o_t can be calculated as:

$$d_t = o_t - y_t \tag{5.2}$$

The error derivative δ at time step t is:

$$\delta_t = d_t * y_t * (1 - y_t) \tag{5.3}$$

where $y_t * (1 - y_t)$ is the partial derivative $\frac{f'(x)}{f(x)}$ of Sigmoid.

Starting from the last time step and backward to the first time step, at each time step we calculate the error derivative δ and update the weight matrix and bias as:

$$\begin{gathered} w_{hh} += \delta_t * h_{t-1} \\ w_{xh} += \delta_t * x_t \\ b += \delta_t \end{gathered} \tag{5.4}$$

Weight matrix update

After the backward propagation step is completed, the weight matrix is updated following a learning rate as:

$$w_{hh} += w_{hh} * learning\_rate \tag{5.5}$$

5.6.3. Long Short-Term Memory

As a recurrent neural network architecture, we study Long Short-Term Memory (LSTM) and accelerate an LSTM model on FPGA. In this section, we briefly describe how LSTM performs forward propagation.

RNN can learn from inputs before the current time. Traditional RNN has one input gate and one output gate. Thus, it cannot learn from inputs that are far back in time. Long short-term memory (LSTM) has two more RNN gates called cell gate and forget gate. The four RNN gates together form an LSTM cell. Figure 5.4 shows the structure of an LSTM cell.

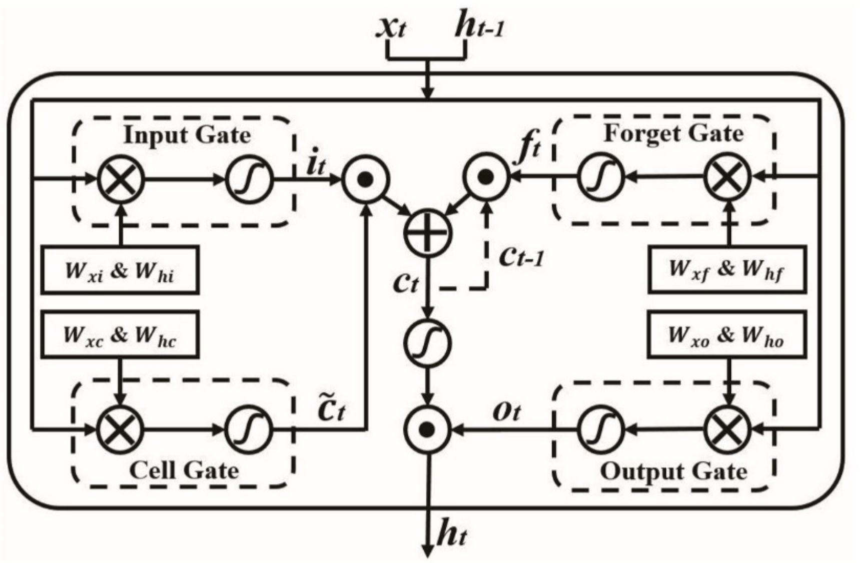


FIGURE 5.4. Standard LSTM cell architecture

At time t, these gates produce outputs as follows :

$$
\begin{aligned}
i_t &= \delta(w_{xi}x_t + w_{hi}h_{t-1} + b_i) \\
f_t &= \delta(w_{xf}x_t + w_{hf}h_{t-1} + b_f) \\
o_t &= \delta(w_{xo}x_t + w_{ho}h_{t-1} + b_o) \\
a_t &= tanh(w_{xa}x_t + w_{ha}h_{t-1} + b_a) \\
c_t &= f_t * c_{t-1} + i_t * a_t \\
h_t &= o_t * c_t \\
y_t &= softmax(w_{hy}h_t + h_y)
\end{aligned} \tag{5.6}
$$

Where f_t, i_t, o_t, a_t, c_t, h_t and y_t represent the output vectors of forget gate, input gate, output gate, current input state, cell state, hidden state, and output value at time step t, respectively. Operations "+" and "*" represent element-wise addition and multiplication or Hadamard product of two vectors.

Note that the preceding logic gates perform floating-point matrix or vector operations. Both RNN and LSTM involve the following floating-point computations.

Addition, subtraction, and multiplication functions of floating-point scalars.

Floating-point matrix multiplication.

Activation functions, including Sigmoid; Tanh; partial derivative of sigmoid and more.

The deep learning libraries in Python allow application developers to use and or design RNN and LSTM models in Python. A single line of code in Python, for example, multiplying two matrices could lead to hundreds of floating-point computations. How to implement and accelerate deep learning models and applications efficiently on FPGA, which is equipped with reconfigurable logics such as lookup tables instead of general-purpose processor cores is challenging

5.7. Accelerating Floating-Point Computations on FPGA

As floating-point operations are fundamental to deep learning applications, we first investigate how to accelerate floating-point computations on FPGAs.

5.7.1. Floating-Point Calculation Feature IP

Modern FPGAs are usually equipped with Intellectual Property (IP) cores [18] for integer and floating-point calculation. Those IP cores can be configured to perform various operations, including addition, subtraction, multiplication, division, comparison, and exponential function. We can declare the calculation device using the IP cores in the register-transfer level (RTL) design, and we can configure their latency.

Both calculation devices declared from the same IP will have the same type of data I/O handshaking protocol. For example, when inputting one data into the calculation device, the part of the input is the master part. The part of receiving, or our calculation device, is the slave part. When the master part sees the slave part raise its INPUT_READY signal means it is ready to receive new incoming data. The master part will write the input data to the slave part's data input port register and raise the slave part's INPUT_VALID signal. If both the INPUT_READY and INPUT_VALID signals are all high, both the master and slave parts can think the design has successfully transferred one input data and the slave part can begin its calculation.

In a sequential logic circuit, all write data or signal values are using a blocking assignment to assign a value to the registers. The Blocking assignment will store a data value in flip-flop first and release to designated registers in the next edge of the rising or downing of the clock signal. And one data or signal assigning operation must be placed in only one process and triggered by one same group of control signals listed in this process. That means no matter we need to calculate how much data, a calculation device can only receive one input data in one clock cycle.

5.7.2. FPGA Help to Accelerate Floating-Point Calculation Performance

The calculation commands are executing sequentially in the high-level designs. That means before the program receives the current command's output data, it will not execute the next command or move forward to the next loop body and input the next data, even though the current command has no data dependency with the next command. However, we can eliminate this situation can by using FPGA. As we have found, when we input one data

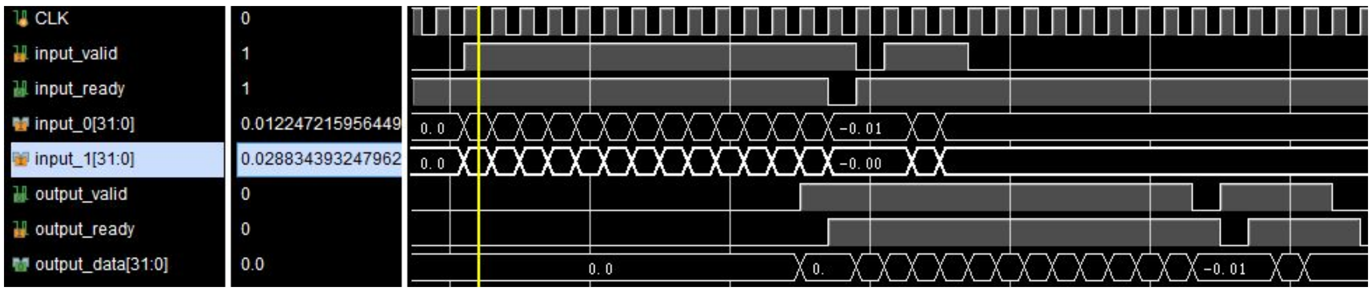


FIGURE 5.5. The waveform of floating-point data adder device's data flow pipeline example

into the calculation device, the device will not lower its INPUT_READY signal immediately in the next clock cycle. That means the calculation device is capable of receiving multiple input data, even previous input data's calculation results are not output. Such characteristic inspired us to make FPGA accelerate floating-point calculation in two ways:

Data flow pipelining

As we mentioned, in an RTL design input master part can keep inputting data to the slave device in every clock cycle if the slave device does not lower its INPUT_READY signal. And at the device's result outputting side, the device will become the master part, and our RTL design will become the slave part. As we have found, the calculation device's capacity is limited. A device can keep receiving data in a continuous limited number of cycles, but if the slave does not receive the output data, the device will keep holding the current output data, keep its OUTPUT_VALID signal high, and reject to receive new input data (lower its INPUT_READY signal). In other words, if we make our design always ready to receive output data from the device, we can keep input data in every cycle and receive output data in every cycle shortly, looks like data is flowing as pipelining

We can estimate the performance of the data flow pipeline like this: imagine a floating-point adder device's calculation latency is 12 cycles. That means if the device receives data at the N-th clock cycle, we can expect to receive the calculation result at the (N+12)-th cycle. If we have 16 pairs of floating-point numbers need to add. And we make our design fully pipelined, inputting and receiving results from the device, theoretically, we can receive 16 adding results in just 28 clock cycles. Figure 5.5 is showing the waveform of such a data flow pipeline example.

