Prediction of Deadlocks in Concurrent Programs Using Neural Network

Elmira Hasanzad
Department of Computer Engineering, University of Kashan, Iran
elm.hasanzade@grad.kashanu.ac.ir
Seyed Morteza Babamir*
Department of Computer Engineering, University of Kashan, Iran
babamir@kashanu.ac.ir

Abstract
The dependability of concurrent programs is usually limited by concurrency errors like deadlocks and data races in allocation of resources. Deadlocks are difficult to find during the program testing because they happen under very specific thread or process scheduling and environmental conditions. In this study, we extended our previous approach for online potential deadlock detection in resources allocated by multithread programs. Our approach is based on reasoning about deadlock possibility using the prediction of future behavior of threads. Due to the nondeterministic nature, future behavior of multithread programs, in most of cases, cannot be easily specified. Before the prediction, the behavior of threads should be translated into a predictable format. Time series is our choice to this conversion because many Statistical and Artificial Intelligence techniques can be developed to predict the future members of the time series. Among all the prediction techniques, artificial neural networks showed applicable performance and flexibility in predicting complex behavioral patterns which are the most usual cases in real world applications. Our model focuses on the multithread programs which use locks to allocate resources. The proposed model was used to deadlock prediction in resources allocated by multithread Java programs and the results were evaluated.

Keywords: Detecting Potential Deadlocks, Time Series Prediction, Multithread Programs, Behavior Extraction.

1. Introduction

Multithread programs are becoming increasingly common. Since multi-core processor generation has brought more cores, developers must parallelize programs if they want to speed the program execution up. However, applying concurrency method causes some integrity and mutual exclusion issues in allocating resources. To resolve them, locking mechanism was developed. However, this mechanism leads to some other known problems like *starvation* and *deadlock* in resources allocated by concurrent systems. Detection of such errors in the program testing phase may be difficult since they often occur in the special sequence of events [1]. This is why that, these errors are sensitive to timings, workloads, compiler options and memory models. In addition, if a deadlock or *data race* in resource allocation emerges in the testing phase, it is difficult to find out its root cause; because in a multithread program, even if there is a deadlock between some threads in allocating resources at runtime, other threads still can run. The effects of such a situation can manifest itself millions of cycles after occurring the error. Deadlock is a common form of bug in software nowadays. Sun's bug database showed that 6,500 bug reports out of 198,000 contain "deadlock" [2]. Main reasons of deadlock are: (1) software systems are often written by diverse programmers; therefore, it is difficult to follow a lock order discipline in allocating resources, (2) programmers often introduce deadlocks when they fix race conditions by adding new locks and (3) using third-party software such as plug-in because the third-party software may not follow the locking discipline followed by the parent software [3]. This is why that "deadlock avoidance" techniques became unusable. Such techniques are simple in theory but so restrictive in real application.

Therefore "detecting potential deadlocks" became an acceptable method to solve deadlock problem in resources allocation. "Potential deadlock detection" techniques are Online or Offline, which Online ones try to find the concurrency errors at runtime. Such approaches mostly use a monitor to observe the program

---

execution and based on the observations, they decide about the error possibility. In comparison with offline techniques, online ones have the following advantages:

1. They only visit feasible paths of program executions and have accurate views of the values [1],

2. Because of their accurate view, they generate fewer false alarms. False alarm means a fake report of an error (in our case, a deadlock),

3. They don't need considerable programmer effort,

4. These approaches are language independent meaning that the solution is not depended on features of a specific programming language.

In this paper, we demonstrate and extend a novel online potential deadlock detection approach, whose base was presented in [4]. It was based on the prediction of processes or threads behavior at runtime and dealt with reasoning about the deadlock possibility in the future. In this work, we introduce *time series* analysis approaches in configuring prediction parameters. Also, we include the environmental conditions in predicting the threads behavior to improve the correctness of obtained results. We obtained considerable improvement in detecting potential deadlocks in comparison with our previous work.

This paper is organized as follows: Section 2 overviews the related works and our proposed model is discussed in Section 3. We analyze our approach and evaluate its results in Section 4. We draw conclusions in Section 5.

2. Related works

As mentioned in the previous section, our approach is based on finding potential deadlocks in allocating resources at runtime using program behavior extraction and time series prediction. Therefore in this section, we first overview online approaches detecting potential deadlocks in resources allocated by concurrent programs. Afterwards, we discuss different approaches used time series for the prediction.

2.1 Online potential deadlock detection

Informally, in multi-threaded systems used shared memory, deadlocks in allocating resources happen when a set of threads are blocked forever; this is because each thread in the set is waiting to acquire a lock held by some thread [2]. Generally in a concurrent system, the order of acquiring and releasing locks in

allocating and freeing resources can be described as a directed graph where nodes indicate locking resources so that an edge from node A to node B means the system has locked resource A and is waiting for resource B. There will be a deadlock in allocation of resources if a circle is found in the graph. Lock graphs and their variations have been used for detection of deadlocks in resources allocated by concurrent programs.

GoodLock algorithm [5] is an approach to detect potential deadlocks in multithread programs. It only detects potential deadlocks caused through interleaving locks by just two threads. To overcome this limitation, some generalized versions of GoodLock algorithm was presented in [6] and [7] which detect potential deadlocks caused by any number of threads. Their approach address programs that use bloc and non-block structured locking.

In [8], authors constructed an online lock graph and found specific paths, which named "not guarded SCC (strongly connected components)". "Not guarded SCC" indicates one or more potential deadlocks because there can be several cycles in the SCC. They tried to exhibit the deadlocks using injection of noises in the SCCs. A noise is inserted to create a delay to acquire a lock; accordingly, they raised the probability of manifesting the real deadlocks. Although this approach is based on GoodLock algorithm, its advantage over one that presented in [6] and [7] is regarding different runs. The Goodlock looks at the scope of one process run. This means, when a cycle in the graph is caused by sequences of two different runs, Goodlock can't detect.

