Concurrency Bug Detection and Avoidance Through Continuous Learning of Invariants Using Neural Networks in Hardware

Mejbah Ul Alam, Rehana Begam, Sabidur Rahman, Abdullah Muzahid
{malam, rbegam, krahman, muzahid}@cs.utsa.edu
University of Texas at San Antonio

ABSTRACT

As parallel architectures become mainstream, the issue of writing and debugging parallel programs becomes a major concern for today's computing world. Researchers have shown that different types of concurrency bugs can be effectively detected by checking the validity of various invariants (e.g. inter-thread communication invariants). Existing invariant based proposals collect a large number of execution traces by using various test inputs available from the developers. Then, they analyze those traces and extract the required invariants. As a result, these proposals suffer from a number of limitations arising from this invariant extraction process. First, they cannot test all possible thread interleavings and hence, miss many invariants. Second, often times these proposals cannot replicate the production run environments and inputs during the extraction process. This results in missing and sometimes incorrect invariants. Third, as a program is extended and fixed over its lifetime, many invariants might become invalid and new invariants might arise. These proposals cannot cope up with this situation properly. Finally, existing invariant based proposals cannot check the correctness of an invariant if it is not found during the extraction process. Therefore, these proposals allow (or disallow) any new invariant blindly. These limitations severely hampers the applicability of any invariant based approach. In order to remedy this situation, we propose a novel scheme that uses hardware based neural networks to continuously learn and verify invariants dynamically during production runs. This alleviates all of the above mentioned limitations, thereby making any invariant based approach more applicable and effective. In this paper, we present our initial design and some preliminary results of our proposed scheme.

1. INTRODUCTION

As parallel architectures become mainstream, programmers get burdened with the responsibility to write correct parallel programs. However, writing a correct parallel program is not easy. In fact, various concurrency bugs make it a daunting task. One of the reasons why concurrency bugs become so notorious, is their highly non-deterministic nature. They might manifest in the most unfortunate moment and cause catastrophes. Therefore, it is imperative to keep innovating novel techniques to *detect* as well as *avoid* concurrency bugs.

A concurrency bug can be informally defined as a bug related to the interactions among different threads of a parallel program. There are many categories of concurrency bugs. The main categories found in literature are data races, deadlocks, atomicity violations, ordering violations, and sequential consistency violations. There has been significant research effort to debug these bugs. Data races are the most commonly studied [18, 2, 16, 1] concurrency bugs. Researchers have started to focus on other concurrency bugs too. Proposals like AVIO [10], AtomTracker [14], MUVI [9], LifeTx [22], etc. detect atomicity violation bugs where a group of accesses that are supposed to be processed atomically get interleaved with conflicting accesses from the other threads. Recently, works like Vulcan [15], DRFx [13], Gharachorloo et al. [5], etc. focus on detecting sequential consistency violations where memory operations get reordered unintuitively. Most of these proposals share a common shortcoming — they target only one type of bugs. To remedy this, researchers have proposed schemes like PSet [21], Bugaboo [11], DefUse [19] etc. that do not rely on the symptoms of any particular bug. Instead, these proposals focus on identifying correct data communications among threads and provide a general solution to handle any type of concurrency bugs.

In general, we can divide the research in concurrency bug detection into two broad categories — invariant based and symptom based. Proposals like AVIO, AtomTracker, LifeTx, MUVI, PSet, Bugaboo, DefUse, etc. can be considered as invariant based approaches. All of them extract some invariants from the program. On the contrary, other proposals [18, 2, 16, 15, 13, 5] focus on identifying various symptoms of concurrency bugs. As an example, Eraser [18] detects data races by determining whether a shared variable is consistently protected by at least one lock, whereas AVIO detects data races and atomicity violations by first inferring which pair of instructions accessing the same variable in the same thread should not be interleaved by conflicting accesses from a remote thread and then checking whether this invariant is maintained at runtime. Invariant based approaches can extract invariants that closely match with the programmer's actual intention in the code. Therefore, these approaches often detect more than one type of concurrency bugs and have a higher detection ability than symptom based approaches.

