Production-Run Software Failure Diagnosis via Adaptive Communication Tracking

Mohammad Mejbah ul Alam, Abdullah Muzahid
University of Texas at San Antonio
{*mohammad.alam, abdullah.muzahid*}*@utsa.edu*

***Abstract*—Software failure diagnosis techniques work either by sampling some events at production-run time or by using some bug detection algorithms. Some of the techniques require the failure to be reproduced multiple times. The ones that do not require such, are not adaptive enough when the execution platform, environment or code changes. We propose ACT, a diagnosis technique for production-run failures, that uses the machine intelligence of neural hardware. ACT learns some invariants (e.g., data communication invariants) on-the-fly using the neural hardware and records any potential violation of them. Since ACT can learn invariants on-the-fly, it can adapt to any change in execution setting or code. Since it records only the potentially violated invariants, the postprocessing phase can pinpoint the root cause fairly accurately without requiring to observe the failure again. ACT works seamlessly for many sequential and concurrency bugs. The paper provides a detailed design and implementation of ACT in a typical multiprocessor system. It uses a three stage pipeline for partially configurable one hidden layer neural networks. We have evaluated ACT on a variety of programs from popular benchmarks as well as open source programs. ACT diagnoses failures caused by 16 bugs from these programs with accurate ranking. Compared to existing learning and sampling based approaches, ACT has better diagnostic ability. For the default configuration, ACT has an average execution overhead of 8.2%.**

***Keywords*-Concurrency bugs; Sequential bugs; Failures; Dependence; Neural hardware;**

I. Introduction

A. Motivation

Software bugs are a disappointing aspect of programming. A recent study [1] has reported that programmers spend 50% of their effort in finding and fixing bugs. This costs $316 billion a year throughout the world. The situation is worsened by the recent flourish of diverse platforms (e.g., multicores, many cores, data centers, accelerators etc.). Many sequential as well as concurrency bugs escape into production systems causing failures. Diagnosing these failures can take weeks or even months [2], [3]. Besides the challenge of debugging, our community is currently facing stringent technological challenges, especially, for energy and faults. Researchers are forced to investigate alternative designs [4], [5], [6]. The next frontier in computer architecture is likely to be heterogeneous systems with a mix of cores and accelerators. One type of accelerators that shows a lot of promise is Neural Network based accelerators [4], [6]. Esmaeilzadeh et al. [6], Belhadj et al. [4], and Jimenez et al. [7] have demonstrated how neural networks can be used to approximate computations in general purpose programs, implement signal processing applications, and predict branches respectively. Companies like IBM [8] and Qualcomm [9] have designed chips that use neural network as a processor building block. Given the trend, it is likely that we will have some form of neural hardware (as accelerators or co-processors) in near future. The ability of such hardware to process data quickly and accurately in a noisy environment makes it a perfect fit for production-run failure analysis. Therefore, we propose a feedback oriented approach that utilizes neural hardware to diagnose production-run software failures caused by many sequential and concurrency bugs.

B. Limitations of Existing Software Failure Diagnosis Approaches

Software failures can be diagnosed by collecting various events and then, analyzing them. As an example, Arulraj et al. [10] propose to collect events such as L1 cache accesses and branch outcomes. Similarly, Lucia et al. [11] collect thread communication events. The events are collected from successful and failure runs. The collected events are analyzed using statistical [10] or machine learning [11] techniques to pinpoint the root cause. Sampling is used effectively [10], [12], [13], [14] for low-overhead event collection during production-run. Instead of collecting all events, sampling techniques collect events only at certain points during an execution. To account for the missing events, multiple executions need to be sampled randomly at different points. In other words, the failure execution needs to be reproduced and sampled multiple times. Reproducing the failure run in testing environment can be quite challenging, especially for concurrency bugs due to their dependance on thread interleaving. Even for sequential bugs, reproducing the failure can be often difficult due to inputs, environments, platforms etc. Thus, there is a need for techniques that can diagnose failures at production site without requiring to reproduce them.

Alternatively, we can think of using dynamic bug detection techniques during production-run to diagnose software failures. Such approach would pinpoint the bug as soon as it occurs. Proposals such as Savage et al. [15] and Choi et al. [16] detect data races whereas AVIO [17], and AtomAid [18] detect atomicity violation bugs. Researchers have proposed schemes like PSet [19], Bugaboo [20], DefUse [21] etc. that focus on identifying correct data communications among threads and provide a general solution to detect any concurrency bug. Among the existing schemes, AVIO, PSet, Bugaboo, DefUse, DIDUCE [22] etc. can handle more than one type of bugs and hence, are suitable for production-run use. These schemes extract some form of program invariants (e.g., data communication, correlation, value range etc.) mainly by analyzing execution traces. This is referred to as *training* of these proposals. The invariants are stored along with the program. At production-run time, if any invariant is violated, the software failure can be attributed to the violation.

Software failure diagnosis using an invariant based technique has the advantage that it can pinpoint the root cause without reproducing the failure; however, invariants can change as we observe more executions with different platforms, environments or code modifications. Therefore, the invariants need to be retrained when such changes occur. On the other hand, event sampling based diagnosis is unaffected by the changes but requires the failure to be reproduced multiple times. The goal of this paper is to achieve the best of the both worlds by relying on the machine intelligence provided by neural hardware.

C. Our Approach

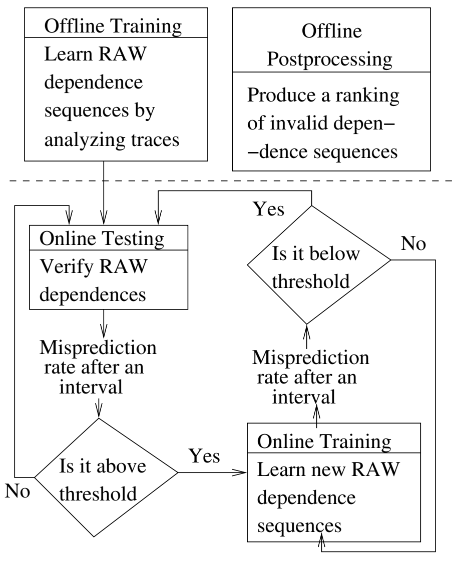

Offline Training
Learn RAW dependence sequences by analyzing traces
Offline Postprocessing
Produce a ranking of invalid depen−−dence sequences
Online Testing
Verify RAW dependences
Misprediction rate after an interval
Is it above threshold
Yes
No
Online Training
Learn new RAW dependence sequences
Misprediction rate after an interval
Is it below threshold
Yes
No

Figure 1. Overview of how ACT works.