GoodLock algorithm also was used in combination with other techniques to find the potential deadlock at runtime such as DEADLOCKFUZZER [2]. This approach consists of two phases. In the first phase, a simple variant of the Goodlock algorithm, called informative Goodlock, was used to discover cycles of potential deadlock. In the second phase, DEADLOCKFUZZER executed the program at a random schedule in order to create a real deadlock corresponding to a cycle which reported in the previous phase. In [3] "deadlock immunity" concept was introduced for avoiding occurrence of deadlocks occurred in the past. When a deadlock occurs for the first time, the deadlock information is saved in a "context" in order to avoid the similar contexts in future runs. This approach achieved "immunity" against the corresponding deadlocks. To avoid deadlock whose context has been already seen, the approach changed the schedule of threads. As the several deadlocks occur, the numbers of contexts

increases; therefore it can avoid a wider range of deadlocks. However, if a deadlock does not have a pattern similar to one that already encountered, the approach cannot avoid its occurrence.

As we mentioned in Section Introduction, all the online deadlock detection approaches share some common advantages like: language independency, accurate views of the values, fewer false alarms and programmer efforts. But online techniques suffer from some disadvantages too. The most common problem is imposing the heavy overhead at runtime, both in time or space. All the mentioned techniques try to extract some relevant traces before their real execution based on observed current execution. In fact, they *pre-run* these extracted traces to find out whether there is any deadlock in the trace or not. This phase is time consuming because of extracting and running relevant traces. Also, one of the important steps for online techniques is the code *instrumentation*. Code instrumentation means modifying the target code for runtime monitoring code behavior. This step could be time consuming too and for the legacy codes it is more difficult. Sometimes, online potential deadlock detection techniques may show deadlocks late. This leads to finding a potential deadlock when the rollback mechanism is impossible because of some preclude actions like I/O.

2.2 Time series prediction approaches

Because of weaknesses of the online potential deadlock detection techniques mentioned in previous section, we proposed a novel online approach which targets the increase of performance, decrease of instrumentation and the enhancement of the prediction [4]. In this approach, we needed to predict future members of the generated *time series* at runtime. In general, time series prediction techniques can be classified in two categories: statistical and neural network based techniques. The statistical prediction techniques such as Autoregressive (AR), Moving Average (MA) and combined AR and MA (ARIMA) [9] have several limitations, such as inefficiency for real world problems which are often complex and *nonlinear*. This is due to the fact that these techniques assume that a time series is generated by a *linear* process. Thus, they are called linear statistical predictors.

The *nonlinear* statistical predictors such as predictors, "threshold", "exponential", "polynomial" and "bilinear" were proposed to increase of the prediction precision [9],[10]. However, the selection of a suitable nonlinear model and the computation of its parameters are difficult tasks for a practical problem especially when the time series behavior is non-

deterministic. Moreover, it has been shown that the capability of the nonlinear model is limited, because it is unable to provide a long-term prediction [11].

In recent years, artificial intelligence tools have been extensively used for time-series based prediction [12, 13]. In particular, artificial neural networks are frequently exploited for time-series based prediction of systems behavior. A neural network is an information processing system that is capable of treating complex problems of pattern recognition, dynamic and nonlinear processes. In particular, it can be an efficient tool for prediction applications. The advantage of neural networks based approaches over statistical ones is the capability of learning and accordingly generalization of their knowledge [14]. Also the neural networks are based on training and in many cases their prediction results are more precise, even if the training set has considerable noise [14]. These approaches are much more suitable for real world problems which *do not have specific rules*.

There are some composite approaches which try to take the advantage of the accuracy of statistical models and the generality of neural network approaches. In [15], authors composed statistical model ARIMA and *a feed-forward neural network* to forecast time series. A feed forward network is a type of neural networks where all of its connections have the same direction [16]. This composition could be efficient in predicting some *well-known* time series. However, in the case of other time series, finding the proper value for statistical part of the composition is a difficult task and wrong values could affect the accuracy of prediction. Also it has been proved that the capability of *recurrent neural networks* is equivalent to the Turing machine [17]. Recurrent network is a class of neural network where connections between the layers of it could be backward or forward [16]. Therefore recurrent networks can approximate any function by learning from the function inputs and outputs.

3. Potential deadlock prediction

In our previous work we proposed an online predictive model to detect the potential deadlocks in multithread programs which is the basis of our approach [4]. Figure 1 shows our proposed model architecture. This model is consists of four components which are collaborating together at runtime. In this work, we aim to extend the basis model, indeed we extend "predictor" component, to be able to generate much more accurate prediction.