Parallelism Computing

In the software level, calculate operations are executed sequentially, even though two operations have no data dependency. However, in the sequential logic circuit, every device has its driver process, and all such processes are triggered every clock cycle. That means we can make operations that have no data dependency execute at the same time on FPGA. One RTL design can accommodate numbers of the same type of calculation device. For example, we can create an RTL design that has four input ports and two output ports and its feature is doing multiple for each pair of input data. At the device input side, the input data master can input two pairs of data at the same time. And at the device output side, the output data slave can receive two output data at the same time. Figure 5.6 is showing such RTL design.

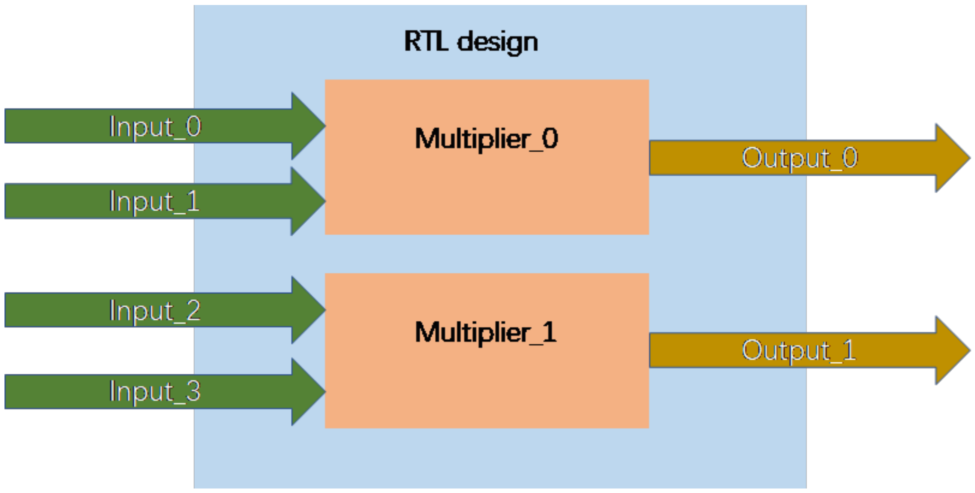


FIGURE 5.6. Parallelism Computing RTL design

5.7.3. Calculate Operations Packaging

As we introduced at Section 5.6, RNN usually doing floating-point calculations on vectors, not only one element. For example, we need to use the activation function Sigmoid to act on a 2-D matrix. The Sigmoid function can be described as:

$$\frac{1}{1+e^{-x}} \tag{5.7}$$

Imagine the size of this 2-D matrix is N*M, means we need to input N*M of floating-point numbers x_{ij} into the exponential device to calculate N*M times of $e^{x_{ij}}$. Then go to the next step. We package all those operations related to Sigmoid into one calculation device to make such a procedure running in the dataflow pipeline .

To calculate the Sigmoid result from one input data x, we need three steps of calculation:

1. Calculate x's exponential result e^{-x};
2. Add e^{-x} result with 1.0;
3. Divide $(1.0 + e^{-x})$ with 1.0.

That means we need three floating-point calculation devices: exponential device, add device, and divide device.

According to IEEE 754 standard [20], a single-precision 32-bit floating-point number has one sign bit, 8 bits to represent the exponent number and 23 bits to represent the significand precision. That means for an input number x; we can get its negative value -x by flip its 32-nd bit and don't need to multiply it with -1.0.

We use combinatorial logic to implement the Sigmoid device. Input data x will be first input to the exponential device, exponential device's data output port will be connected to add device's input port directly, exponential device's OUTPUT_VALID signal will be connected to add device's INPUT_VALID signal and add device's INPUT_READY signal will be connected to exponential device's OUTPUT_READY signal. And the connection between add device to divide device is in the same way. This feature will guarantee the previous device's output result can be received by the next step device immediately.

The adding device and the dividing device have two input ports. We also need to input 1.0 to the other input port. We declared a 32-bit register to store parameter 1.0's binary format 0x3f800000 and connect this register to add and divide devices' other input port. The output side of the dividing device will become the whole Sigmoid device's output side. Figure 5.7 is showing the structure of the packaged Sigmoid device.

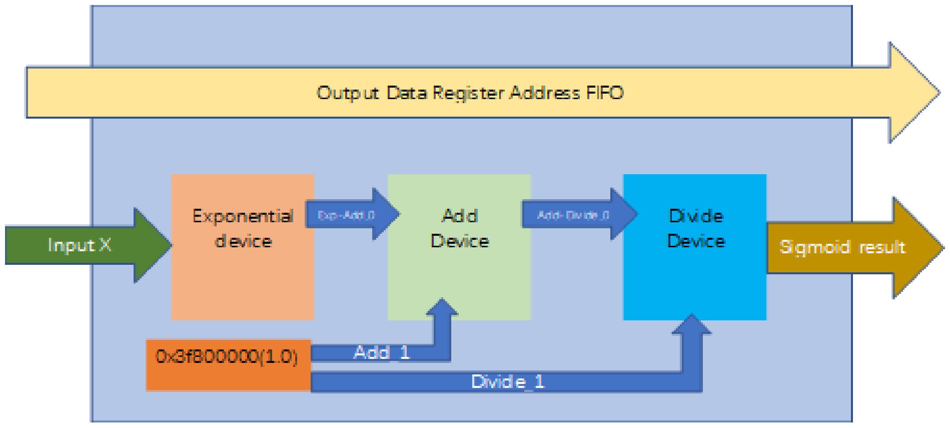


FIGURE 5.7. Packaged Sigmoid device with result address FIFO

We can use the similar way to design combined operations such as Sigmoid function's partial derivative:

$$x * (1 - x) \tag{5.8}$$

or tanh:

$$\frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5.9}$$

or a multiply-add combined operation to handle weight matrix update by learning rate operation which showing in function (5.5).

Sometimes we need an internal FIFO data register to store input data. For example, in the Sigmoid function's partial derivative function(5.8), the whole device has only one input port to receive input data x. We must input x into a subtract device first to calculate (1 - x), and we need to wait for the subtract device's calculation latency to its output. To guarantee data flow pipeline and input x with its corresponding (1 - x) result into multiplication device, we need to store x into FIFO at the same time inputting x into subtracting device.

5.7.4. Performance Evaluation

As we mentioned, Xilinx IP based floating-point calculation devices' latency can be configurated. For example, the floating-point add device requires 2 of DSP48E1 slices[59], and the default latency is 12 cycles. The exponential device requires 1 of DSP48E1 slice in Medium Usage mode, and the default latency is 21 cycles. The dividing device must use LUT logic to implement, and default latency is 29 cycles. So we can accurately evaluate each device will cost how many resources on the FPGA chip and its total latency. For example, our Sigmoid device requires 3 DSP48E1 slices, and its default total latency is 62.

We also test the performance of the RTL implementation, which is generated from Vivado_HLS[17]. Vivado_HLS tool do not support calling the exponential function from the C math library or allow the user to designate using Vivado exponential IP. So, the above works are all using a piecewise linear approximation of nonlinear function (PLAN) approach to do the exponential calculation. For example, Taylor's Formula:

$$e^x \approx 1 + \sum_{i=1}^{n} \frac{x^i}{i!} \tag{5.10}$$

As we find, to achieve exponential calculation's accuracy, the upper limit n in Taylor's formula must be at least 30. We tried to implement exponential calculation function in the C program and generate RTL design by using Vivado_HLS, the resource usage of the RTL design is like this:

From Table 5.4, we can find the exponential device's RTL, which is generated by Vivado_HLS requires more resources and has a longer latency than the exponential device which is implemented using the floating-point IP core in our sigmoid design (DSP: 1 vs. 5; Latency: 21 vs. 755).