We choose data communication invariant in our scheme due to its ability to detect concurrency, semantic as well as memory bugs [19], [20], [21] (Section II-B). We use Read After Write (RAW) dependences to represent data communications. Any other representation is equally applicable within our proposed approach. Figure 1 shows the high level idea of our scheme, called *ACT*. ACT first learns the sequence of RAW dependences by analyzing some execution traces offline. This is done to make ACT aware of the most common RAW dependence sequences. ACT uses both inter-thread and intra-thread RAW dependences. During an execution, as each RAW dependence occurs, ACT checks online using neural hardware whether the RAW dependence is likely to be valid. ACT does so by looking at the sequence of past few RAW dependences. Keep in mind that we are considering a futuristic system where each processor core is tightly integrated with the neural hardware. If the RAW dependence is predicted to be invalid, ACT logs the relevant instructions. Since neural hardware can occasionally predict a valid dependence as invalid, ACT periodically checks its misprediction rate. If the misprediction rate is above a certain threshold, it starts learning new RAW dependence sequences online until the misprediction rate drops below the threshold again. Thus, ACT goes through online testing and training phase alternatively during an execution. When a failure occurs, ACT postprocesses (offline) the log of invalid RAW dependences to produce a ranking of the root cause. Since ACT is able to learn new dependences online even after deployment, it can adapt to any change in platform, environment or code. Since ACT inspects every RAW dependence and logs only the ones that are likely to be invalid, its postprocessing analysis can pinpoint the root cause fairly accurately *without requiring to observe the failure again.*

Scheme	Suitable for production run?	Effective with a single failure run?	Can adapt to changes?
PBI [10], Aviso [12], CCI [23]	✓	✗	✓
Recon [11], [14]	✗	✗	✓
Avio, PSet, Bugaboo	✓	✓	✗
ACT	✓	✓	✓

Table I
COMPARISON WITH SOME EXISTING SCHEMES.

D. Contributions

1. We present the *first* neural hardware based approach, ACT, to diagnose production-run software failures. It can diagnose failures without requiring to reproduce them. ACT employs an adaptive feedback oriented online learning technique. ACT seamlessly handles many concurrency as well as sequential bugs. The paper presents a detailed design and implementation of ACT in a typical multiprocessor system. Table I shows a comparative analysis of ACT with respect to some existing schemes.

2. ACT uses a three stage pipeline design for a partially configurable one hidden layer neural network. We use the number of multiply-add units in a neuron as a knob to control the network's latency.

3. We simulated ACT architecture in a cycle accurate simulator [24]. We compared our proposed neural network implementation with an alternative implementation [6] to justify our design choice. We evaluated ACT on a variety of open source programs, SPLASH2, PARSEC, SPEC INT 2006, and GNU coreutil applications. We experimented with 11 real world and 5 injected bugs. ACT diagnosed failures due to all of the 16 bugs and pinpointed the buggy dependences with accurate ranking. ACT has an average execution overhead of 8.2% which makes it suitable for production-run deployment. We compared ACT with state-of-the-art learning based [12] and sampling based [10] approaches to show its effectiveness.

This paper is organized as follows: Section II provides some background; Section III explains the main idea of ACT; Section IV & V outline implementation and additional issues; Section VI provides the results; Section VII points out some future directions; Section VIII discusses related work and finally, Section IX concludes.

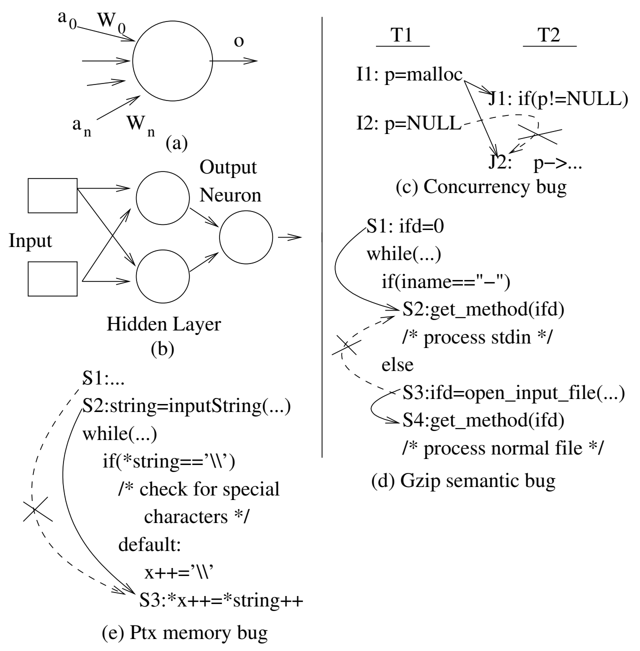
II. Background

A. Neural Network

A neural network is a machine learning algorithm to learn a target function. We use learning and training interchangeably throughout the paper. A neural network consists of a number of artificial neurons connected by links. Figure 2(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 [25]. The output of one neuron can act as an input to another neuron as in Figure 2(b). Here, we have a neural network with an input layer with two inputs, a hidden layer with two neurons and an output layer with one neuron. The input layer does not contain any actual neurons. It provides raw inputs to the hidden layer. Each hidden layer neuron calculates its output which then acts as an input to the output neuron. 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. Back propagation algorithm is the most widely used learning algorithm. If the expected output of the neuron in Figure 2(a) is t, then $err = o \times (1-o) \times (t-o)$ for sigmoid function or $err = (t-o)$ for threshold function. Then, the link weights are updated as $W_{j\_new} = W_j + err \times o$ for $0 \leq j \leq n$. After the weights are updated for a neuron, error is propagated back (as a proportion of the link weights) to the neurons of the previous layer. This process continues for each training input until an error threshold is reached.

B. Diagnosis via Communication Tracking

The use of data communication invariants to detect bugs has been popularized by works such as PSet [19], Bugaboo [20], DefUse [21] etc. ACT uses both inter-thread and intra-thread RAW dependences to capture the invariants. Let us consider the concurrency bug shown in Figure 2(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. In a correct execution, I2 should not interleave between J1 and J2. If we choose to represent a RAW dependence as $W \rightarrow R$ where instruction W wrote some data that instruction R reads, then the valid



```
I1: p=malloc
I2: p=NULL
                J1: if(p!=NULL)
                J2:   p->...
```

```
S1: ifd=0
while(...)
  if(iname=="−")
    S2:get_method(ifd)
    /* process stdin */
  else
    S3:ifd=open_input_file(...)
    S4:get_method(ifd)
    /* process normal file */
```

```
S1:...
S2:string=inputString(...)
while(...)
  if(*string=='\\')
    /* check for special
       characters */
  default:
    x++='\\'
  S3:*x++=*string++
```

Figure 2. (a), (b) show a neuron and a neural network respectively. (c) - (e) show how RAW dependences can be used.