3.1 The basis of proposed model

We used dependency graph in our model which nodes are the concurrent threads or processes. There is an edge from node A to node B if and only if thread A wants to acquire a lock which held by thread B and A has to wait until B release the lock, after that the edge will be erased. There is a deadlock in the system if there is a cycle in dependency graph. Therefore, except requesting or releasing the locks, other behavior of threads does not play any role in deadlock occurrence. For this reason in our proposed model we target only the instructions which are related to acquiring or releasing the locks. We named this type of instructions deadlock-prone behavior. The main difference between our approach and other online potential deadlock detection approaches which we explained in Section 2, is that we try to predict the future deadlock-prone behavior of threads at runtime rather than try to abstract different execution traces from the current execution by changing threads schedules or noise injection. If we could have an accurate view of future deadlock-prone behavior of threads then we can accurately result about the deadlock occurrence in the future [4].
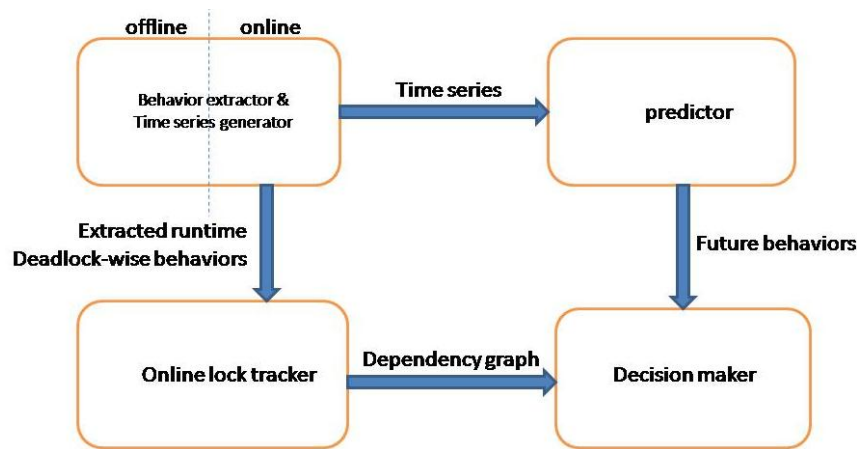

Fig 1. The basis of proposed model [4]

The start point of our model is the "Behavior extractor & Time series generator" component. Actually this component is composed of two elements:

Two annotated Java functions: one for extracting deadlock-prone behavior and another for converting extracted behavior to univariate time series. Figure 2 shows these two functions: *1- extractor & convertor () 2- this Period behaviors ()*. The first one task is catching lock () and unlock () at runtime and the second one task, is appending these instructions to the proper time series.

1. @AfterRunning(pointcut = "execution(* java.util.concurrent.locks.unlock(..))")
2. @Before(pointcut = "execution(* java.util.concurrent.locks.lock(..))")
3. public void extarctor&convertor (JoinPoint joinPoint) {
4. String functionName=joinPoint.getSignature().getName();
5. If(functionName.eguals("lock")){
6. thisPeriodBehaviors("1", Arrays.toString(joinPoint.getArgs()),this.name);
7. }
8. Else{
9. thisPeriodbehaviors("2", Arrays.toString(joinPoint.getArgs()),this.name);
10. }
 }

Fig 2. Functions pseudo code

Line 1, shows an annotation which means: whenever an Unlock() instruction executed, the *extractor&convertor(...)* method, should be executed immediately. Line 2, shows an annotation which means: right before the execution of a Lock() instruction, the *extractor&convertor(...)* method, should be executed. Line 3 is the method sign and line 4, is for obtaining the name of the event which caused the *extractor&convertor(...)* method to be executed. In line 5 to 10, based on the name of event (lock or unlock), a specific character will be appended to a specific time series. In this way all the lock() and unlock() events which are

issued from threads at runtime, are caught and converted to time series.

ApectJ compiler. The task of this compiler is weaving two Java functions to the target multithread programs in the locations which are specified by annotations above the method.

We mentioned that, one of the problems in runtime verification approaches is source code instrumentation step. The instrumentation is a time consuming task and when the verification logic is complex, it could be inefficient at runtime, both in time and space. But our two Java functions which are weaved to the target multithread program, are easy and light weight thus their runtime overhead is negligible.

As we said, there are two Java functions for extracting dedicated behavior and time series generating goals. Time series is a set of observations from past until present, denoted by s(t-i) {i= 0.. P }, where P is the number of observations. Time series prediction is to estimate future observations, let's say s (t+i) for {i= 1.. N}, where N is the size of prediction window. Also, a univariate time series refers to the set of values over the time of a single quantity.

The next component in our model is "Online Lock Tracker". According to Figure 1, this component takes the deadlock-prone behavior from "Behavior extractor & Time series generator" component at runtime and draws a dependency graph. This dependency graph will be updated whenever a thread issues a deadlock-prone behavior.

The "predictor" component takes the generated time series from "Behavior extractor & Time series generator" and tries to predict the next members of the time series. In a multithread program, the order of executed instructions of a thread could be affected by other threads executions. This fact makes the concurrent systems nondeterministic thus it is hard to predict the future thread behavior. We can't assume any pre-defined generator for the time series which are representing threads behavior. This property makes the statistical prediction

techniques useless for our purpose. Because the statistical prediction techniques, assume that a time series is generated by linear or nonlinear process, but the selection of the suitable nonlinear or linear model and computation of its parameters is a difficult task for a practical problem without a priori knowledge about the time series[10]. The prediction requirements of our model lead us to use artificial intelligence prediction techniques. Time series prediction techniques which are based on AI use several Artificial Neural Networks [10]. Based on the properties of time series, there are different network topologies and learning algorithms. The selection of a proper network model and adjustment of its parameters should be carried out by considering the problem requirements.