5.7.5. Address FIFO in Calculation Package

If we make our RTL design drive the calculation devices running in full data flow pipeline mode, our design will receive output data in every cycle. The RTL design must make sure to write the output data into the correct register address. Our solution is to

Resource	*Number*
$DSP48E1$ *slice*	5
FF *units*	1402
LUT logic	2473
Latency	755

Table 5.4. RESOURCE USAGE OF THE EXPONENTIAL CALCULATION DEVICE RTL GENERATED FROM VIVADO_HLS

add an address FIFO into each calculation device. The FIFO's WRITE signal is combined with the device's INPUT_VALID signal, and the FIFO's READ signal is combined with the device's OUTPUT_READY signal. We make our RTL design write each input data's result destination register address to the address FIFO in the same time when inputting data and when a device has output result, the result's corresponding register address will be pop out from the FIFO.

FIFO's internal memory is limited. If the RTL design keeps writing the address into it could make the FIFO written full, which could make the device unable to receive new input data. Because we can evaluate the devices' performance, we can easily solve this problem by extending FIFO's internal memory length a little larger than the device's anticipated latency. For example, if one device's latency is ten cycles, means a result will be output ten cycles after its corresponding input data been accepted by the device, and its corresponding address will be pop from FIFO too. If we make the address FIFO's internal memory larger than 10, e.g., 16, will make the FIFO impossible to be written fully. This solution can achieve a balance between resource and performance because too large FIFO memory will waste FPGA's on-chip BRAM.

5.7.6. Vector Operating Device

In our research, we package operations in packages which include address FIFO. Such packages have different features, but their input and output part data ports and control signals are similar. The only difference is they could have one data input port (e.g., Sigmoid, Tanh), two data inputs (multiply, add, subtract) and three data inputs (multiply-add).

One special kind of device is the vector operating device, which is used for matrix-vector product or finding the maximum value in one vector. These vector operations are implemented by using the tournament structure[92]. We use the matrix-vector product as an example. We have two input matrix A[n][k] and B[k][m] and want to multiply them into another matrix C[n][m]. According to Linear Algebra, each element C[i][j] in matrix C can be calculated by this:

$$C[i][j] = \sum_{u=1}^{k} A[i][u] * B[u][j] \tag{5.11}$$

We can create a vector product device that has the tournament structure. For example, in the first layer, we have 16 of the multiplication device which can input 16 elements from matrix A[i][:] and input 16 elements from matrix B[:][j]. Their corresponding output address is line = i and column = j. In the next layers are all add devices, and we have four layers of such add devices (2^3 to 2^0). The 5th layer of this device only has one add device, and its output is the whole device's output. Figure 5.8 is showing the vector product device's tournament structure.

The example we provided requires 62 (2 * 31) of DSP48E1 slides and its default latency is 57 (9 + 12 * 4) as the multiplication device's default latency is nine and add device's default latency is 12. If resource allows, such device can be extended bigger, e.g., one layer of multiplication device and five layers of add device which can input 32 pairs of data at the same time.

If the vector's size is smaller than the vector operating device, we can input 0 to the rest input ports. If the vector's size is bigger than the vector operating device, for example,

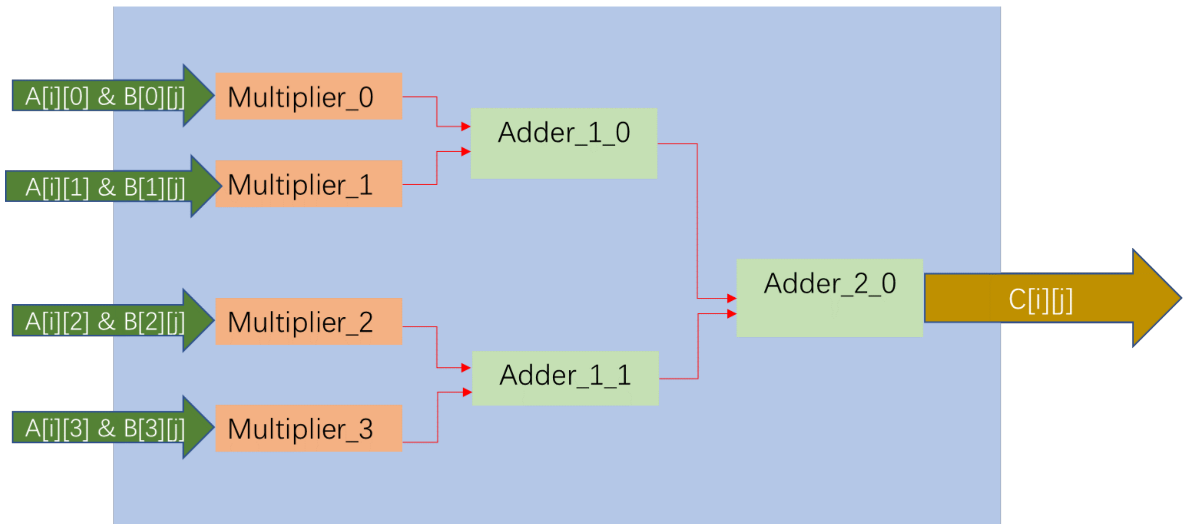


FIGURE 5.8. Four by four-vector product device using tournament structure

the vector operation device can handle 32 pairs of input, but the vector size is 100, we can also get the final matrix multiply results in 4 rounds (100/32 + 1).

5.7.7. Calculation Device Input/Output Handshaking Method

As we introduced in Section 5.7.2 data flow pipelining example, the calculation devices declared from Xilinx IP cores have VALID-READY handshaking protocol on both the input and output part. Such handshaking protocol can guarantee correct data transferring between the master part and the slave part.

Calculation device declared from Xilinx IP cores can receive input data continuously, but their capacity is limited. That means the calculation device may not be able to receive new incoming data by lower its INPUT_READY signal after constantly receiving several clock cycles of input data. When the input master part found the device lower its INPUT_READY signal, it should pause inputting operation.

As we mentioned in Section 5.7.1, we implement our RTL design in a sequential logic circuit. Imagine this situation: the master part is going to use a blocking assignment to write the (i)-th data from total T of data. At (n)-th cycle, the master part sees the slave part(calculation device)'s INPUT_READY signal is high, means the slave is ready to receive data. Master will use blocking assignment to write the (i)-th data to slave's data input register and raise the slave's INPUT_VALID signal. At (n+1)-th cycle, the (i)-th data will be assigned to slave's INPUT_DATA port, and slave's INPUT_VALID signal will be high.

At (n+1)-th cycle, we may meet two kinds of situation:

1. Slave keep its INPUT_READY high, this means the slave part successfully received the (i)-th data. The master part can input the (i+1)-th data to the slave, and the slave's data input register's value will change at (n+2)-th cycle.

2. Slave lower its INPUT_READY. Means at (n+1)-th cycle the slave refuse to receive the (i)-th data. In this situation, the master must keep the (i)-th data reserved and try to input it to slave until seeing the slave's INPUT_READY signal raise again.

We write a master part input driver to handle the above situations. When deploying a calculation device to our RTL design, we also declare local address registers to remember the number of input data. Pseudocode 1 is describing our master part input driver which operating on one-dimension register. Two-dimension register operations are similar and only need to add a line-column number judging logic.

At the device's data output side, the calculation device becomes master, and our RTL design becomes the slave part. Our device output driver is simple: when the device raises its OUTPUT_VALID signal, RTL design raise its OUTPUT_READY signal; otherwise, keep it low. We drive the device output side like this because we combine the address FIFO's READ signal with the OUTPUT_READY signal, and if we raise OUTPUT_READY early, the address will be pop out early than the output data

5.8. Designing RNN Using Scientific Computing Library Numpy

Python has a powerful scientific computing library, Numpy, which is very useful in creating large scale designs that operate on matrix or vectors by converting its mathematics logic into Python code. For example, the first function of (5.2):

$$h_t = \delta(w_{xh}x_t + w_{hh}h_{t-1} + b_h) \tag{5.12}$$

can be described by using Numpy operations like:

Pseudocode 1. Device master part input driver

```
always @(posedge clk)
    if (slave_input_ready == 1 && local_addr_reg != end_num) begin
        if (slave_input_valid == 1) begin
            if (local_addr_reg != end_num - 1) begin
                slave_input_data <= data[local_addr_reg + 1];
                slave_input_valid <= 1;
            end
            else slave_input_valid <= 0;
            local_addr_reg <= local_addr_reg + 1;
        end
        else begin
            slave_input_data <= data[local_addr_reg];
            slave_input_valid <= 1;
        end
    end
    else slave_input_valid <= 0;
end
```

$$h[:,t] = sigmoid(wx[:,x].T + wh.dot(h[:,t-1]) + bh) \tag{5.13}$$

Where sigmoid() function is a predefined Python function, wx[hidden][data], h[hidden][sequence], wh[hidden][hidden], b[hidden][1] are pre-declared numpy.array type two-dimension matrix. In program level w_{xh} x_t is selecting the (x)-th vector from w_{xh}.