The invariant based proposals usually work in two phases — invariant extraction phase and invariant verification phase. During the extraction phase, these proposals collect a large number of execution traces of a program. This is usually done by applying different test inputs to the program. The proposals analyze these traces to find out potential invariants. Once we reach a situation where no more new in-

variants are found by analyzing more traces, the extraction phase is assumed to be over. The whole process is done *offline*. The invariants found this way are stored along with the program binary. During the verification phase, as the program executes, these approaches dynamically check whether an invariant is violated or not. If an invariant is violated, then a concurrency bug is reported.

All the existing invariant based proposals suffer from several limitations arising from the current invariant extraction process. *First*, since the number of possible thread interleavings increases exponentially with the number of instructions, number of variables, and number of threads, it is quite difficult to test all possible interleavings with all possible inputs. Therefore, existing proposals miss many valid invariants during the offline extraction phase. *Second*, the production environment can be substantially different from the extraction environment. With the arrival of data centers and many different types of parallel architectures, it is very difficult to replicate a scaled down version of different production environments during the extraction process. This implies that the existing proposals are not able to collect execution traces that would arise during the production environments. On top of this, the inputs may also vary among different environments. All of these factors can lead to under training. *Third*, as a program evolves, it is continuously modified and extended with new features and fixes. This can invalidate some of the prior invariants and create new invariants. The current approaches cannot cope up with this continuous modification process. *Finally*, during the invariant verification phase, if a new invariant arises, the existing proposals cannot decide its correctness. Therefore, they apply some default strategy (e.g allow any new invariant or disallow them all). This limitation, coupled with the fact that the existing proposals cannot uncover all invariants, severely limits their practical applicability.

In order to remedy this situation, we propose to have *continuous learning and verification*. This is made possible due to a novel approach based on a neural network (a type of machine learning algorithm) implemented in hardware. The high level idea is that as a program starts execution, the neural network initially spends some time learning the invariants. We choose Read-After-Write (RAW) dependences among a write and the following read operations as our preferred invariant. However, we can easily extend this approach to other invariants. After the neural network learns the RAW dependences reasonably well, it enters into the verification phase where it verifies whether a RAW dependence is correct or not. If a RAW dependence is declared as an incorrect one, then the processor re-executes the relevant read operation and attempts to avoid the concurrency bug. This is likely to avoid most of the concurrency bugs. A log of recent dependences is also kept along. In case, the bug cannot be avoided and the program behaves incorrectly, this log provides valuable debugging information to the programmer. During the verification phase, the neural network keeps track of its accuracy. If it drops below a certain threshold, the network starts to learn again for a period of time. Thus, the neural network goes through a continuous learning and verification phase. The main contributions of our proposed scheme are as follows.

- This is the *first* proposal to use a neural network to detect and in many cases avoid concurrency bugs that occur during a program execution.
- This is the *first* invariant based proposal where learning and verification process is done continuously. As a result, the proposed scheme avoids the previously mentioned limitations of the existing invariant based proposals.
- Our proposed scheme avoids most of the concurrency bugs without expensive checkpointing and rollback support.

The rest of the paper is organized as follows: Section 2 discusses some related work and background materials, Section 3 describes the main idea of our proposed scheme, Section 4 shows some preliminary results and finally, Section 5 concludes our work.

2. BACKGROUND AND RELATED WORK

AVIO [10] is one of the earliest works on invariant based concurrency bug detection. AVIO detects single variable atomicity violation bug. Later, AtomTracker [14] and LifeTx [22] generalize this work by incorporating more instructions and more variables. In order to handle different types of concurrency bugs uniformly, PSet [21] proposes to use an invariant based on inter thread communication. It is called Predecessor Set. Predecessor set of a memory access instruction includes all the other remote memory access instructions upon which this one immediately depends. It considers all three types of dependences (i.e. read-after-write, write-after-write, and write-after-read). The proposal then stores the predecessor set information along with the binary. At run time, if a memory access instruction is found to depend on an instruction other than the predecessor set instructions, then PSet finds a concurrency bug. In that case, it either stalls or rolls back the offending thread to avoid the concurrency bug. However, it shares all the limitations of an offline invariant extract process. Bugaboo [11] extends PSet by incorporating a limited form of context information with the communication invariants. DefUse [19] uses slightly different invariants where it finds out all the definitions upon which a read depends. We propose to use both inter thread and intra thread RAW dependences as our preferred invariant. This invariant is similar to the invariants used in PSet, Bugaboo, and DefUse proposals. We choose this invariant to show that our proposed scheme can be applied in conjunction with *any of the existing invariant based approaches*.