RAW dependence sequences of Figure 2(c) are ($I1 \rightarrow J1$, $I1 \rightarrow J2$) and ($I2 \rightarrow J1$). Either one of them leads to a correct execution. If I2 interleaves between J1 and J2, the RAW dependence sequence is ($I1 \rightarrow J1$, $I2 \rightarrow J2$). $I1 \rightarrow J1$ is a valid RAW dependence that requires the next RAW dependence to be $I1 \rightarrow J2$. So, the sequence ($I1 \rightarrow J1$, $I2 \rightarrow J2$) is an invalid one that leads to a crash.

Figure 2(d) shows a semantic bug from Gzip. When "-" appears at the beginning of the inputs, the RAW dependence sequence is ($S1 \rightarrow S2$, $S3 \rightarrow S4$, $S3 \rightarrow S4$, ...) which makes *ifd* zero in S2. So, *stdin* gets processed inside *get_method*. Thus, the sequence leads to a correct execution and is valid. When "-" appears in the middle of the inputs, the dependence sequence is ($S3 \rightarrow S4$, $S3 \rightarrow S2$, $S3 \rightarrow S4$, ...) which makes *ifd* non-zero in S2. So, *stdin* does not get processed which is incorrect. $S3 \rightarrow S4$ is a valid dependence which requires the next dependence to be the same, not $S3 \rightarrow S2$. So, ($S3 \rightarrow S4$, $S3 \rightarrow S2$, $S3 \rightarrow S4$, ...) is not valid.

Figure 2(e) shows a buffer overflow bug from GNU coreutil *ptx*. S2 initializes *string*. If *string* contains an odd number of consecutive "\", S3 causes *string* to go out of bound. As long as *string* remains within bound, RAW dependence $S2 \rightarrow S3$ occurs. Thus, ($S2 \rightarrow S3$, $S2 \rightarrow S3$, ...) is a valid sequence. When *string* goes out of bound, S3 depends on some random instruction S1 that writes to the address next to *string*. Therefore, the sequence ($S2 \rightarrow S3$, $S1 \rightarrow S3$, ...) is an invalid one which causes the failure. Thus, many software failures due to concurrency, semantic, and memory bugs can be diagnosed by tracking RAW data communications.

C. Why Neural Networks Can Be Useful?

Neural networks are very effective in recognizing patterns. We can formulate bug diagnosis as a pattern recognition problem. Figure 3(a) shows a code section. The code accesses variables *a*, *b*, and *c* in sequence. The RAW data communications that occur with *a* at different load instructions are shown as A1 and A2. Thus, each of A1 and A2 represents a dependence similar to $I1 \rightarrow J1$ in Figure 2(c). Similarly, RAW data communications with *b* and *c* are shown as B1, B2, B3 and C1, C2, C3. Consider that (A1, B1, C1) is a data communication sequence at a particular instant. Similarly, (A2, B2, C2) and (A1, B3, C3) are other data communication sequences that occur at different instances. Let us assume that these three are the only sequences that are correct. If the first communication is A2, then we can predict that the next two communications will be B2 and C2. If the first one is A1, then we know that the next communication will be either B1 or B3. After seeing the next communication, we can predict whether the third communication is going to be C1 or C3. Thus, if we inspect the sequence of past communications, we can predict whether the communication that has just happened is valid or not. In other words, if a neural network is trained with data communication sequences, then it can fairly accurately validate the current communication based on the past few communications. Later, some postprocessing analysis of the invalid communications can accurately pinpoint the bug without requiring to reproduce it.

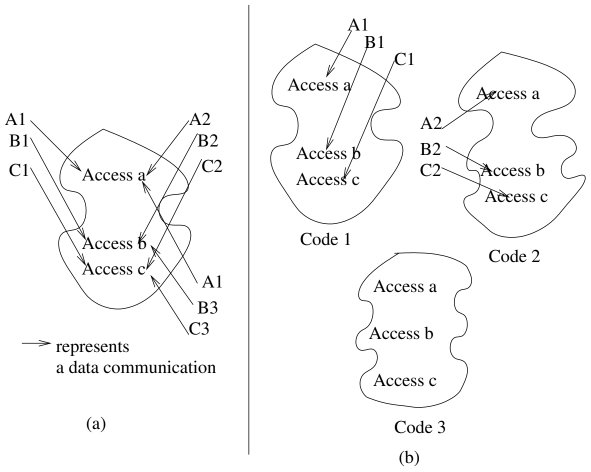


Figure 3. (a) shows various communication patterns. (b) shows the case for communication similarity.

Why would one choose neural networks? First, since a code section often accesses some of the same data that other code sections access, it can behave similar to those other sections. That is why, neural networks can predict the behavior of a completely new code section. This is illustrated in Figure 3(b) where $Code_1$ and $Code_2$ are code sections that access *a*, *b*, and *c*. (A1, B1, C1) and (A2, B2, C2) are communication sequences for these two sections. $Code_3$ is another code section that accesses the same variables. Hence, it is very likely that the communication sequence for $Code_3$ will be similar (completely or partially) to those sequences. So, if a neural network is trained with $Code_1$ and $Code_2$, it can predict how $Code_3$ is supposed to behave even though it never saw $Code_3$. This property is extremely useful because a program is continuously modified during its lifetime. For example, Apache has over 40 releases during the last 13 years. Existing invariant based approaches would require retraining of the whole program every time the program is modified. Neural networks, on the other hand, can seamlessly adopt such modifications by doing prediction based on similarity. Our experiments show that neural networks can correctly predict **94%** of the communication sequences of some newly added code (Section VI-D). Last but not the least, neural networks, when implemented in hardware, can learn communication sequences on-the-fly during production-run. Thus, the networks can adapt to new control and data flow as well as thread interleaving. Hence, neural network based diagnosis techniques can cope up with continuous change in code and platforms.