Before we use Python and Numpy to create our RNN design, we need to decide some important values:

1. **hidden_size** represents how long a vector in the hidden layer will be;
2. **data_size** represents how many kinds of data an RNN will meet;
3. **sequence_size** represents how long a data sequence is.

These values are easy to understand. In natural language processing research, a common way of preprocessing and digitalize a context file which has thousands of lines of sentences is using Natural Language Tool Kit (NLTK)[13] to create the top frequent happening words and create a dictionary. In this dictionary, each word has a unique number representing itself. Such dictionary size is the data_size. Sequence_size is the length of the sequence which RNN need to receive. We need to decide the sizes before we train the RNN model. However, digitalized sequences' length could change. We can use the pad_sequence[10] function provided by Deep Learning Library Keras to make our RNN training sequences have the same length.

Numpy has overloaded the add operator to support adding vectors. However, all vectors must have the same size. In our example, the variables in sigmoid function which adding together are the same size (hidden * 1) vectors. If their sizes have a difference, Numpy will report an error.

5.9. The Idea of Converting Python Commands into RTL Design

After understanding of how FPGA accelerate floating-point calculating and how to use Python and Numpy create RNN design, we can try to convert the Python-based commands into pure sequential logic RTL based design which can deploy on FPGA.

In Python code (5.13) there are three types of floating-point calculations:

1. Vector add;
2. Matrix multiply;
3. Sigmoid calculation.

We can use the calculation devices we introduced in Section 5.7.3, which function is addition, vector product and Sigmoid to handle code (5.13).

In Section 5.7.2, we introduced the concept of FPGA based Parallelism Computing can be implemented between operations which have no data dependency to speed up the calculation. As we can find from code (5.13), some operations do not have data dependency so that we can schedule code (5.13) in three steps:

1. Use vector product device to calculate wh.dot(h[:, t-1]) as result_1.

   Use add device to calculate wx[:, x].T + bh as result_2.

2. Use add device to calculate result_1 + result_2 as result_3.

3. Use the sigmoid device to calculate sigmoid(result_3) and store all result data from the sigmoid device into the corresponding address at register h[:, t].

As we can find, the two operations in step 1 have no data dependency, and their calculation is using different devices. So, these operations can be placed in the same step and parallelly computed. Step 1's total latency is decided by the operation which latency is longer. Step 2 requires calculation results from step 1, so step 2 need to wait for step 1 finished. In each step, all device can achieve data flow fully pipelining by using a drive which is similar to Pseudocode 1.

For temporary results like result_1, we need to store them into temporary registers. So, besides weight matrix registers, we also need to declare some fixed length temporary registers using on-chip BRAM. Such registers are reusable and do not need to be very large but enough to accept the largest output data size from the calculation device, e.g., hidden_size.

As we introduced, Numpy library can operate on vectors, but all vectors' size must be the same. This feature can ensure all RTL level calculation device can have definite input and output size. So, our RTL design can know when it has to input enough input data or receive enough output results. In our RTL design, each register has a DATA_RECEIVE_DONE signal. In each step, all registers have a condition of rising such DONE signal with received a designated number of output data from one device. In one step, if all registers which need to operate in this step have risen their DONE signals, it can move forward to the next step. For example, in calculation step 1, the vector product device and add device have different latency. But if temporary registers which storing result_1 and result_2 have risen their DONE signals, means all result_1 and result_2 have received. The RTL design can move forward to calculation step 2.

5.10. Large Scale Python-RTL Code Conversion

In the above Sections 5.7, 5.8 and 5.9, we introduced the main ideas of designing RTL level packaged floating-point calculation devices, designing RNN by using Python and Numpy and converting Python commands into RTL designs. In this section, we will introduce our approach of converting the whole Python-based RNN design into the RTL code.

In our research, we have created an RTL code automatic generate system. Different from general propose HLS tools, our system is designed especially for generating RTL level RNN designs from Python code. In our system, we created an RTL level calculation device basic operation driver library. Such driver example is introduced in Section 5.7.7 Pseudocode 1, which can make the calculation devices achieve data flow pipeline fully. The user does not need to worry about how the devices operate in RTL level operates but only focus on their Python-based high-level design. To generate their RTL level RNN design, the user needs to provide one configuration file and one design flow file.

The configuration file list all the devices and registers, including each gates' weight matrix and temporary registers which are needed in the RNN design. All the registers should be two-dimension registers with definite line and column sizes. Some important parameters, such as sequence_size, hidden_size, and data_size also listed here. We can evaluate the number and type of devices and registers in Python designing section. The overall strategy of declaring devices and registers is the more calculate operations the user wants to execute parallel, the more same type devices and temporary registers they need to declare.

With the configuration file, our system will generate the calculation devices and registers declaration codes in the RTL design file. Device declaration code also includes data I/O ports and control signal registers and wires.

The design flow file is the RNN operation flow converted from the original Python design. In Section 5.6.2, we introduced the whole RNN design have three main sections. The user can split each section into multiple stages. The overall strategy and restraint of splitting stages are like this. In each stage:

1. All calculate operations will execute in parallel;
2. One calculation device can only receive input data from one register;
3. One register can only receive output data from one device;
4. Data dependency can be supported by using temporary registers;
5. Users can resue all calculation devices and temporary registers in different stages;
6. When all registers in this stage receive designated number of output data from the devices assigned to them, move forward to the next stage.

Our suggestion is to make one line of Python code as one stage. For example, we can describe the function (5.2) by Python code (5.14):

$$\begin{aligned} h[:,t] &= sigmoid(wx[:,x].T + wh.dot(h[:,t-1]) + bh) \\ y[:,t] &= why.dot(h[:,t]) + by \end{aligned} \tag{5.14}$$

If we make the first line of code (5.14) as stage_0, the second line of code (14) as stage_1, to meet our restriction of splitting stages, we need two add devices **Add_0** and **Add_1** because there are two add operations in the first line of code (5.14). One vector product device **Vmp_0**, one sigmoid device **Sigmoid_0**. We also need to declare three temporary registers reg_0 to reg_2.

Users can write a Python-based design flow code like Pseudocode 2 to describe all calculate operations in each stage. Our RTL generate system will analyze this design flow file and create a list for all devices and registers of their operations in each stage. With simple pattern recognition functions, our system can also find each device's input data size and each register's data receive size. The device and register operation list are like Table 5.5 and Table 5.6

The sequential logic circuit must assign each register by only one process. Our system will generate one data input and one data output process for each device based on the device's operation list. And generate one data assigning a process for each register. Our design is like inserting an arbiter state machine in front of all calculate devices and data registers. In our RTL design, there will be a stage number counter register. The arbiter will decide whether

Pseudocode 2. Device calculation flow in two stages

```
stage = 0
  Add_0.opt = (i_0 = "wx[:,x].T",  i_1 = "bh", o = "reg_0")
  Vmp_0.opt = (i_0 = "wh",  i_1 = "h[:,t-1]", o = "reg_1")
  Add_1.opt = (i_0 = "reg_0",  i_1 = "reg_1", o = "reg_2")
  Sigmoid_0.opt = (i_0 = "reg_2", o = "h[:,t]")
stage = 1
  Vmp_0.opt = (i_0 = "why",  i_1 = "h[:,t]", o = "reg_0")
  Add_0.opt = (i_0 = "reg_0",  i_1 = "by", o = "y[:,t]")
```

Device	*Stage_0*	*Stage_1*
Add_0	$i_0 = $ "$wx[:,x].T$"; $i_1 = $ "bh"	$i_0 = $ "reg_0"; $i_1 = $ "by"
Add_1	$i_0 = $ "reg_0"; $i_1 = $ "reg_1"	
Vmp_0	$i_0 = $ "wh"; $i_1 = $ "$h[:,t-1]$"	$i_0 = $ "why"; $i\_1 = h[:,t]$"
Sigmoid_0	$i_0 = $ "reg_2"	

Table 5.5. DEVICE DATA INPUT SOURCE LIST

Register	*Stage_0*	*Stage_1*
reg_0	*Add_0*	*Vmp_0*
reg_1	*Vmp_0*	
reg_2	*Add_1*	
$h[:,t]$	*Sigmoid_0*	
$y[:,t]$		*Add_0*

Table 5.6. REGISTER RESULT DATA RECEIVE OPERATION LIST

to trigger the device to standby by the value in the stage number register and make the device receive input data from the designated data register. Data registers' arbiters make registers receive the result from a device in the same way.

From Pseudocode 2 we can find line 2 to 5 belongs to the same stage, that means the four devices are all standby when stage number equal 0. However, operation in code line 4's input data are all from temporary registers, so line 4's calculation will not execute if reg_0_REC_DONE and reg_1_REC_DONE signals are not all high yet.