A neural network is a machine learning algorithm to learn a target function. It consists of a number of artificial neurons connected by links. Figure 1(a) shows an artificial neuron i with inputs a_0 to a_n. W_0, to W_n are the weights of the links. The neuron calculates its output as $o = g(\sum_{j=0}^{n} W_j a_j)$ where g is an activation function. g can be a simple threshold function or a more complex sigmoid function [17]. The output of one neuron can act as an input to another neuron like in Figure 1(b). Here, we have a neural network with two inputs, one hidden layer of two neurons and one output neuron. Each hidden layer neuron first calculates its output and then the output neuron provides the final output. The learning process of a neural network consists of adjusting the weight of each link to approximate the function that it tries to learn. After the learning process is over, neural network simply calculates the final output based on the inputs and weights.

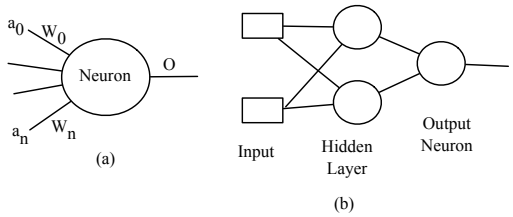

Figure 1: Artificial neural network

There are many proposals that use some machine learning algorithm in hardware to accomplish different tasks. One of the earliest effort is Perceptron Branch Predictor [8] that uses a neural network to predict the branch outcome. Ipek et. al [7] uses some reinforcement learning algorithm to design an optimizing memory controller. Recently, Esmaeilzadeh et. al [4] uses a multilayer neural network in hardware to approximate the computations of some code regions.

3. MAIN IDEA

The high level idea of our scheme is to design a framework that uses a neural network to learn and verify the correctness of an invariant a program runs. We choose RAW dependences as our preferred invariant to show the generality of the framework.

3.1 Using RAW Dependences as Invariants

For each memory read instruction, there is usually a fixed set of memory write instructions upon which the read depends. A concurrency bug occurs when a memory read instruction gets the value from a memory write instruction outside the set of valid writers. This intuition leads us to choose the set of correct Read-After-Write(RAW) dependences as our invariant. We consider RAW dependences between local as well as remote instructions.

If a program runs correctly, all the RAW dependences that occur during the execution are assumed to be correct. On the other hand, if a program behaves incorrectly because of a concurrency bug, we are likely to observe at least one RAW dependence that we have not seen during the correct executions. This can be explained with the example shown in Figure 2(a)-(c). Here, p is a pointer allocated and freed by thread T1. T2 is another thread that uses p if it is not NULL. Note that none of the threads uses any synchronization. Hence, the program has data races.

Figure 2(a) & (b) show two scenarios that produce correct results. Therefore, the correct RAW dependences are $I1 \rightarrow J1$, $I1 \rightarrow J2$, and $I2 \rightarrow J1$. The interleaving in Figure 2(c) causes a crash of the program. The RAW dependences, in this case, are $I1 \rightarrow J1$ and $I2 \rightarrow J2$. Although the first one is a correct one, the second one does not match with any correct dependences found before. Hence, $I2 \rightarrow J2$ is an incorrect RAW dependence that leads to this bug.