The predictions of the "predictor" component are also in the form of time series. These predictions and current dependency graph (the output of "online lock tracker" component) are injected to the "Decision maker" component. This component is responsible for deciding about the deadlock possibility in the next period. We try to clarify our model using an example. Assume that we have four threads named T1,…,T4 and five locks named L1, …,L5. Also assume the current dependency graph is something like Figure 3 (a). This graph represents that T1 has held L1 and L3 and wants to hold L2 which held by T2 thus T1 stops proceeding and waits until T2 releases L2. Also T4 has held L5 and wants to hold L3 which held by T1 thus T4 stops proceeding. Suppose the predictions of "Predictor" component are the following:

"Predictor" component predictions → T3={ will request L5}, T2={ will request L4}

"Decision maker" takes current lock graph and predictions and composes them together to construct an abstract graph. Afterwards, decision maker searches the abstract graph to find a cycle. If so, it reports a possibility of deadlock in the next period. Figure 3 (b) shows the abstract graph of our example as a composition of predictions and dependency graph.
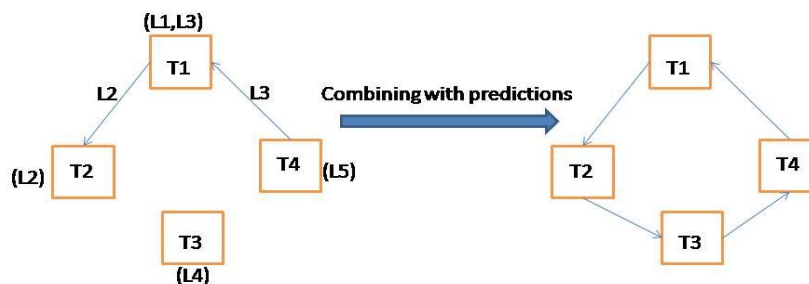

Fig 3 (a). Current lock graph   Fig 3(b). Resulted abstract graph

In our example, the abstract graph has a cycle, therefore the "Decision maker" component reports: (1) a deadlock possibility in the next period and (2) T1 to T4 as the threads will be involved in this deadlock. But, if the predictions are:

"Predictor" component predictions → T3={will request L5}, T2={ will request L4 and will release L2}

For this case, Figure 4 shows the abstract graph where there is not any cycle. Therefore, the "Decision maker" component will not report any possibility of deadlock in the next period.
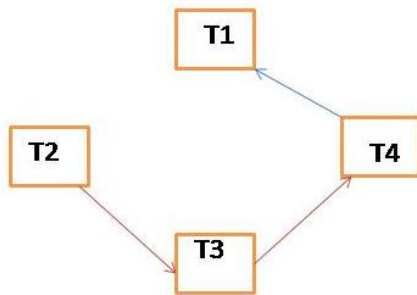

Fig 4. Resulted another abstract graph

3.2 Applying the Extensions

In our previous work we used a recurrent neural network named non-linear autoregressive (NAR) in predictor component. A NAR network tries to predict the future element of a given time series using d last values of that series [18]. That is, NAR network assumes the future element of a series is a function of its last values (Formula 1).

$$y(t) = f(y(t-1), y(t-2), \ldots, y(t-d)) \qquad (1)$$

The structure of NAR network has been shown in Figure 5. This network has d inputs, each for one of the last values of time series.
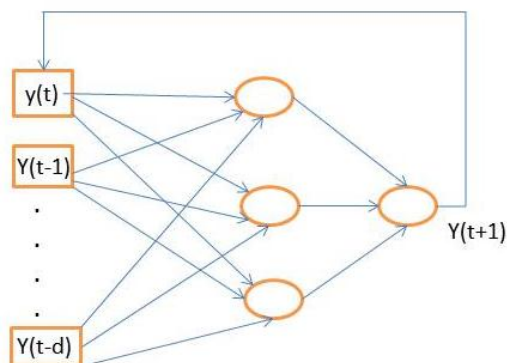

Fig 5. Structure of a NAR network

We named d as the delay parameter and it is one of the important factors which directly imposes the precision of predictions in a predictor neural network. Suppose in a time series each element is dependent on two last

elements, That is $y(t) = f(y(t-1), y(t-2))$. If we try to predict $y(t)$ using a predictor neural network such as NAR, the most accurate results will be acquired if delay $= 2$. Actually in this way the network considers two last elements in predicting the future element. In previous work we obtained the proper value for delay parameter using "try & fail" approach. That is, we gathered the runtime behavior of our multithread test program and converted them into the time series. Then we tried to predict the future members of test time series, using multiple NAR networks so that every network had a different value for delay parameter from others. After that we chose the delay value of a network which made the most precise predictions.

In this work we improve the prediction precision of our "predictor" component, by configuring the delay parameter of network using a time series analysis methods. "Embedded dimension" is a factor which determines the relationships among the past and future members of a time series [19]. The value of the "Embedded dimension" for a time series represents the optimum number of last elements which every element is dependent on. Therefore we apply the "Embedded dimension "as the delay parameter in our predictor network. To obtain the "Embedded dimension" of a time series there are multiple approaches. The most known approach is False-Nearest-Neighbor, algorithm. This algorithm was firstly proposed by Kennel et al [20]. The calculation of the "Embedded dimension" allows one to extract the process behavior parameters from the observed series of events [19]. The predictor network can be further configured according to the obtained results from False-Nearest-Neighbor (FNN), in order to remember the required number of last elements in time series.

In this work, in addition to applying "Embedded dimension" as the delay parameter, we use "Nonlinear Autoregressive with External input" (NARX) network instead of NAR network. Because in our model the major task of the predictor network is predicting threads behavior at runtime. But the behavior of threads is not completely separate from each other, actually the future behavior of each thread is affected by other threads past and current behavior. Thus we need a prediction methodology which could satisfy this requirement. As it is obvious, the NAR does not consider an external input in its prediction procedure. Because of this limitation of NAR, it may not meet our prediction requirements properly. We need a prediction method which could consider other series (that is,

other thread's behavior) in predicting a time series.