5.11. FPGA-Based RNN Experiment

5.11.1. Experiment Case

After finishing the RTL design of all calculation devices and create the RTL level RNN design generate a system, we tried to use a medium-size real-world RNN design to test our idea. We download our original Python RNN design is from [8], a basic tutorial to help the reader to have the first step understanding about how RNN works. This RNN design contains both forward propagation, backward propagation, and matrix update sections. Comparing with our previous large scale natural language commercial complaint data clustering LSTM design [83] we find this tutorial code contains all basic calculation methods which RNN uses. The only difference is this tutorial code is smaller.

This RNN design program requires two 8-bit integer numbers a and b which range is from 0 to 127 as RNN inputs. The RNN program's output c is the predicted a and b's adding result, which c's range is from 0 to 255, and one predicts overall error which compared with a and b's real adding result. This RNN model treats both a, b, and c as 8-bits binary sequences, so c's each bit is predicted by a and b's current bit's and previous bits' input values.

In the RNN program, the designer randomly generates 10,000 pairs of input data. The RNN program will run 10,000 rounds, and each round will have forward propagation section to predict the output result c, then run backward propagation to get the partial derivative of the error between predicted result with expected result and finally use backward propagation section's update matrix to multiple learning rate to update all gates' weight matrix. After running the Python program we can find in the beginning 5,000 rounds there may have mispredicted results and overall error values are all greater than 1.5, but in the last 4,000 round the prediction results are almost all correct, and the overall error values are all less than 0.5. To test our RTL design's correctness, we recorded all the input data predicted output data and the overall error in all rounds.

5.11.2. Our Configuration Files for RTL Generate System

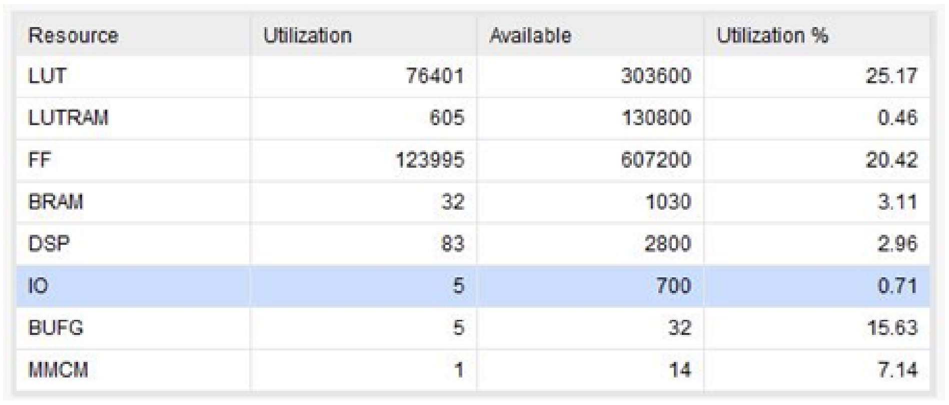
In Section 5.8, we introduced three important variables in RNN design. In this RNN design, the **data_size** is two because each bit of binary number can have only two possible values, 0 and 1. The **sequence_size** is eight because the python design treats the input data a and b and output data c as 8-bit binary sequences. The **hidden_size** in the original design is 16. In the original RNN design there are 3 gates, input_gate [2][16], hidden_gate[16][16] and output_gate[16][1], in original design there also have same size of update matrix for each gate. The learning rate in this RNN design is 0.1.

We also declared three of 16*16 32-bits temporary registers in our configuration file. In our design, we have declared two add devices, two multiplication devices, one sigmoid device, one 16 by 16 vector product device, one subtract device, one derivative device, one multiply-add device, and one floating-point number rounding device

As the forward propagation section in the original RNN design has five main commands, we divide the forward propagation section into five stages. The backward propagation section has three main commands, but as we have two additional devices, we can make two commands which do not have data dependency execute in the same time by placing them in the same stage. So, we divide the backward propagation into two stages. As there are three gates and we only declared one multiply-add device, the matrix update section would have three stages.

5.11.3. Experiment Setup

We generate our RTL level RNN design file in Verilog. As the RTL design file been generated, we used Microblaze, a reduced instruction set computer (RISC) system to wrap our RNN design. Our input data is stored on the desktop; as FPGA and desktop are in different clock areas, we also declared two data FIFO to buffer I/O data. We synthesize the whole design into a bitstream by using Vivado 2018.1 version and deploy the bitstream file on Xilinx VC707 board which is using Xilinx Virtex7-485t FPGA chip to test our design's accuracy and performance. The clock frequency is set to 150 MHz. We run a C program on the desktop to transfer our I/O data through UART port. Figure 5.9. is showing the resource usage report of our whole onboard system generated by Vivado.



Resource	Utilization	Available	Utilization %
LUT	76401	303600	25.17
LUTRAM	605	130800	0.46
FF	123995	607200	20.42
BRAM	32	1030	3.11
DSP	83	2800	2.96
IO	5	700	0.71
BUFG	5	32	15.63
MMCM	1	14	7.14

FIGURE 5.9. Our RNN design's resource usage on VC707

5.11.4. Experiment Result in Accuracy and Performance

We want to test our RTL design's feature by not only the trained RNN model can make an accurate prediction, but also every single variable in both RTL and Python can match. Only in this way we can say our RTL level RNN designing method can reflect all the features in Python design. When we evaluate the original Python design, we record the initial values in each gate's matrix which is randomly generated by the Python program and record each round's input data, output result and overall error. When testing our RTL design, we write the initial values in an RTL level configuration file, and when the FPGA board raises its RESET signal, these initial values will be written to the gates' register matrix. Our PC desktop program keeps writing the same input data as Python program used to FPGA board and receives output result and overall error from FPGA board, then compare the output results and overall error values with their corresponding round's Python program result. Their value can match. That means the calculation feature of our RTL design is correct.

The calculation performance is also very important to evaluate how FPGA can accelerate RNN floating-point calculation. As we mentioned in Section 5.7.4, we can reconfigure the latency of Xilinx Floating-point calculation IP cores; we have done two experiments using the same RTL design file with different device latency. In the first experiment, all the floating-point calculation devices use the default latency as Vivado generated; in the second experiment, we manually configured the devices' latency as 1. All the floating-point IP generated devices' DSP slice usage, and latency is listed in Table 5.7.

	DSP	$Default\ latency(exp.1)$	$Set\ latency(exp.2)$
Add	2	12	1
Multiply	2	9	1
Exp	1	21	1
Divide	*N/A*	29	1
Substract	2	12	1

Table 5.7. EACH DEVICE'S DSP USAGE AND LATENCY IN TWO EXPERIMENTS

Our RTL generate system will estimate the overall latency in each round. We can measure the overall latency from when our design receives one pair of input data until finished update all the gates' matrix. Even though operations in the same stage can achieve parallelism computing partially, the total latency of one stage is decided by the operation, which has the largest amount of data need to calculate. And the overall latency of one round is adding all stages' total latency together. To evaluate the real latency lasted on FPGA, we add a latency summarize process in RTL design. The total latency lasted in one round will be output with this round's output result. The estimated latency and real latency in our two experiments are listed in Table 5.8.

	$Estimatedlatency$	$Reallatency$
$Experiment1$	$6,742$	$7,072$
$Experiment2$	$5,247$	$5,439$

Table 5.8. ESTIMATED AND REAL LATENCIES IN TWO EXPERIMENTS

These two experiments show even use extreme low latency which experiment 2 has; the overall latency does not have very obvious improvement. That is because we achieve data flow pipeline in each stage fully. As we have shown in Section 5.7.2, when inputting data into the device, the estimated overall latency is (total_data_num + device latency). So if the total data number is very high, the shortened device latency cannot speed up too much. And as we have observed by running simulation, the devices which have extremely low latency will pause more times than making this device use default latency. This feature proved when making a device run in data flow pipeline mode the extremely low latency is not a good choice.

As we set our design's clock frequency as 150 MHz, means one clock cycle lasts 6.7 nanoseconds. So, ignore the I/O time cost, by using the default device latency the overall FPGA calculation time cost on all 10,000 rounds is only 7072 * 10000 * 6.7 = 473,824,000 ns = 0.47s.

We also run the original Python-based RNN design on our desktop which uses Intel Core i7-8700 3.2GHz CPU, 16 Gigabytes of memory, and Ubuntu 18.04 OS. Its execution time is 3.7 seconds. That means even using default latency; our RTL design can still achieve 7.87 times of speed-up. And according to Vivado's report, our design's on-chip power is only 1.343W. Our desktop's power supply output is 450W. This experiment shows comparing to CPU, FPGA is an ideal platform to accelerate RNN calculate performance by consuming very low energy.