It should be noted that whether a RAW dependence is correct or not, may depend on the state of the system. This can be explained with the help of the Figure 2(d)-(f). Here, a point object p has x and y members to denote x and y co-ordinate. So, intuitively, we can assume that both of them should be accessed together atomically. However, the programmer mistakenly accesses them in different critical sections. Figure 2(d) & (e) show the scenarios where atomicity is not violated and hence, correct results are produced. So, the correct RAW dependences are $I1 \rightarrow J1$, $I2 \rightarrow J2$, $K1 \rightarrow J1$, and $K2 \rightarrow J2$. In Figure 2(f), thread T1 ends up using updated value of x from T2 and y from T1. As a result, atomicity is violated and the program becomes incorrect. Here, both of the RAW dependences (i.e. $K1 \rightarrow J1$ and $I2 \rightarrow J2$) match with the correct RAW dependences found before. Still, this is a buggy execution. It should be noted that thread T1, at the point of instruction J1 and J2, should consume both updated values either from T1 or T2 entirely. This implies that when the T1 is at J2, the state of the system is such that $I2 \rightarrow J2$ is not the correct RAW dependence anymore. During that state, $K2 \rightarrow J2$ is the correct dependence. For this reason, we should consider the state of the system to determine the correctness of a RAW dependence. The state usually constitutes of some attributes of the system. Some example attributes can be last N_i RAW dependences of the local processor, last N_j RAW dependences of the remote processors, last N_k shared memory accesses, last N_l function calls, last N_m synchronization variables used etc. We choose last N_i RAW dependences of the local processor to represent the current state.

3.2 Continuous Learning and Verification of RAW Invariants

We envision an architecture where each processor has a Learning Module (LM). The LM contains a neural network. We choose neural network algorithm because of its wide applicability. More specifically, we use multilayer perceptron algorithm.

When a program starts execution, the LM of every processor is provided with the RAW dependences related to that processor Initially, the LM spends some time learning the dependences. We optimistically assume that all the RAW dependences during the learning phase are correct. If, however, some of them are not correct and cause incorrect program behavior (e.g. program crash), we fail to detect the bug and start over the learning process from scratch during the next execution. However, we can still provide the log of a number of recent dependences to the programmer to detect the bug. During the learning process, we also provide negative examples to the neural network. Negative examples are formed by looking at the writes (e.g. the write before the last write) upon which the read does not depend. Once the learning phase is over (as indicated by the neural network algorithm), the LM starts verifying the RAW dependences. One of the advantages of using neural network is that even if a dependence never appears during the learning phase, the network can declare whether it is likely to be correct or not. The processor allows the dependences that are declared to be correct. If a RAW dependence is declared to be incorrect, then the processor re-executes the memory read operation to avoid it. This prevents the concurrency bugs most of the time. However, if the incorrect RAW dependence cannot be avoided this way, the bug eventually happens and causes the program to behave incorrectly. When this happens, the LM provides a log of recent dependences to the programmer to pin point the bug

3.2.1 Detailed Architecture

There are many possible implementations [23, 4, 3, 20, 8] of neural networks in hardware. We focus on an ASIC design.