NARX network, like NAR network, is a recurrent network with an external input [21]. The main idea of recurrent networks is providing a weighted feedback connection between layers of neurons and adding time significance to entire network. Therefore, recurrent neural networks simulate a temporal memory and are suitable for tasks like prediction which need a memory for the past events. NARX network assumes the future element of a given time series is a function of its last elements and another series last elements (Formula 2).

$$y(t+1) = f\big(y(t), y(t-1), \ldots, y(t-d), x(t), x(t-1), \ldots, x(t-k)\big) \quad (2)$$

Using this external input, it is possible to predict a time series considering the last elements of the time series under prediction and also considering other time series last elements. Figure 6 shows our extended "Predictor" component.
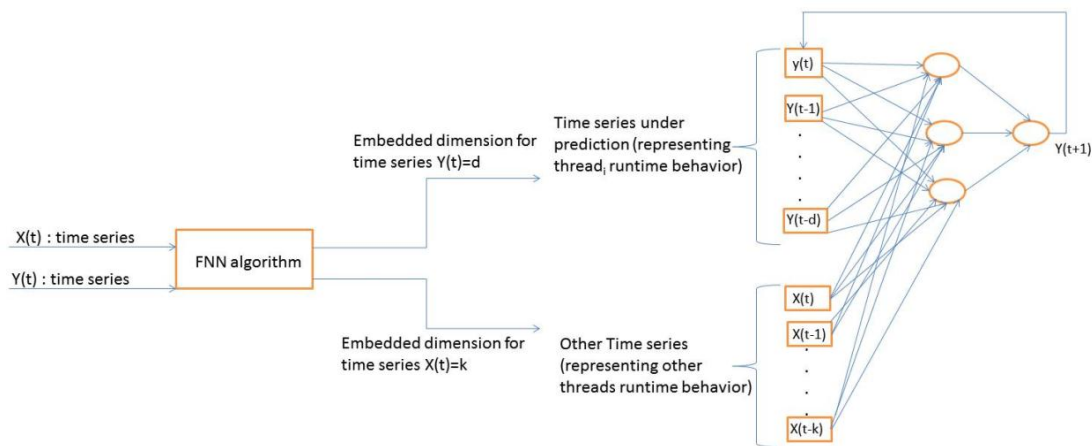

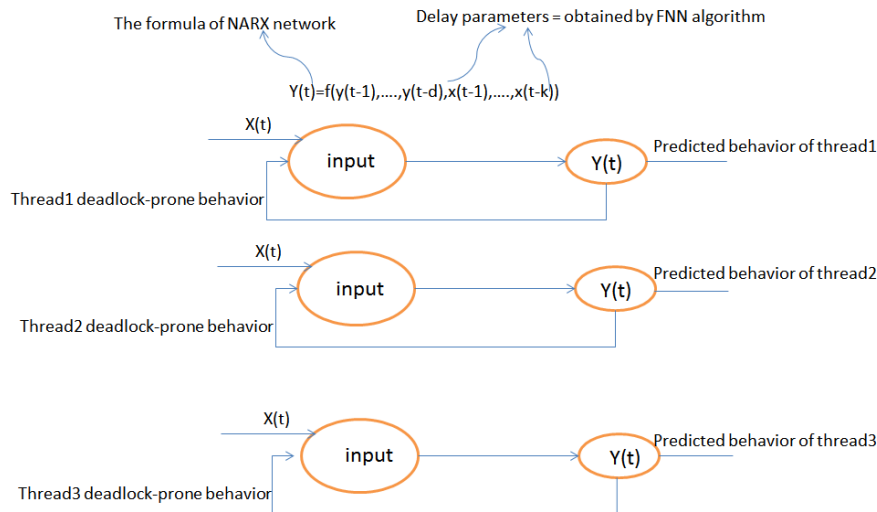Fig 6. The extended "Predictor" component

Fig 7. The NARX networks of example

To clarify the differences between the previous and current "Predictor" component at runtime, we use an example. Suppose there are three time series at runtime, then the "predictor" component will have three networks, each for predicting one of the series future elements. Each network uses some last members of target time series named y(t), and some last members of the other series named x(t), as its inputs. Therefore the new predictor networks have been shown in Figure 7, but the networks of our "previous" predictor component have been shown in Figure

8. It is obvious from the Figure 7 that, in the "predictor" component there are three NARX networks, each for predicting one of the threads (time series) future behavior. The output of a NARX network is a function of its two inputs named x(t) and y(t), therefore each network takes a target time series last behavior and another time series which represents the last behavior of the other threads. Future behavior of y(t) predicted by its past behavior and also the past behavior of x(t) and the number of last behavior obtained by FNN algorithm.

But in our previous work for this example, we there were three NAR networks and Future behavior of y(t) predicted by only its past

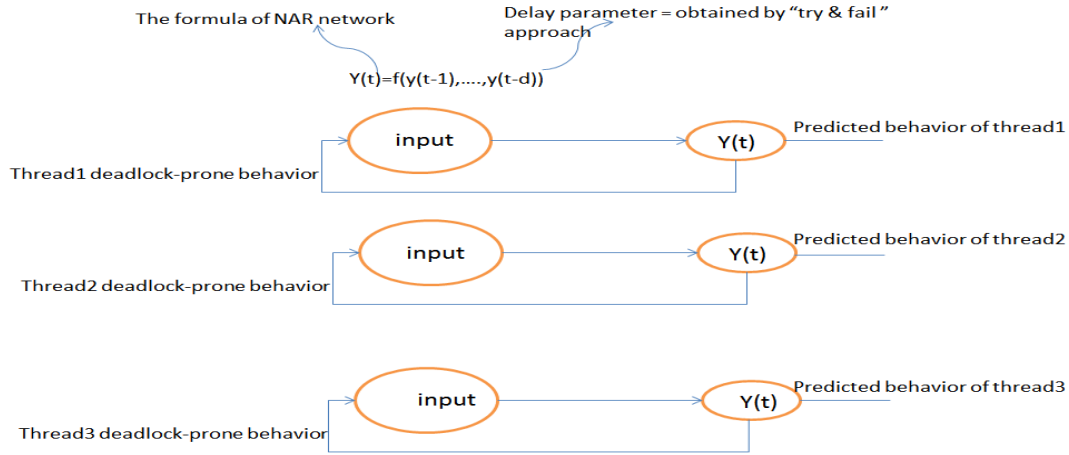behavior and the number of last behavior obtained by "try & fail " approach.