5.11.5. Advanced Parallelism Computing Design

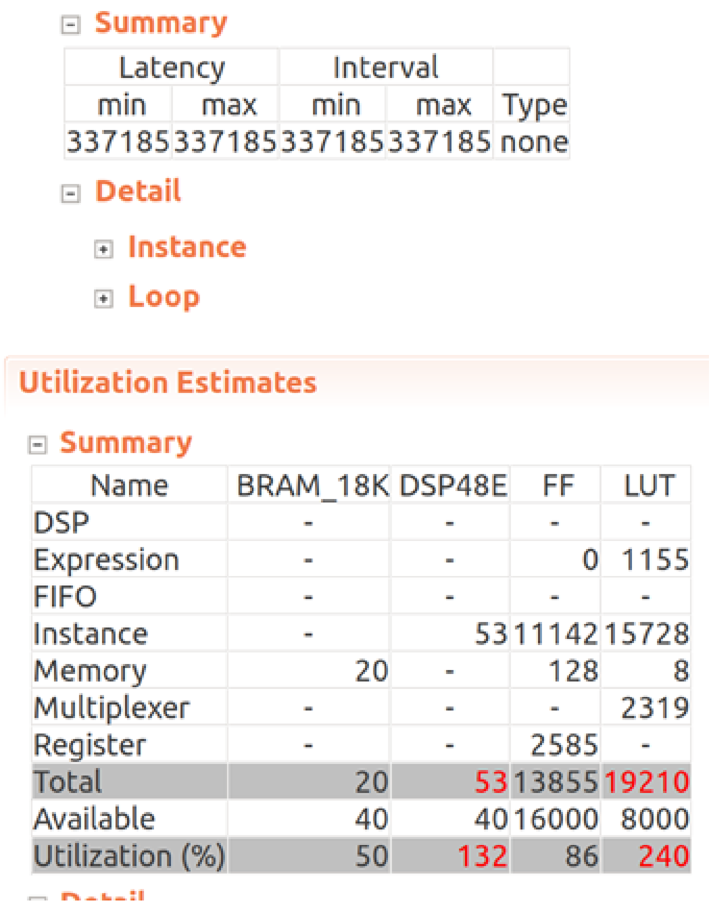
In our first step design, we only declared one multiply-add device to support matrix update section and split it into three stages. However, as the three operations in matrix update section have no data dependency, we tried to declare three multiply-add devices to support these three operations parallelly compute in the same stage. With very simple modification on configuration file and design flow file can make this new design.

In our new design, the new one round latency is 6,970 cycles, which used 83,885 LUT logic slices, 135,109 FF units, and 91 DSP48E1 slices. This result has proved by changing our configuration files, we can declare more calculation devices and can achieve better performance. However, we need to use more on-chip resources.

5.11.6. Generating RTL Design by Using Vivado_HLS

In addition to comparing the performance with the original Python program, we also compare the performance of our implementation with that of the RTL code generated by Vivado_HLS. We translate the original Python program into a C program. The translated C program's execution results perfectly match with the Python program. Then we use Vivado_HLS to generate RTL design from the C program. Figure 5.10 is the latency and resource usage reported from Vivado_HLS.

From Figure 5.10 we find the Vivado_HLS generated RTL uses fewer DSP slices, FF, and LUT than my RTL design, but its latency is 47.6 times longer than my design. This is because the Vivado_HLS generated RTL performs operations in serial, and it does not need a lot of resources to support high parallel computing and dataflow pipeline. It also cannot achieve parallel computing and dataflow pipeline. That is why the Vivado_HLS generated RTL requires fewer resources than my design. However, the Vivado_HLS generated RTL has a large latency. If we set FPGA's clock to 150MHZ, the Vivado_HLS generated RTL runs for 337185 * 10000 * 6.7 = 22,591,395,000 ns = 22.5 seconds, which is 6.1 times longer than the original Python program.



Summary

Latency		Interval		
min	max	min	max	Type
337185	337185	337185	337185	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1155
FIFO	-	-	-	-
Instance	-	53	11142	15728
Memory	20	-	128	8
Multiplexer	-	-	-	2319
Register	-	-	2585	-
Total	20	53	13855	19210
Available	40	40	16000	8000
Utilization (%)	50	132	86	240

FIGURE 5.10. Latency and resource usage reported by Vivado_HLS

5.12. Summary

In this chapter, we introduced four things:

1. Our method of detecting HPC system anomaly events from the event block lists we have converted from the original system log files by using SLEBD.

2. Our method of collecting all the events from one specific type of event, smooth the bias between the sequences before success and fail events, and use the LSTM model to cluster the event sequences.

3. Our method of using TSM to detect each type of event's critical workflow, then use this workflow to predict such type of event's occurrence and use its LSTM model to predict it's occurring success or failure.

4. Our method of implementing Python level RNN designs to RTL level designs. We

packaged RNN needed floating-point calculation operations into devices with destination address FIFO and created an RTL level design code generating system. Users can design and train their RNN model to verify its feasibility and calculation accuracy, then use our RTL generating system to generate pure RTL level codes by inputting simple configuration files. The user can declare as many calculation devices as they want within the range of FPGA can afford. And the user can achieve parallelism computing besides operations which have no data dependency like LSTM function (9) into the same calculation stage. Our system generates RTL level codes without any help from the HLS tool. Our automatic generated RTL designs can be synthesis into the bitstream and deploy on FPGA chips without using SOC running on the ARM processor. The experiment shows the RTL level RNN design can significantly increase the RNN training performance.

In our above experiments, we had a good result. We detected several types of event-level failure, we proved that the LSTM sequence clustering method could successfully cluster the system event sequences extracted by SLEBD, and we proved the critical workflow which detected by using TSM can help us to predict future event's occurrence.

However, we still faced some difficulty:

1. We can only find a very small number of events which can be used to do our Deep Learning experiment from the SLEBD event list. That is because the dataset we are using is not very big and some specific events only occurred very little times (including success and fail), not enough to create LSTM model.

2. Setting the most effective and accurate LSTM model which can work on most Event Block's sequences are challenging. For example, for one specific event A, the event sequences occurred before it will be different from the event sequences occurred before the other event B. So, the LSTM model trained on event A's event sequences will not work on event B.

3. Training an LSTM model by using Keras is very time-consuming. That is why we try to research to use FPGA to accelerate RNN performance.

4. As we have mentioned, the distribution between success and fail event are biased.

So, the ratio of how much-failed sequence been emphasized by using SMOTE in the learning set can achieve the most accuracy also needs us to do research.

5. Our basic RTL level design and RTL generate system are completed. However, to generate RTL level designs, we still need the user to assign what calculation device and temporary registers they will use in the configuration file and convert their Python level RNN design in the design flow file. We plan to create a Python code compiler to compile and evaluate the device and register usage directly from Python design in the future.

6. The RNN design we are using in our experiment is small. We have provided implementing RNN on FPGA and train the RNN model on-ship in three main sections by using our way is feasible, and our experiment's accuracy and performance make us satisfied. However, by seeing the FPGA resource usage report, we found we still have space to extend our design bigger. We plan to implement a larger RNN/LSTM design on FPGA in the future.

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH PLAN

6.1. Conclusions

In my dissertation research, I have completed the following milestones.

1. I have developed an HPC syslog Event Block pattern grouping and anomaly detect tool, called SLEBD, which can group syslog messages and convert the original unstructured syslog messages into structured event lists. Such event lists can facilitate anomaly detection.

2. By using the event blocks extracted by SLEBD, we design a topology-aware sequence mining (TSM) method to discover sequence patterns among events to characterize the execution sequences of a system. The sequence patterns produced by TSM help us perform event sequence-level anomaly detection and event prediction.

3. SLEBD can detect event-level anomalies, that is anomalies which are related to a type of Event Block. This feature allows us to group event sequences that appear before such type of Event Block or the occurrence of an anomaly. We have also explored recursive neural networks to develop an LSTM model to cluster such event sequences. Our experimental results show the LSTM model can effectively predict anomalous events.

4. Running deep learning algorithms on CPU and GPU are time-consuming and power-hungry. To address this issue, I have developed a design flow which can automatically generate synthesis-able RTL level designs from Python deep learning programs. I have implemented this design on a VC707 FPGA board. Experimental results show the RTL design efficiently explores dataflow pipelining and parallel computing and significantly reduces the execution time, energy consumption, and development efforts.

6.2. Future Research Plan

Built on top of the frameworks, tools, and methods that I have developed, my research will be extended in the following directions.