III. ACT Design

A. Overview

ACT learns and tests RAW dependence sequences continuously during the execution of a program. Initially, ACT analyzes some traces of the program to learn RAW dependence sequences offline. Note that for a neural network, learning essentially means determining the weights of the links. Before production-run deployment, the neural network is initialized with the weights just learned. As each RAW dependence occurs, ACT hardware tests the dependence sequence's validity online using a neural network. This mode of operation is referred to as *Online Testing* mode. The (potentially) invalid RAW dependence sequences (i.e., instruction addresses) are logged in a buffer. When the software fails (i.e., crash, incorrect output, corrupted report etc. occur), the log is further processed (offline) to produce a ranking of the buggy dependences to diagnose the failure. The offline processing does not require to observe the failure again. ACT periodically checks its misprediction rate during an execution. If it is too high, ACT enters into *Online Training* mode. In this mode, ACT starts learning RAW dependence sequences. When the misprediction rate drops below some threshold, ACT enters into online testing mode again. Thus, ACT operates in online testing and training mode alternatively.

B. Offline Training

ACT first learns the RAW dependence sequences of a program by analyzing some execution traces offline. Similar learning has been done in prior approaches [19], [20], [21]. Only the traces from correct executions are used. These traces can be collected by running test suites of a program.

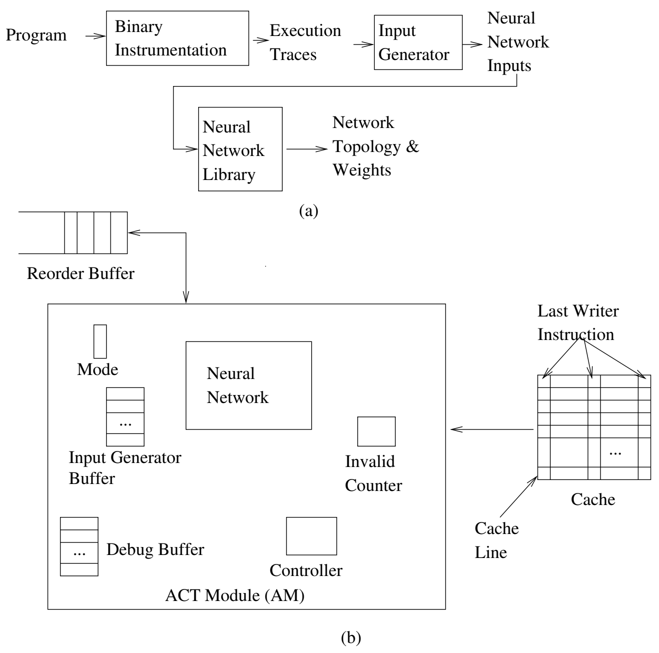


Figure 4. (a) shows how offline training works. (b) shows hardware extensions required

Figure 4(a) shows how offline training works. Initial offline training ensures that when ACT operates in online mode, it is aware of the most common RAW dependence sequences. The traces essentially contain a sequence of memory access instructions along with the memory addresses. A binary instrumentation tool (e.g., PIN [26]) can be used for this purpose. An input generator program analyzes the traces and forms RAW dependences. Recall that a RAW dependence is represented as $S \rightarrow L$ where L is the instruction address of a load that reads a memory word last written by a store at the instruction address S. S and L instructions may not be executed by the same processor. A RAW dependence $S \rightarrow L$ is assumed to belong to the processor that executes L. Each RAW dependence is labelled as either inter-thread or intra-thread. The input generator produces a RAW dependence sequence by forming a group of N consecutive RAW dependences (and their labels) together. More specifically, the generator associates each $S \rightarrow L$ with last $(N-1)$ RAW dependences $(S_{-i} \rightarrow L_{-i})$ for $1 \leq i \leq N-1$, where $(S_{-i} \rightarrow L_{-i})$ indicates i-th previous RAW dependence from the same processor. Sequences formed with $S \rightarrow L$ act as positive examples for the neural network. For every valid $S \rightarrow L$, the input generator also produces an invalid RAW dependence $S' \rightarrow L$, where S' is the store before the last store (to the same address) and associates it with the last $(N-1)$ valid dependences to create a negative example. Both positive and negative examples are provided to a neural network library [27] to determine the network topology as well as weights of the links. For a concurrent program, we use the same topology for each thread. However, the weights can be different across threads. The program binary is augmented to store the topology and weights of each thread (Section IV-B & IV-C).

C. *Online Testing and Training*

For online testing and training, we propose to add a per processor module, called *ACT Module* (AM). Figure 4(b) shows how it is integrated with a processor. It is connected to the reorder buffer (ROB) and the cache hierarchy of the processor. AM has a flag, called *Mode*, to indicate whether it is operating in online testing or training mode. The module contains a neural network, an *Input Generator Buffer*, and a *Debug Buffer*. The Input Generator Buffer stores recent N RAW dependences to form an input for the neural network. The Debug Buffer stores few recent invalid RAW dependences along with their neural network output. When a failure occurs, the debug buffer provides necessary information to diagnose the failure. Recall that the contents of the debug buffer are further processed offline to pinpoint the root cause. Although it is possible to use a single buffer instead of two, we use separate buffers to keep the design simple. Finally, the module has an *Invalid Counter* to keep track of invalid dependences and a controller to control the overall operation. We extend each cache line to store last writer instruction address. This section explains the operation of ACT assuming that the last writer information is stored at word granularity in the cache line. We also assume that when the line is displaced or invalidated, we write back the last writer information to main memory. The last writer information is piggybacked with cache coherence messages. Section V relaxes these constraints.

When a program starts execution, AM initializes its neural network with the topology and weights stored in the program binary. AM uses the weights of the current thread running in the processor. Whenever a load, L finishes execution (i.e., data is loaded into the local cache and returned to the pipeline) and becomes non-speculative (i.e., no unresolved branch in front of it in ROB), a RAW dependence, $S \rightarrow L$ is formed by using the last writer, S stored with the corresponding word in the cache line. Figure 5 shows the overall steps for processing the RAW dependence. The RAW dependence is inserted into the Input Generator Buffer. This buffer keeps the dependences in FIFO order. If the buffer is full, the oldest entry is dropped. The newly inserted dependence along with the last $(N-1)$ dependences from the same buffer forms an input for the neural network.