The formula of NAR network

Delay parameter = obtained by "try & fail" approach

$$Y(t)=f(y(t-1),....,y(t-d))$$

Thread1 deadlock-prone behavior → **input** → **Y(t)** → Predicted behavior of thread1

Thread2 deadlock-prone behavior → **input** → **Y(t)** → Predicted behavior of thread2

Thread3 deadlock-prone behavior → **input** → **Y(t)** → Predicted behavior of thread3

Fig 8. The NAR networks of example

4. Evaluation of the results

4.1 Experiments

Our model needs a preparation phase before that it could be used at runtime. This phase is related to configuring and training the predictor networks. For this reason first of all we should run the target multithread program for a while and gather the generated time series by "Behavior extractor & Time series generator" component during these test runs. We named these time series training phase information. Therefore we have to apply this information to train the networks and to measure the embedded dimension of time series using False-nearest-neighbor algorithm. Afterwards the obtained embedded dimensions should be used as the delay parameters in the networks. After this phase our model is ready to be used at runtime. We tested our proposed model using a Java written multithread program which consists of 50 threads and 10 shared locks. We will refer to the test multithread program as the target program in the remaining of this paper. We ran the target program 100, 500 and 1000 times. We measured

and divided the failure rate in predicting future behavior of threads in four categories:

1. Failure rate based on our previous work [4] (which we: (1) considered no embedded dimension as the delay parameter and (2) did not count the other threads behavior in predicting each thread behavior)

2. Failure rate when we count the other threads behavior in predicting each thread behavior

3. Failure rate when we include embedded dimension as the delay parameter

4. Failure rate when we: (1) include embedded dimension as the delay parameter and (2) count the other threads behavior in predicting each thread behavior

Each category was considered using different trains, validations and test sets. Tables 1 to 4 show results using Markov Chain where 15%, 20%, 30% and 40% of data were respectively used for testing and 85%, 80%, 70% and 60% of data were respectively used for validating and training the networks. Similarly, Tables 5 to 8 show results using NARX model where 15%, 20%, 30% and 40% of data were respectively used for testing and 85%, 80%, 70% and 60% of data were respectively used for validating and training the networks.

Table 1. Failure rate using markov chain with 15% test data and 85% validation and train data

Runs	Failure Rate	Test Data Percentage	Validation Data Percentage	Train Data Percentage
100	-2.16	15%	15%	70%
500	-2.3	15%	15%	70%
1000	-2.5	15%	15%	70%

Table 2. Failure rate using Markov Chain with 20% test data and 80% validation and train data

Runs	Failure Rate	Test Data Percentage	Validation Data Percentage	Train Data Percentage
100	-2.2	20%	15%	65%
500	-2.45	20%	15%	65%
1000	-2.36	20%	15%	65%

Table 3. Failure rate using Markov Chain with 30% test data and 70% validation and train data

Runs	Failure Rate	Test Data Percentage	Validation Data Percentage	Train Data Percentage
100	-2.39	30%	15%	55%
500	-2.49	30%	15%	55%
1000	-2.62	30%	15%	55%

Table 4. Failure rate using Markov Chain with 40% test data and 60% validation and train data

Runs	Failure Rate	Test Data Percentage	Validation Data Percentage	Train Data Percentage
100	-3.59	40%	15%	45%
500	-3.89	40%	15%	45%
1000	-3.87	40%	15%	45%

Table 5. Failure Rate using NARX with 15% test data and 85% validation and train data

Runs	Embedding Dimension	Environment Conditions	Test Data Percentage	Validation Data Percentage	Train Data Percentage	Failure Rate
100	NO	NO	15%	15%	70%	6.119e-1
100	YES	NO	15%	15%	70%	6.054e-1
100	NO	YES	15%	15%	70%	6.043e-1
100	YES	YES	15%	15%	70%	5.801e-1
500	NO	NO	15%	15%	70%	8.719e-1
500	YES	NO	15%	15%	70%	4.57e-1
500	NO	YES	15%	15%	70%	5.962e-1
500	YES	YES	15%	15%	70%	4.411e-2
1000	NO	NO	15%	15%	70%	8.212e-1
1000	YES	NO	15%	15%	70%	4.008e-1
1000	NO	YES	15%	15%	70%	6.009e-1
1000	YES	YES	15%	15%	70%	3.089e-1

Table 6. Failure Rate using NARX with 20% test data and 80% validation and train data

Runs	Embedding Dimension	Environment Conditions	Test Data Percentage	Validation Data Percentage	Train Data Percentage	Failure Rate
100	NO	NO	20%	15%	65%	6.093e-1
100	YES	NO	20%	15%	65%	6.043e-1
100	NO	YES	20%	15%	65%	6.085e-1
100	YES	YES	20%	15%	65%	5.221e-1
500	NO	NO	20%	15%	65%	8.332e-1
500	YES	NO	20%	15%	65%	4.431e-1
500	NO	YES	20%	15%	65%	5.101e-1
500	YES	YES	20%	15%	65%	4.01e-2
1000	NO	NO	20%	15%	65%	8.77e-1
1000	YES	NO	20%	15%	65%	3.981e-1
1000	NO	YES	20%	15%	65%	6.764e-1
1000	YES	YES	20%	15%	65%	3.821e-1