At the high level, each processor has an LM connected

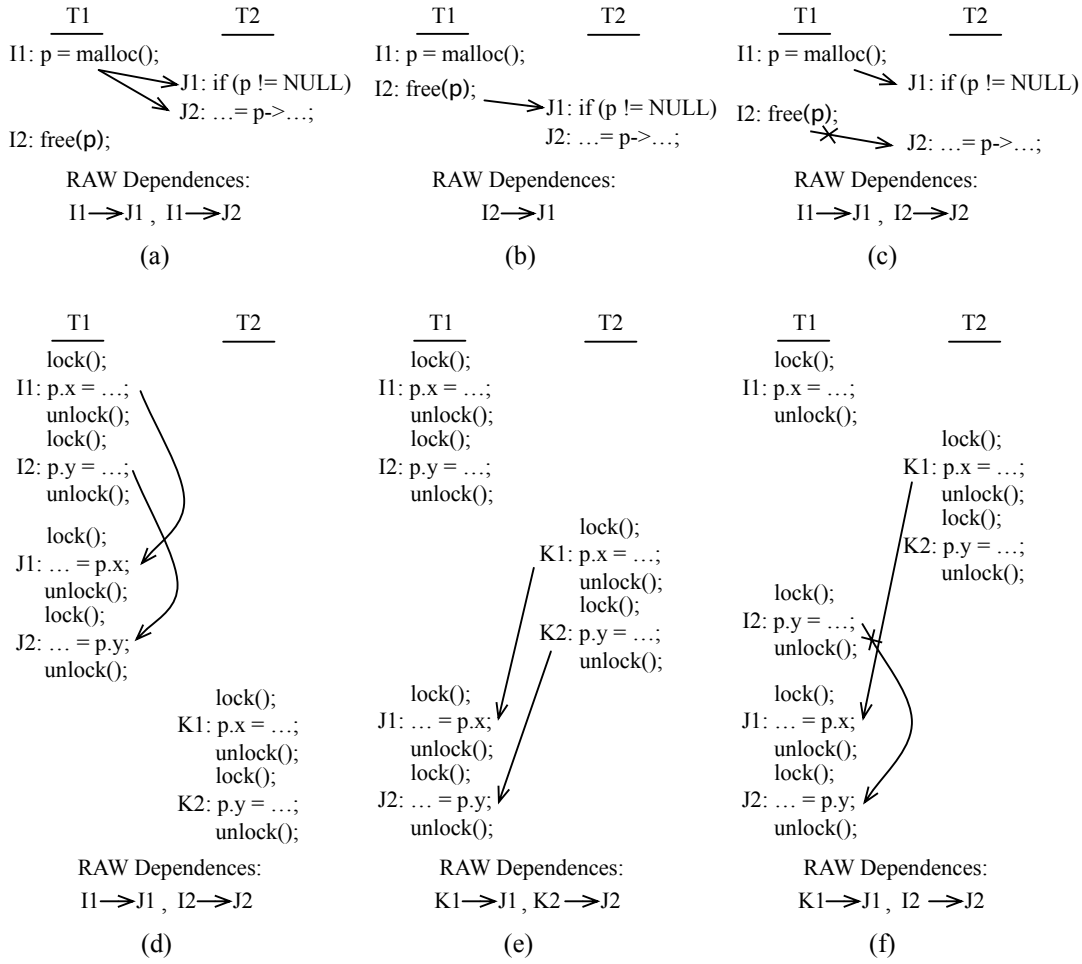

Figure 2: How RAW dependences can be used as invariants

with load buffer, reorder buffer, and cache controller. This is shown in Figure 3(a) and (d). In addition to this, we need to augment each cache line to store the instruction address of the last write operation on different words of the line. This is needed to form the RAW dependences. This information is piggybacked with cache coherence messages.

The design of the LM is shown in Figure 3(b). Whenever the processor issues a read operation, the LM collects the memory address and the corresponding instruction address from load buffer and reorder buffer. When the relevant cache line is accessed, the cache controller returns the last writer instruction address of the requested word. This allows the LM to form the RAW dependence which is then provided to the neural network.

We assume a highly pipelined design for the neural network. Therefore, we need an Input FIFO and an Output FIFO inside the LM. In addition to these, we have a Scaling Register and a Configuration Register as shown in Figure 3(b). The scaling register contains any scaling factor necessary for the input and the output. The configuration register contains the initial weights for the links of the neural network.

Figure 3(c) shows the inside of an individual artificial neuron. It consists of a Weight Register, an Input Buffer, an Output Buffer, a Sigmoid unit, a Multiplication-Addition unit, and a Controller. Each neuron takes its input from the Input FIFO of the LM or the Output Buffer of another neuron. It then copies the input to its own Input Buffer. After the output is produced, it is placed in the Output Buffer.

3.2.2 How It Works?

The LM has two modes of operation — learning mode and verification mode. It should be noted that both of them are done as a program runs. When a program starts execution, the LM of each processor starts working in the learning mode. During this mode, the LM trains its neural network with the RAW dependences. During this mode, the LM treats every RAW dependence as a correct one. The LM trains its neural network with these dependences. The LM needs to calculate its accuracy in order to determine when the learning process is over. Every time, a RAW dependence is provided to the neural network, it first calculates the output. The output either declares the newly found RAW dependence as a correct one or an incorrect one. If the neural network declares the dependence as a correct one, then it is right. Otherwise the declaration is wrong. When the accuracy, calculated this way, rises above a certain threshold, the learning process is assumed to be over.