During online testing, the neural network calculates its output for the RAW dependence sequence formed with $S \rightarrow L$. If the output is positive, the sequence is valid. Otherwise, the sequence is invalid. The magnitude of the output can be thought of as an approximation of prediction confidence. If the sequence is predicted to be invalid, the sequence (i.e., instruction addresses) along with the neural network output is recorded into the Debug Buffer. When a failure occurs,

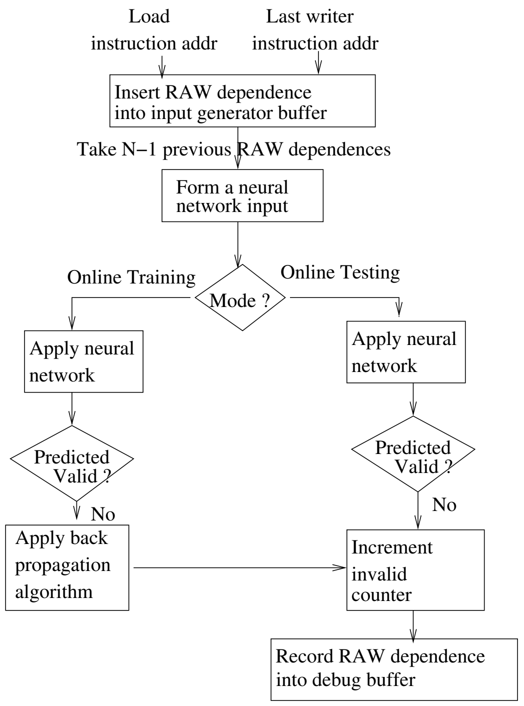

Load instruction addr
Last writer instruction addr
Insert RAW dependence into input generator buffer
Take N−1 previous RAW dependences
Form a neural network input
Online Training
Mode ?
Online Testing
Apply neural network
Apply neural network
Predicted Valid ?
Predicted Valid ?
No
No
Apply back propagation algorithm
Increment invalid counter
Record RAW dependence into debug buffer

Figure 5. How each RAW dependence is processed.

the contents of the Debug Buffer are used for pinpointing the root cause.

When the dependence sequence is valid but the neural network predicts it to be invalid, the program continues to execute without failure. For this reason, every time a dependence sequence is predicted to be invalid, the Invalid Counter is incremented. Just in case the program continues without failure, AM can check the counter to determine how many mispredictions have occurred since it last checked the counter. Thus, this counter is used to calculate misprediction rate periodically. If this rate goes above a threshold, AM enters into online training mode.

During online training, all RAW dependences are considered to be correct. For every load, a RAW dependence sequence is formed. If the RAW dependence sequence is predicted to be invalid, the prediction is considered to be incorrect. The neural network, then, applies back propagation learning algorithm (Section II-A) to adjust the weights of the links. AM also increments the Invalid Counter. If the dependence sequence, on the other hand, is predicted to be valid, then the neural network does not learn it anymore. Although we consider all RAW dependences to be valid during online training, some of them might, in fact, be invalid and can cause a failure. Therefore, as before, all dependence sequences classified by the neural network as invalid are still logged into the debug buffer. In case of a failure, the contents of the debug buffer are used to find the root cause. Similar to online testing mode, AM periodically checks the Invalid Counter to calculate the misprediction rate. If the rate drops below the misprediction threshold, AM enters into online testing mode again. Thus, AM continuously alternates between online testing and training mode. Note that it is not possible to form negative examples during online training mode without keeping more than one writer information per word. To keep the hardware requirements minimal, we choose to accept this limitation.

Note that in both mode of operations, a RAW dependence is formed when a load finishes execution and is waiting for its turn to retire. Since AM is a local structure and tightly integrated with the processor core, the process of forming a RAW dependence and feeding it to the neural network can be done before the load reaches the head of the ROB. Only if the neural network has its internal buffer full (more on this in Section IV) and cannot accept any more input, the load may get stalled. When the neural network is finally able to take further inputs, the load is free to retire. Note that the load does not need to wait until the neural network calculates its prediction. As soon as, the neural network accepts the input corresponding to the load, it is free to retire.

If the neural network predicts an invalid RAW dependence sequence to be valid and a failure occurs, ACT will not be able to diagnose it. Since ACT learns continuously during every execution, this happens rarely in steady state. We have not encountered this case in any of the 16 failures in our experiments (Section VI). If such case occurs and the programmer, with the help of other approaches, is able to pinpoint the invalid dependence sequence, the sequence can be fed to the neural network (similar to offline training) as a negative example.

D. *Offline Postprocessing for Ranking*

The Debug Buffer contains last few (e.g., 600) invalid RAW dependence sequences. Analyzing all of them to find the root cause of a failure is counterproductive. To rectify this problem, ACT processes the contents of the buffer in two steps - Pruning and Ranking. The processing is done offline after a failure. For this purpose, we run the program few more times (e.g., 2 times in our experiments) to collect traces of correct executions (similar to initial offline training). Unlike other schemes [10], [11], we do not reproduce the *failure* execution, rather correct executions. In fact, it is possible to use the same execution traces that were used for initial offline training. We need to ensure that the traces contain RAW dependences from the code sections where the dependence sequences of the Debug Buffer belong. We can do so if we use actual production inputs (if available) or test inputs (otherwise) for those code sections. The traces are passed through Input Generator to form RAW dependence sequences. These are the sequences that occur during correct executions. Let us call the set of these sequences as Correct Set. Any dependence sequence that is present in the Correct Set is removed from the Debug

Buffer. We call it pruning process. A bug occurs due to any of the remaining sequences.

After pruning, we need to rank the remaining sequences according to their likelihood of being the root cause. Each of the remaining sequences has one or more RAW dependences that cause a mismatch with the sequences from the Correct Set. Our ranking algorithm produces higher rank for a sequence if it has more matched dependences. This is due to the intuition that if we see more RAW dependences to match, we are more confident that the mismatched dependence is the buggy one. One could argue that the sequence with the most mismatches is the root cause. Such a sequence indicates that something has already gone wrong, the program starts to execute wrong sequence of instructions and hence, we find lot of mismatched dependences. Therefore, the sequence with the highest match (and hence, the lowest mismatch) is the point from where the program is likely to go wrong. For each remaining sequence $(A_1,A_2,...,A_N)$ of the Debug Buffer (where A_i, for $1 \leq i \leq N$, denotes a RAW dependence), we count the number of matched RAW dependences. The sequences of the Debug Buffer are sorted (descending) according to this number. If two sequences have the same number, then the one with the most negative neural network output appears first in the sorted order. Recall that the Debug Buffer contains a dependence sequence as well as its neural network output. After sorting, a programmer can inspect the sequences from the top as the root cause.