1. Our experiments on LSTM only test if LSTM can help us cluster event sequences. We need to do more experiment to find the optimal configuration(s) of LSTM models or

SMOTE's fold ratio to achieve the best accuracy.

2. We have detected some types of event-level anomalies. However, we do not know why those anomalies happen in production systems. I plan to use LSTM models to find the root causes. The root causes will allow us to predict future anomalies and even design countermeasures to prevent them.

3. We use TSM to discover critical execution workflows which help us predict occurrences of event blocks is in the future. Moving further, I plan to use an event block's LSTM model to predict if an event will successfully occur or not. More experiments will be conducted.

4. The current test cases of RNN on FPGA is still small and not suitable for real-world deployment and applications. I plan to develop a larger-scale RNN design, convert this design to RTL-level implement, and test it on FPGA.

5. Transferring a large amount of data between a computer and an FPGA board is not trivial. I am researching on how to use the PCIe IP cores from Vivado to transfer data to and from FPGA through the VC707 PCIe port. If succeed, we will be able to test larger-scale RNN designs on large, real-world data sets.

REFERENCES

[1] https://pypi.org/project/pyfpgrowth/.

[2] *http://docs.h5py.org/en/stable/*.

[3] *http://portal.nersc.gov/project/m888/resilience/*.

[4] *https://github.com/chuanconggao/prefixspanpy/blob/master/prefixspan.py*.

[5] *https://github.com/ovis-hpc/ovis*.

[6] *https://github.com/rangeonnicolas/prefixspan/blob/master/prefixspan.py*.

[7] *https://harnesscloud.github.io/2015-07-15-feltham/maxeler/codecarpentry-maxelerdataflow1.pdf*.

[8] *https://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/*.

[9] *https://keras.io/*.

[10] *https://keras.io/preprocessing/sequence/*.

[11] *https://pytorch.org/*.

[12] *https://www.accellera.org/downloads/standards/systemc*.

[13] *https://www.nltk.org/*.

[14] *https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/*.

[15] *https://www.tensorflow.org/*.

[16] *https://www.top500.org/lists/2019/06/*.

[17] *https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html*.

[18] *https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf*.

[19] *http://www.numpy.org/*.

[20] Whitehead N; Fit-Florea A, *Precision & performance: Floating point and ieee 754 compliance for nvidia gpus*, 2011.

[21] E. Grosicki; H. El Abed, *Icdar 2009 handwriting recognition competition*, July 2009, pp. 1398–1402.

[22] R. Agrawal and R. Srikant, *Fast algorithms for mining association rules*, 1994, In Proc. 1994 Int.Conf VeryLarge Data Based (VLDB).

[23] Rakesh Agrawal and Ramakrishnan Srikant, *Fast algorithms for mining association rules*, 1994, Proceedings of the 20th International Conference on Very Large Data Bases (VLDB).

[24] M. Rosvall; C. T. Bergstrom, *Maps of random walks on complex networks reveal community structure*, 2008, National Academy of Sciences, Volume 105 Issue 4, pp. 1118.

[25] Zongze Li; Song Fu; Sean Blanchard, *Converting unstructured system logs into structured event list for anomaly detection*, 2018, The 13th IEEE International Conference on Availability Reliability and Security (ARES).

[26] A. Ul-Hasan; S. B. Ahmed; F. Rashid; F. Shafait; T. M. Breuel, *Offline printed urdu nastaleeq script recognition with bidirectional lstm networks*, Aug 2013, Proc. 12th Int. Conf. Document Anal. Recognit. (ICDAR), pp. 1061-1065.

[27] Ziming Zheng; Li Yu; Wei Tang; Zhiling Lan; R. Gupta; N. Desai; S. Coghlan; D. buettner, *Co-analysis of ras log and job log on blue gene/p*, 2011, Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS).

[28] Jeatrakul P; Wong K.W; Fung C.C, *Classification of imbalanced data by combining the complementary neural network and smote algorithm*, 2010, Neural Information Processing. Models and Applications. ICONIP.

[29] Catello Di Martino; Marcello Cinque; Domenico Cotroneo, *Assessing time coalescence techniques for the analysis of supercomputer logs*, 2012, Proc. of IEEE/IFIP DSN.

[30] Andre Xian Ming Chang; Berin Martini; Eugenio Culurciello, *Recurrent neural networks hardware implementation on fpga*, 2015, arXiv preprint arXiv:1511.05552,.

[31] George A; Binu D, *An approach to products placement in supermarkets using prefixspan algorithm*, 2013, Journal of King Saud University-Computer and Information Sciences.

[32] J. Han; J. Pei; B. Mortazavi-Asl; Q. Chen; U. Dayal; and M.-C. Hsu, *Freespan: Frequent pattern-projected sequential pattern mining*, 2000, In Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD).

[33] Mikolov T; Kombrink S; Burget L; et al, *Extensions of recurrent neural network language model*, 2011, Acoustics Speech and Signal Processing (ICASSP); IEEE International Conference on. IEEE.

[34] Taerat N; Brandt J; Gentile A; et al, *Baler: deterministic, lossless log message clustering tool.*, 2011, Computer Science Research Development 26: 285.

[35] Min Du; Fei fei Li; Guineng Zheng; VivekSrikumar, *Deeplog: Anomaly detection and diagnosis from system logs through deep learning*, 2017, CCS '17 Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security Pages 1285-1298.

[36] K Pedretti S Olivier G Shipman W Shu K Ferreira, *Exploring mpi application performance under power capping on the cray xc40 platform*, 2015, Proc. of EuroMPI.

[37] E Baseman; S Blanchard; Z Li; S Fu, *Eelational synthesis of text and numeric data for anomaly detection on computing system logs*, 2016, Proc. of IEEE International Conference on Machine Learning and Applications (ICMLA).

[38] Q. Guan; N. Debardeleben; S. Blanchard; S. Fu, *F-sefi A fine-grained soft error fault injection tool for profiling application vulnerability*, 2014, Procȯf IEEE IntlṖarallel & Distributed Processing Symposium (IPDPS).

[39] Q. Guan; S. Fu, *A data mining framework for autonomic anomaly identification in networked computer systems*, 2010, Proc. of IEEE IPCCC.

[40] Qiang Guan; Song Fu, *Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures*, 2013, Proc. of IEEE SRDS.

[41] Z. Zhang; Q. Guan; S. Fu, *An adaptive power management framework for autonomic resource configuration in cloud computing infrastructures*, 2012, Proc. of IEEE IPCCC.

[42] Z. Zhang; S. Fu, *Characterizing power and energy usage in cloud computing systems*, 2011, IEEE CloudCom.

[43] Zongze Li; Song Fu, *Accelerating rnn on fpga with efficient conversion of high-level designs to rtl*, 2019, In the proceedings of IEEE BigData 2019.

[44] ______, *Accelerating rnn on fpga with efficient conversion of high-level designs to rtl*,

2020, Submitted to the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS).

[45] Raghul Gunasekaran; Sarp Oral; David Dillow; Byung Park; Galen Shipman; Al Geist, *Real-time system log monitoring/analytics framework*, 2011, Proc. of Annual Cray User Group Conference (CUG).

[46] Ziming Zheng; Zhiling Lan; Byung H. Park; Al Geist, *System log pre- processing to improve failure prediction*, 2009, Proc. of IEEE International Conference on Dependable Systems and Networks (DSN).

[47] J. Hong; C. Liu; M. Govindarasu, *Integrated anomaly detection for cyber security of the substations*, July 2014, IEEE Transactions on Smart Grid; vol. 5; no. 4, pp. 1643-1653; doi: 10.1109/TSG.2013.2294473.

[48] Hu M; Zheng G; Wang H, *Improvement and research on aprioriall algorithm of sequential patterns mining*, 2013, In Proceedings of International Conference on Information Management, Innovation Management and Industrial Engineering.

[49] Michiel Hazewinkel, *Topology general*, 2011, Encyclopedia of Mathematics; Kluwer Academic Publishers.

[50] S. He; J. Zhu; P. He and M. R. Lyu, *Experience report: System log analysis for anomaly detection*, 2016, 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), Ottawa, ON, pp. 207-218.

[51] Guan Y; Yuan Z; Sun G; Cong J, *Fpga-based accelerator for long short-term memory recurrent neural networks*, 2017, 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 629–634 (2017). doi: 10.1109/ASPDAC.2017.7858394.

[52] Hochreiter S; Schmidhuber J, *Long short-term memory*, 1997, Neural Comput. 9, 1735",780.

[53] J. Fonseca J. C. Ferreira, *An fpga implementation of a long short-term memory neural network*, 2016, ReConFigurable Computing and FPGAs (ReConFig) 2016 International Conference on, IEEE, pp. 1-8,.