Table 7. Failure Rate using NARX with 30% test data and 70% validation and train data

Runs	Embedding Dimension	Environment Conditions	Test Data Percentage	Validation Data Percentage	Train Data Percentage	Failure Rate
100	NO	NO	30%	15%	55%	7.498e-1
100	YES	NO	30%	15%	55%	6.327e-1
100	NO	YES	30%	15%	55%	6.59e-1
100	YES	YES	30%	15%	55%	6.481e-1
500	NO	NO	30%	15%	55%	10.112e-1
500	YES	NO	30%	15%	55%	6.001e-1
500	NO	YES	30%	15%	55%	6.439e-1
500	YES	YES	30%	15%	55%	5.021e-1
1000	NO	NO	30%	15%	55%	9.114e-1
1000	YES	NO	30%	15%	55%	5.11e-1
1000	NO	YES	30%	15%	55%	7.872e-1
1000	YES	YES	30%	15%	55%	5.082e-1

Table 8. Failure Rate using NARX with 40% test data and 60% validation and train data

Runs	Embedded Dimension	Environment Conditions	Test Data Percentage	Validation Data Percentage	Train Data Percentage	Failure Rate	Average Failure
100	NO	NO	40%	15%	45%	13.309e-1	8.61E-01
100	YES	NO	40%	15%	45%	7.006e-1	
100	NO	YES	40%	15%	45%	8.12e-1	
100	YES	YES	40%	15%	45%	6.006e-1	
500	NO	NO	40%	15%	45%	14.589e-1	7.99E-01
500	YES	NO	40%	15%	45%	5.043e-1	
500	NO	YES	40%	15%	45%	8.229e-1	
500	YES	YES	40%	15%	45%	4.1e-1	
1000	NO	NO	40%	15%	45%	12.984e-1	4.90E-01
1000	YES	NO	40%	15%	45%	9.034e-2	
1000	NO	YES	40%	15%	45%	5.002e-1	
1000	YES	YES	40%	15%	45%	7.001e-2	

The 1st, 5th and 9th rows from every NARX table show the results of prediction based on our previous work. The failure rate of the rows which consider the extensions is much more accurate. Therefore we can say, importing the new extensions in this work, that is, embedded dimension as the delay parameter and considering each thread behavior in predicting other threads future behavior, made considerable improvement in prediction results particularly when the number of runs increases. We also showed the prediction results of NARX networks was much more accurate than the results obtained by Markov Chain, which is a statistical approach. As we stated, our test target program behaves randomly at runtime. Therefore, it was not possible to suppose an accurate model for Markov Chain prediction strategy. This is why that the failure rate of this strategy, as shown in Tables 1 to 4, are imprecise in comparison with the similar tables of the NARX prediction.

The average results of every NARX table (Figure 9) show a comparative view of the results of this strategy. Every line marked with a (X,Y,Z) statement, which X means the test set percentage, Y means validation set percentage and Z means the training set percentage. When the training set percentage is significantly lower than twofold test set percentage, the failure rate will increase. Also as the number of runs increases the effect of training is much more visible. According to the chart, the best overall result is in the case of (20, 15 and 65). This result is dedicated for our target multithread program and it may differ for other multithread programs.

In [4], after training networks we ran target program 500 times and tried to predict the deadlock possibility during these runs. During these runs deadlock occurred 17 times. Our approach reported 13 before their occurrences and missed 4. Also in 3 cases, it reported false positive, thus the precision was about 74%. In this work, after training the networks using considered extensions, we again ran test multithread program 500 times to see how many deadlocks will be reported correctly. It results 15 deadlocks during 500 times. Our model, this time, reported 14 and missed just one deadlock not reported; also it didn't report any false positive. This time, the precision was about 88%. In

comparison with our previous work [4], the extensions made a clear improvement in the results up to 15%.

5. Conclusion

Online potential deadlock detection techniques received lots of attention in recent years. But these techniques often are not cost efficient, neither in time or space. Also they need extra programmer effort to instrument the code and in some cases the results of these techniques may come too late. Considering these problems we proposed a novel online model to predict the deadlock at runtime in multithread programs rather than discovering deadlocks by pre-running some execution traces to find the potential deadlocks. In our proposed model the main runtime overhead is through the predictor component which predicts the future behavior of threads using neural network. In this work we used the "Nonlinear Autoregressive with external input (NARX)" network. The learning phase of NARX network has the order of complexity $O(n^3)$ in worst case [22]. But this complexity is related to offline phase of our proposed model

and once the networks were trained, then at runtime the output of predictor will be generated with a lower order of complexity, therefore our model doesn't force considerable overhead at runtime. Also our model could be execute on a completely different core from the main program and because of the simplicity of instrumentation logic it doesn't interfere in the target program execution.

In this work we extended our previous work in two ways:

1. Using time series analysis approaches in configuring predictor network parameter
2. Using NARX network instead of NAR network.