As an example, let us assume the RAW dependence sequence length, N to be 3. Also assume that the Debug buffer has three sequences (A_1,A_2,A_4), (B_1,B_2,B_3) and (A_1,A_5,A_6). Correct Set contains (A_1,A_2,A_3) and (B_1,B_2,B_3). Pruning will remove (B_1,B_2,B_3) from the Debug Buffer since it is present in the Correct Set. Consider one of the remaining sequences (A_1,A_2,A_4). Correct Set has (A_1,A_2,A_3). So, after A_1 and A_2, mismatch occurs. Thus, (A_1,A_2,A_4) has 2 matched RAW dependences. Similarly, (A_1,A_5,A_6) has 1 matched RAW dependence. So, our ranking will place (A_1,A_2,A_4) before (A_1,A_5,A_6). Had the Debug Buffer has any more sequences with 2 matched RAW dependences, we would consider the associated neural network output.

IV. Implementation of ACT Module

We focus on a digital neural network design (similar to Esmaeilzadeh et al. [6]). We add three new instructions to the ISA to facilitate the operation of AM.

A. Neural Network Design

A fully configurable neural network time multiplexes an arbitrary network topology to a fixed number of neurons and incurs scheduling overhead. To eliminate the overhead, we use a partially configurable neural network with only one hidden layer. We limit the maximum number of inputs to a neuron to M. ACT requires only one output from the neural network. Therefore, the output layer contains one neuron. Thus, we can have a network topology in the form of $i \rightarrow h \rightarrow 1$, where the input layer has i inputs for $1 \leq i \leq M$, the hidden layer has h neurons for $1 \leq h \leq M$, and the output layer has one neuron. This gives us a search space of M^2 topologies. This neural network can be easily mapped to a three stage pipeline (Figure 6(a)) without time multiplexing.

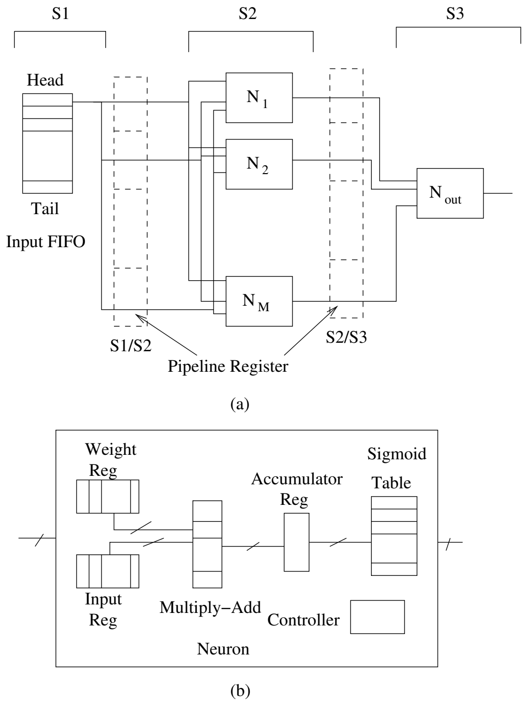


Figure 6. Neural network design for ACT.

The first stage, S1, is the input layer that contains an input FIFO. When an input corresponding to a RAW dependence arrives, it is stored into the tail of the FIFO. If the FIFO is full, then the load corresponding to the RAW dependence is prevented from retiring. The FIFO supplies an input from its head to the next stage, S2. S2 forms the hidden layer. When neurons N_1, N_2, ... N_M calculate results, they are sent to the next stage S3. S3 is the output layer that contains one neuron. S1 can be done in one cycle. Each of S2 and S3 takes T cycles where T is the number of cycles it takes for a neuron to calculate the output. If the FIFO is full, this pipeline takes an input after every T cycles. If the FIFO is not full, it can take an input in every cycle.

A neural network operates in a pipelined fashion only if it is used during online testing. During online training, after producing the output, the output neuron calculates the error and propagates it back to the hidden layer which, then, propagates it back to the input layer. To enable back propagation, the connections between S1, S2, and S3 have to be bidirectional. During learning, a neural network has to process an input completely and then, can accept another input. Thus, the neural network takes an input after every

4T cycles if the input FIFO is full. If the FIFO is not full, it takes an input in every cycle.

Figure 6(b) shows the internal structure of a neuron. It is similar to the one proposed by Esmaeilzadeh et al. [6]. It contains Weight Registers and Input Registers to hold M weights and M inputs respectively. A weight of zero is used to disable a particular input. A neuron also contains some multiply-add units, an accumulator register, a sigmoid table, and a controller. During testing, a neuron produces output by calculating $o = g(\sum_{j=0}^{M} W_j a_j)$ where g is a sigmoid activation function, W_j and a_j are weights and inputs respectively. The multiply-add units are used to perform M multiplications and M additions. If a neuron uses only one multiply-add unit, then we need $M \times T_{mul-add}$ cycles to complete all the multiplications and additions, where $T_{mul-add}$ cycle is the latency of a multiply-add unit. After that, T_{rest} cycles are needed for using the accumulator register and sigmoid table. Typically, $M \times T_{mul-add}$ would be much larger than T_{rest}. However, if we use x multiply-add units in a cascaded fashion, we would need $M/x \times T_{mul-add}$ cycles for completing the multiplications and additions. Thus, by increasing the number of multiply-add units, we can reduce the latency of a neuron.

Instruction	Description
chkwt %r	Check if a thread with id [r] has weights
ldwt %r_i, %r_w	Read weight register at index [r_i] into register r_w
stwt %r_i, %r_w	Write [r_w] into weight register at index [r_i]

Table II
NEW INSTRUCTIONS ADDED BY ACT. [.] IS USED TO DENOTE THE CONTENT OF A REGISTER.

During training, a neuron updates each of its weights, which requires one multiplication and one addition (Section II-A). Thus, for M weights, a neuron performs M multiplications and M additions. This is similar to testing mode. Therefore, the same analysis also applies here. However, for propagating the error back, a neuron requires M additional multiplications. We use as many additional multipliers as the number of multiply-add units. Thus, the additional multiplications can be done in parallel with the original multiplications-additions.

B. New Instructions

ACT adds three new instructions to the ISA. They are shown in Table II. After offline training, the weights of links along with thread id are stored in the program binary. It is possible that during offline training, a particular thread has not been created. As a result, that the thread won't have any weights. This can be checked by using *chkwt* instruction. *ldwt* and *stwt* are used to read from and write into weight registers of the neural network that belongs to the current processor's AM. The weight registers of all neurons of the neural network form an array and therefore, the registers are accessed with an index. Register r_i specifies the index of the weight register and r_w contains the corresponding weight.