[54] M. Chen; A. X. Zheng; J. Lloyd; M. I. Jordan; and E. Brewer., *Failure diagnosis*

using decision trees, 2004, In ICAC'04: Proc. of the 1st International Conference on Autonomic Computing, pages 36–43. IEEE.

[55] Wei Xu; Ling Huang; Armando Fox; David Patterson; Michael Jordan, *Mining console logs for large-scale system problem detection*, 2009, Proc. of ACM Symposium on Operating Systems Principles (SOSP).

[56] S. Alspaugh; Archana Ganapathi; Marti A; Hearst Randy Katz, *Analyzing log analysis: An empirical study of user log mining*, 2014, Proc. of USENIX Large Installation System Administration Conference (LISA).

[57] Gisung Kim; Seungmin Lee; Sehun Kim, *A novel hybrid intrusion detection method integrating anomaly detection with misuse detection*, March 2014, Expert Systems with Applications Volume 41, Issue 4, Part 2, Pages 1690-1700.

[58] Ana Gainaru; Franck Cappello; Stefan Trausan-Matu; Bill Kramer, *Event log mining tool for large scale hpc systems*, 2011, Proc. of Euro-Par.

[59] Hui Yan Cheah ; Suhaib A. Fahmy ; Douglas L. Maskell ; Chidamber Kulkarni, *A lean fpga soft processor built using a dsp block*, February 22-24, 2012, Monterey, California, USA, Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays.

[60] Wang L, *Directed acyclic graph*, 2013, Encyclopedia of Systems Biology: 574-574.

[61] Sandia National Labs, *http://hpc.sandia.gov/aces/*.

[62] Li Yu; Ziming Zheng; Zhiling Lan, *Filtering log data: Finding the needles in the haystack*, 2012, Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).

[63] Zongze Li; M Davidson; S Fu; S Blanchard; M Lang, *Event block identification and analysis for effective anomaly detection to build reliable hpc systems*, 2018, 20th IEEE International Conference on High-Performance Computing and Communications (HPCC).

[64] Zongze Li; Matthew Davidson; Song Fu; Sean Blanchard; Michael Lang, *Event block analysis for effective anomaly detection on production hpc systems*, 2017, Poster ac-

cepted by ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC).

[65] Zongze Li; Song Fu; Sean Blanchard; Michael Lang, *Topology-aware event sequence mining for understanding hpc system behavior and detecting anomalies*, 2019, Accepted by the 21st IEEE International Conference on High-Performance Computing and Communications (HPCC).

[66] JianGuang Lou; Qiang Fu; Yi Wang; Jiang Li, *Mining dependency in distributed systems through unstructured logs analysis*, 2010, ACM SIGOPS Operating Systems Review; Volume 44 Issue 1.

[67] Qiang Fu; JianGuang Lou; Yi Wang; Jiang Li, *Execution anomaly detection in distributed systems through unstructured log analysis*, 2009, Proc. of International conference on Data Mining (ICDM).

[68] Qinru Qiu; Qing Wu; Martin Bishop; Robinson E Pino; Richard W Linderman, *A parallel neuromorphic text recognition system and its implementation on a heterogeneous high performance computing cluster*, 2013, Computers, IEEE Transactions on, 62(5):886–899.

[69] Ronaldo Messina; Jérme Louradour, *Segmentation-free handwritten chinese text recognition with lstm-rnn*, 2015, 3th International Conference on Document Analysis and Recognition (ICDAR).

[70] Xiaoyu Fu; Rui Ren; Jianfeng Zhan; Wei Zhou; Zhen Jia; Gang Lu, *Logmaster: Mining event correlations in logs of large-scale cluster systems*, 2012, Proc. of IEEE Symposium on Reliable Distributed Systems (SRDS).

[71] Ab Rahman; A.AH; Prihozhy A; Mattavelli M, *J image video proc. (2011) 2011: 19*, 2019, https://doi.org/10.1186/1687-5281-2011-19.

[72] David Niju, *Law of total probability*, 2008.

[73] G Pandeeswari N; Kumar, *Anomaly detection system in cloud environment using fuzzy clustering based ann*, 2016, Mobile Netw Appl (2016) 21: 494. https://doi.org/10.1007/s11036-015-0644-x.

[74] W. Xu; L. Huang; A. Fox; D. Patterson and M.I. Jordon, *Detecting large-scale system*

problems by mining console logs, 2009, In SOSP'09: Proc. of the ACM Symposium on Operating Systems Principles.

[75] Zdzislaw Pawlak, *Rough sets decision algorithm and bayes's theorem*, 2002, European Journal of Operational Research 136(1):181-189.

[76] J Han; J pei; Y Yin, *Mining frequent patterns without candidate generation*, 2000, Proceeding SIGMOD '00 Proceedings of the 2000 ACM SIGMOD international conference on Management of data Pages 1-12.

[77] B Zheng; A Tiwari; N Vijaykumar; G Pekhimenko, *Ecornn: Efficient computing of lstm rnn training on gpus*, 2018, The 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2018).

[78] D. Svozil; V. Kvasnicka; J. Pospichal, *Introduction to multi-layer feed-forward neural networks*, 1997, Chemometrics Intell Lab Systems, 39 (1997), pp. 43-62.

[79] Alan Graves; Abdel rahman Mohamed; and Georey Hinton, *Speech recognition with deep recurrent neural networks*, 2013, In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pages 6645–6649. IEEE,.

[80] Hasim Sak; Andrew Senior; and Francoise Beaufays, *Long short term memory recurrent neural network architectures for large scale acoustic modeling*, 2014, In Proceedings of the Annual Conference of International Speech Communication Association (INTERSPEECH).

[81] R. Srikant and R. Agrawal, *Mining quantitative association rules in large relational table*, 1996, In Proc. of ACM International Conference on Management of Data (SIGMOD).

[82] Adam Oliner; Jon Stearley, *What supercomputers say: A study of five system logs*, 2007, Proc. of IEEE/IFIP DSN.

[83] Zongze Li; Xiaoguang Tian, *An exploratory study of long short-term memory on consumer complaints to financial services*, 2019, Presentation by the 50th Southwest Decision Sciences Institute (SWDSI).

[84] Pei J; Han J; Mortazavi-Asl; B Pinto; H Chen; Q Dayal U and Hsu M.-C, *Prefixspan:*

Mining sequential patterns efficiently by prefix-projected pattern growth, 2001, In Proc. of International Conference on Data Engineering (ICDE).

[85] R Vaarandi, *Data clustering algorithm for mining patterns from event logs*, 2003, Proc. of IEEE Workshop on IP Operations and Management.

[86] Sicheng Li; Chunpeng Wu; Hai Li; Boxun Li; Yu Wang; and Qinru Qiu, *Fpga acceleration of recurrent neural network based language model*, 2015, In Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on, pages 111–118. IEEE.

[87] P. Bodik; M. Goldszmidt; A. Fox; D. B. Woodard and H. Andersen., *Fingerprinting the datacenter: automated classication of performance crises*, 2010, In EuroSys'10: Proc. of the 5th European conference on Computer systems, pages 111–124. ACM.

[88] Y. Liang; Y. Zhang; H. Xiong and R. Sahoo, *Failure prediction in ibm bluegene/l event logs*, 2007, In ICDM'07: Proc. of the 7th International Conference on Data Mining.

[89] J. Lou; Q. Fu; S. Yang; Y Xu and J. Li., *Mining invariants from console logs for system problem detection*, 2010, In ATC'10: Proc. of the USENIX Annual Technical Conference.

[90] S. Fu; C.Z. Xu, *Exploring event correlation for failure prediction in coalitions of clusters*, 2007, IEEE conference on Supercomputing (SC).

[91] B Li; E Zhou; B Huang; J Duan; Y Wang; N Xu; J Zhang; H Yang, *Large scale recurrent neural network on gpu*, 2014, 2014 International Joint Conference on Neural Networks (IJCNN).

[92] P.A. Scarf; M.M. Yusof, *A numerical study of tournament structure and seeding policy for the soccer world cup finals*, 2011, Statistica Neerlandica, 65 (2011), pp. 43-57.

[93] Mohammed J. Zaki, *Spade: An efficient algorithm for mining frequent sequences*, 2011, Machine Learning, v.42 n.1-2, p.31-60.

[94] J Cong; B Liu; S Neuendorffer; J Noguera; K Vissers; Z Zhang, *High-level synthesis for fpgas: From prototyping to deployment*, 2011, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. pp. 473-491.

[95] Q. Lin; H. Zhang; J.G. Lou; Y. Zhang and X. Chen, *Log clustering based problem iden-*

tication for online service systems., 2016, In ICSE'16: Proc. of the 38th International Conference on Software Engineering.