Once the learning phase is complete, the LM enters into verification mode. Here, every time the processor executes a memory read operation, the RAW dependence is formed

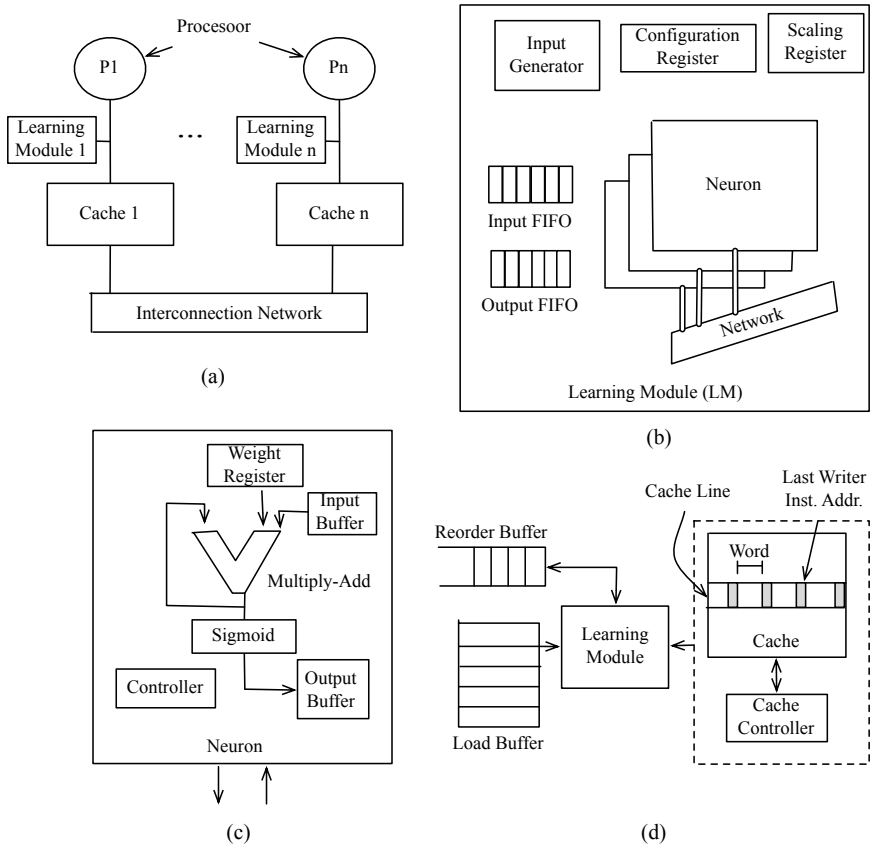

Figure 3: Architecture of the Learning Module.

and sent to the LM. This is done before the read operation commits from the ROB. The LM, based on its training, categorizes the dependence as a correct one or an incorrect one. If the dependence is categorized as an incorrect one, the LM signals the pipeline to flush the read operation and re-execute it. When the read operation is executed again, chances are the processor will read from another writer instruction, thereby avoiding the incorrect RAW dependence. There can be several re-execution attempts to avoid the incorrect RAW dependence. Most of the concurrency bugs can be prevented this way. Even after several re-execution attempts, if the incorrect RAW dependence still occurs, the LM allows the dependence to happen. This is to ensure forward progress of the program. The LM logs a number of recent dependences in some memory mapped file. In case, a crash occurs later because of one of the incorrect dependences, this log provides valuable debugging information to the programmer. Whenever the network declares a dependence as an incorrect one and lets it happen and the program does not crash within the next few hundred instructions, it is considered as a wrong declaration. The neural network calculates its accuracy in this way. If the accuracy drops below some threshold, then the network starts its learning process all over again. Thus, the neural network goes through a continuous learning and verification process during a program run.

It should be noted that the neural network can have both false positives (labeling a correct RAW dependence as an incorrect one) and false negatives (vice versa). In case of a false positive, there can be several re-execution attempts to avoid the RAW dependence. Eventually, the dependence is allowed to happen. This can at most slows down the program a little bit. On the other hand, in case of a false negative, the incorrect RAW dependence happens and the program eventually starts to behave incorrectly. Then, the log of dependences provides debugging information to the programmer to pin point the bug.

This paper presents the initial design of our idea. Therefore, we ignore cache eviction, effect of cache modification, and other implementation issues. These issues will be considered in a more advanced version of the design.