C. Modifications to Thread Library

We consider a general programming model using a thread library e.g., *pthread*. Other models (e.g., task parallel) are left as a future work. Each thread is identified by an id provided by the thread library. Although different execution can create threads in different order, we can uniquely identify a thread by the order in which the parent thread spawns it [28]. Therefore, we modify the thread library to generate id based on the parent thread and spawning order. We also extend the thread creation function (e.g., *pthread_create*) to initialize the link weights. The function should check if the thread has any weights stored in the binary. If not, it should initialize the registers with some default weights. These weights will cause too many mispredictions and eventually, AM is forced to enter into online training mode to learn RAW dependence sequences. If, on the other hand, the weights are available for the thread, we initialize each weight by using a load followed by a *stwt* instruction. On termination of a thread, we read out the weight registers of the executing processor. We do this by augmenting the thread termination function of a threading library (e.g., *pthread_exit*) by a sequence of *ldwt* instructions. The weights read this way, are saved in a special log file. The log is used to patch the binary with new weights. Thus, the training done during one execution can be utilized during the subsequent executions. For a neural network of $M \rightarrow M \rightarrow 1$ topology, there can be at most $M^2 + 1$ weights. So, each thread can have at most $(M^2 + 1) \times 4$ byte additional data in the binary. The data is accessed using a loop of *ldwt* or *stwt* instruction.

D. Handling Context Switch and Thread Migration

The weight registers are considered to be part of the architectural state of the processor. Therefore, during context switch or thread migration, these registers should be saved and restored by the OS. This is done by issuing a sequence of *ldwt* and *stwt* instructions. To reduce the overhead, OS can use some lazy context switch technique [29]. The neural network pipeline should flush the in-flight inputs before context switch or thread migration starts.

V. ADDITIONAL ISSUES

Cache Line Extension: Maintaining last writer information precisely is very expensive. Like prior work [19], [20], we choose three simplifications. *First*, we track last writer information only at cache line granularity. This increases misprediction rate of a neural network. However, our results indicate the increase to be insignificant. *Second*, during cache line eviction, we do not write last writer information back to the main memory. As a result, we fail to form RAW dependences for some loads. Since ACT operates all the time during an execution and even subsequent executions, eventually it will be able to form RAW dependences for later occurrences of the same load and diagnose bugs. *Third*, we piggyback last writer information with coherence messages only for a read miss that generates a cache-to-cache message. In MESI protocol, this implies that we piggyback last writer information only for a read miss on a dirty line. As before, the bug diagnosis ability is not affected in the long run.

Dynamic Loading of Libraries: In order to handle dynamically loaded libraries, last writer instruction address is stored in the form of a library id and an offset into the library.

Filtering of Loads: In order to reduce overhead, only RAW dependences corresponding to loads of non-stack data are considered. We use a simple filtering technique where any load that uses stack registers (e.g., ESP and EBP) is ignored.

Overfitting: The neural network of AM may overfit (i.e., learn only frequently occurring RAW dependences). When a rare RAW dependence occurs, it may be predicted as invalid. If a failure occurs after that and the offline processing cannot prune the rare dependance, the rank of the root cause might be off by at most one due to the rare dependence.

VI. Evaluation

The goal of this section is to (i) demonstrate prediction ability of ACT, (ii) show its failure diagnosis ability, (iii) calculate its overhead, and (iv) assess the impact of false sharing.

Architecture	Chip multiprocessor with **4**, 8, 16 cores
Pipeline	Out-of-order; 2-issue/3-retire, 104 ROB
Private L1 cache	32KB WB, 4-way associative., 2-cycle rt
Private L2 cache	512KB WB, 8-way associative, 10-cycle rt
Cache line size	4, **32**, 64, 128 B
Coherence	Snoopy MESI protocol at L2; 32B-wide bus
Memory	300 cycle rt
Parameters of a neuron	
Max input	10
Multiply-add unit	1, **2**, 5, 10 with 1-cycle latency
Accumulator	1-cycle latency
Sigmoid unit	1-cycle latency
Input FIFO	4, **8**, 16 entries
Parameters of ACT Module	
Total neuron	11
Input generator buffer	5 entries
Debug buffer	600 entries
Misprediction threshold	5%

Table III
Multicore architecture simulated. Bold faced ones are the default parameters.

A. *Experimental Setup*

We use a PIN [26] based tool to collect traces. The traces are analyzed using a neural network library implemented in OpenCV [27]. We use PIN along with a cycle accurate simulator [24] and OpenCV to model a multicore with ACT modules. We use applications from SPLASH2, PARSEC, SPEC INT 2006, and GNU coreutil in our evaluation. We test with 11 real and 5 injected bugs from Apache, MySQL, PBZip2, Aget, Memcached, GNU coreutil, PARSEC, and SPLASH2. Table III shows configuration parameters.

Program	# of Traces for Training	# of RAW Dep	Topology	%Mispred. Rate
fft	9	4	8→6→1	0.017
fmm	6	4	8→5→1	0.114
radix	1	4	8→4→1	0.107
volrend	10	4	8→6→1	0.226
lu	1	4	8→4→1	0.000
canneal	1	5	10→4→1	0.001
bodytrack	1	5	10→4→1	2.242
raytrace	6	4	8→5→1	0.008
swaptions	4	5	10→5→1	0.301
fluidani.	10	5	10→4→1	0.090
gzip	10	4	8→10→1	0.000
mcf	10	4	8→10→1	1.179
bc	10	4	8→10→1	1.561
Average				0.448

Table IV
Training of neural networks.

B. *Prediction Ability*

We collect 20 execution traces. 10 traces are used for testing. Among the rest, up to 10 traces are used for training. We vary number of RAW dependences from 1 to 5 to form inputs. The number of neurons in the hidden layer is varied from 1 to 10. We use back propagation algorithm with a learning rate of 0.02. During testing, we calculate the number of mispredictions. It is shown as a percentage of total instructions. We select the topology that produces the lowest misprediction rate. Note that during these experiments, we do not have any invalid RAW dependences in our testing data. Thus, the mispredictions are essentially false positives. Table IV shows the results. Except for bodytrack, mcf, and bc, others have very low misprediction rate. Overall, we achieve an average misprediction rate of 0.4%. In order to further assess ACT's prediction ability, we intentionally form invalid RAW dependences (e.g., RAW dependences with a store instruction before the last one). Figure 7(a) shows the misprediction (i.e., false negative) rate in such cases. The average misprediction rate is 0.18%.