The obtained results showed that the extensions described in this paper, made improvement in the prediction of potential deadlocks. The configuring a predictor neural network considering the problem specification and requirements resulted the more precise predictions. Because of this experience, in our future work, we are planning to configure the predictor networks parameters based on the static analysis and structure of the target multithread program, we hope to obtain more accurate results.
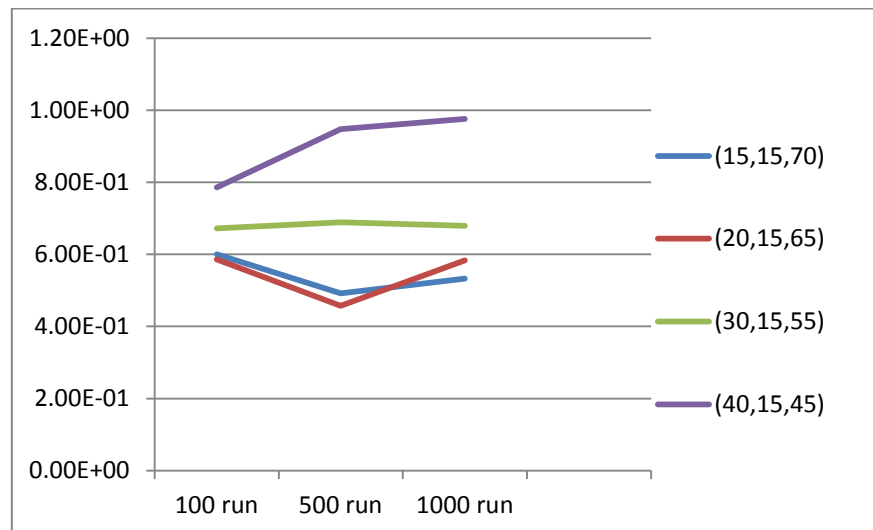

Fig. 9. Average results of NARX strategy with different test, validation and train data

References

[1] D. Engler, K. Ashcraft," RacerX: effective, static detection of race conditions and deadlocks," ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 237–252, 2003.

[2] P. Joshi, C. S. Park, K. Sen, and M. Naik, " A randomized dynamic program analysis technique for detecting real deadlocks," in ACM Sigplan Notices, 2009, vol. 44, pp. 110–120.

[3] H. Jula, D. Tralamazza, C. Zamfir, G. Candea, " Deadlock immunity: Enabling systems to defend against deadlocks," in Proceedings of the 8th USENIX conference on Operating systems design and implementation, 2008, pp. 295–308.

[4] E. Hasanzade , S. M. Babamir, "An Artificial Neural Network Based Model for Online Prediction of Potential Deadlock in Multithread

Programs," in 16th Symposium of Artificial Intelligence and Signal Processing, AISP 2012, IEEE Society, pp. 417-422, 2012.

[5] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In Proc. 7th Int'l. SPIN Workshop on Model Checking of Software,volume 1885 of Lecture Notes in Computer Science, pages 245-264. Springer-Verlag, August 2000.

[6] R. Agarwal, L. Wang, S. Stoller, " Detecting potential deadlocks with static analysis and run-time monitoring," Hardware and Software, Verification and Testing, pp. 191–207, 2006.

[7] S. Bensalem, K. Havelund, "Scalable deadlock analysis of multi-threaded programs," in Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference. Springer-Verlag, 2005.

[8] Y. Nir-Buchbinder, R. Tzoref, S. Ur, "Deadlocks: From exhibiting to healing," in Runtime Verification, 2008, pp. 104–118.

[9] O. Voitcu, Y. Wong, "On the construction of a nonlinear recursive predictor," Science B.V., Journal of Computational and Applied Mathematics, 2004.

[10] N. Baccour, H. Kaaniche, M. Chtourou, M. B. Jemaa, " Recurrent neural network based time series prediction: Particular design problems," studies, vol. 1, p. 7.

[11] Y. Chen B. Yang J. Dong A. Abraham, " Time-series forecasting using flexible neural tree model," Science, Information Sciences pp 219–235, 2004.

[12] C.J. Lin, Y.J. Xu, "A self-adaptive neural fuzzy network with group-based symbiotic evolution and its prediction applications," Science, Fuzzy Sets and Systems, 2 September 2005.

[13] R. Zemouri, P. Ciprian Patic "Recurrent Radial Basis Function Network for Failure Time Series Prediction," World Academy of Science, Engineering and Technology 72, 2010.

[14] R. Zemouri, D. Racoceanu, N. Zerhouni, "Recurrent radial basis function network for time-series prediction," Engineering Applications of Artificial Intelligence pp. 453–463, 2003.

[15] M. Khashei, M. Bijari, " An artificial neural network (p,d,q) model for time series forecasting" Journal of Expert Systems with Applications, pp. 479-489, 2010.

[16] R. Rojas, "Neural networks: a systematic introduction," Springer. pp. 336, 1996.

[17] H. Hyotyniemi, " Turing Machines are Recurrent Neural Networks," Proceedings of STeP'96. Jarmo Alander, Timo Honkela and Matti Jakobsson, pp. 13-24, 1996.

[18] G. Dorffner, "Neural Networks for Time Series Processing," Neural Network World, Vol. 6, No. 4, 447-468, 1996

[19] E. Dodonov, R. F. de Mello, "A novel approach for distributed application scheduling based on prediction of communication events," Future Generation Computer Systems, vol. 26, no. 5, pp. 740–752, 2010.

[20] M.B. Kennel, R. Brown, H.D.I. Abarbanel, "Determining embedding dimension for phase-space reconstruction using a geometrical construction", Physical Review A 45 (6) (1992) 34033411.

[21] T.Lin, B.G. Horne, P.Tino, C. Lee Giles, " Learning long-term dependencies in NARX recurrent neural networks," IEEE Transactions on Neural Networks, Vol. 7, No. 6, 1996, pp. 1329-1351

[22] G. Ferrari, G. De Nicolao, "NARX models: optimal parametric approximation of nonparametric estimators," in American Control Conference, 2001. Proceedings of the 2001, 2001, vol. 6, pp. 4868–4873.