4. PRELIMINARY RESULTS

Before implementing the whole system in an architectural simulator, we did some small scale experiments to show the feasibility of neural network usage. We used WEKA [6], an open source machine learning toolset to implement a multi layer perceptron algorithm. The parameters are shown in Table 1.

Input	Last 5 RAW dependences
Hidden Layer	1
Number of Nodes	Input layer 10, Hidden layer 7, Output layer 1
Activation Function	Sigmoid
Learning Rate	0.3
Number of Epoch	500
Momentum	0.2

Table 1: Parameters of the neural network.

We used FFT program from SPLASH2 with default parameters for 4 processors. We used a PIN [12] based tool to collect traces of 10 executions for WEKA. We split all the data into training data and testing data. After the neural network was trained with our training data, we used it to determine the correctness of the RAW dependences of the testing data. Table 2 shows the obtained results.

# training data	accuracy (%)	false pos. (%)	false neg. (%)
1000	66.12	11.69	22.19
5000	66.25	2.66	31.09
10000	99.67	0.12	0.21
15000	92.07	0.09	7.84
25000	99.74	0.18	0.08
35000	99.77	0.14	0.09
50000	99.78	0.12	0.09
60000	99.82	0.03	0.15
75000	99.78	0.12	0.10
90000	99.76	0.18	0.06
100000	99.81	0.03	0.16

Table 2: Results from FFT.

The neural network quickly obtains 99% accuracy. There is a slight decrease of accuracy from 10000 to 15000 training data points. This is due to inclusion of data from more than one thread. This is an early report of this ongoing project. Through experiments using more benchmarks and full system simulator to show on-the-fly learning and verification of invariants will be included in the advanced version of this paper.

5. CONCLUSION

As parallel architectures become mainstream, programmers get burdened to write correct parallel programs. This paper proposes a novel approach for concurrency bug detection and avoidance using neural networks in hardware. This is the *first* work that proposes to learn and verify RAW dependences continuously during program executions. The proposed scheme has a unique advantage over existing invariant based approaches — it can handle new invariants resulting from new environments and inputs. If the neural network declares a RAW dependence to be incorrect, then the corresponding load instruction is re-executed after a while. The proposed scheme avoids most of the concurrency bugs in this way. However, if the bug does happen, then a log of recent dependences provides valuable information to the programmer to pin point the concurrency bug. We present our initial design of the idea and perform some small scale feasibility study. Our preliminary results show that the neural network can quickly train itself to achieve 99% accuracy in categorizing the correctness of any RAW dependences.

6. REFERENCES

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability*, December 2003.

[2] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, June 2002.

[3] H. Esmaeilzadeh, P. Saeedi, B. Araabi, C. Lucas, and S. Fakhraie. Neural Network Stream Processing Core (NnSP) for Embedded Systems. In *ISCS*, May 2006.

[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs. In *MICRO*, December 2012.

[5] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *SPAA*, July 1991.

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1), 2009.

[7] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, June 2008.

[8] D. A. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *ISCA*, January 2001.

[9] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *SOSP*, October 2007.

[10] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, October 2006.

[11] B. Lucia and L. Ceze. Finding Concurrency Bugs with Context-aware Communication Graphs. In *MICRO*, December 2009.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, June 2005.

[13] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, June 2010.

[14] A. Muzahid, N. Otsuki, and J. Torrellas. AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection. In *MICRO*, December 2010.

[15] A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware Support for Detecting Sequential Consistency Violations in Programs Dynamically. In *MICRO*, December 2012.

[16] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. Sigrace: signature-based data race detection. In *ISCA*, 2009.

[17] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach*. Prentice Hall, 2003.

[18] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 1997.

[19] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition? DeFuse: definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, October 2010.

[20] K. Wojtek Przytula. Parallel Digital Implementations of Neural Networks. In *ASAP*, September 1991.

[21] J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared Memory Multi-processor. In *ISCA*, June 2009.

[22] J. Yu and S. Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *MICRO*, December 2010.

[23] J. Zhu and P. Sutton. Fpga implementations of neural networks - a survey of a decade of progress. In *FPLA*, September 2003.