C. *Bug Diagnosis Ability*

We test 11 real world bugs from open source programs. They are shown in Table V. The third column shows how many traces are used for initial training. We use 10→6→1 as the network topology for all the bugs except gzip. For gzip, we use 8→10→1 as the topology. The fourth column shows where in the debug buffer the invalid RAW dependence sequence (i.e., root cause) was initially found. The fifth column shows what percentage of dependence sequences are filtered by the offline postprocessing. The sixth column shows the final rank of the root cause. We compare ACT against a continuous learning based scheme, Aviso [12] and a state-of-the-art sampling based scheme, PBI [10]. Although the actual goal of Aviso is to avoid failures, it can be used to to diagnose a failure by inspecting the constraints that Aviso finds very likely to be related to the failure. For

Bug	Description	# of Traces for Train.	ACT			Aviso	PBI	Prog. Status
			Debug Buf. Pos.	Filter (%)	Rank	Rank (# of fail.)	Rank (Total pred.)	
Aget	Order. vio. on *bwritten*	4	8	78	4	3 (3)	- (35)	Comp.
Apache	Atom. vio. on ref. counter	15	344	96	7	- (10)	136 (177)	Crash
Mem -cached	Atom. vio. on item data	13	159	92	2	34 (6)	6 (33)	Comp.
MySQL#1	Atom. vio. causing a loss of logged data	8	2478	97	1	628 (3)	222 (748)	Comp.
MySQL#2	Atom. vio. on *thd→proc-info*	10	34	91	1	294 (4)	153 (500)	Crash
MySQL#3	Atom. vio. in *join-init-cache* causing out of bound loop	7	519	96	3	122 (4)	- (390)	Crash
PBzip2	Order. vio. between threads	7	2	0	1	2 (2)	1 (14)	Crash
gzip	Semantic bug for *get_method* wrong file descriptor	2	2	0	1	- (-)	- (1)	Comp.
seq	Semantic bug for wrong terminator in *print_numbers*	2	5	3	5	- (-)	- (11)	Comp.
ptx	Buffer overflow of *string* in *get_method* func.	2	1	0	1	- (-)	1 (2)	Comp.
paste	*collapse_escapes* reads out of buffer of *string*	2	8	0	8	- (-)	4 (4)	Crash

Table V
DIAGNOSIS OF REAL BUGS IN APPLICATIONS.

Prog.	Function	Filter (%)	Rank
fft	Touch -Array	93	7
fmm	VListInter -action	90	23
fluid.	Compute -Densities -MT	93	18
lu	TouchA	71	1
swapt.	worker	84	6
Avg		86	

Table VI
INJECTED BUG IN NEW CODE

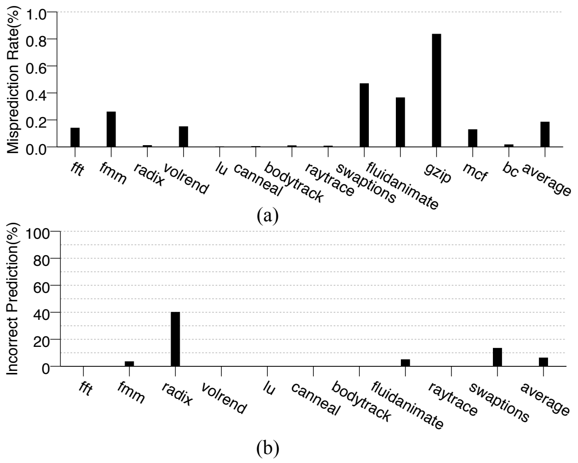


Figure 7. (a) shows misprediction rate with invalid RAW dependences. (b) shows incorrect prediction with new code.

Aviso, after a failure occurs, we check whether it can find the constraint involving the root cause. If it does not find the constraint, we feed it with more failure runs until it finds such a constraint. We use a maximum of 10 failure runs. The seventh column shows the rank and the number of failure runs required for Aviso. PBI, on the other hand, samples cache events and branch outcomes and uses a statistical technique to rank the root cause. We use 15 correct and 1 failure executions for PBI. We use 15 correct executions because ACT uses at most 15 execution profiles for training. PBI, as originally proposed, uses 1000 executions out of which around half are failure runs. In each run, it randomly samples 1 out of 1000 instructions. Since we use only 16 executions, we sample at every instruction to compensate for the reduced samples. In a sense, we implement an "extreme" version of PBI that looks at every instruction and every event to find the root cause. The eighth column shows the rank and total number of predicates (i.e., instruction-event pair) reported by PBI.

Out of 11, 5 bugs cause a crash. For the others, the programs run to completion with ill effects (e.g., log corruption). Except for *MySQL#1*, buggy RAW dependence sequences are always found in the default sized (i.e., 600 entries) debug buffer. For *MySQL#1*, ACT cannot find the buggy sequence without a larger buffer. Offline filtering removes (on average) 50% dependences. ACT finds the buggy sequence of every failure with a very accurate ranking. For 9 bugs, the ranking is within 5. The worst rank is 8. Aviso requires the same bug to occur more that once before it can find a bug related constraint. The ranks of the constraints are worse too. For Apache, it does not find such constraint even after 10 failures. Aviso does not work for sequential bugs. PBI works for both types of bugs. For a single failure run, its ranks are worse than ACT except for *paste*. Moreover, PBI misses 2 sequential bugs because the branch outcomes do not change from a successful run to a failure run. It also misses *Aget* and *MySQL#3* bugs. The first miss occurs because the buggy instruction observes the same cache event (invalid state) in both successful and failure runs. The second miss occurs because the buggy instructions are so far apart that the cache events do no show any consistent pattern. PBI produces a rank less than 5 only for 3 out of 11 failures. Overall, ACT performs better than PBI in 10 out of 11 failures.

D. Adaptivity

To demonstrate ACT's adaptivity, we collect RAW dependences of a program and then remove all dependences from a randomly chosen function. We only use concurrent programs because they are the hardest to predict. ACT is trained with the remaining dependences. Then, it tests the excluded dependences. This shows how ACT behaves when the program is augmented with the new code. Figure 7(b) shows the percentage of new RAW dependences reported as incorrect. On average, ACT predicts 6.16% of total unique dependences incorrectly (i.e., accuracy is 93.84%). Note that we did not use "misprediction rate" because some of the excluded functions contain loops and hence, the same

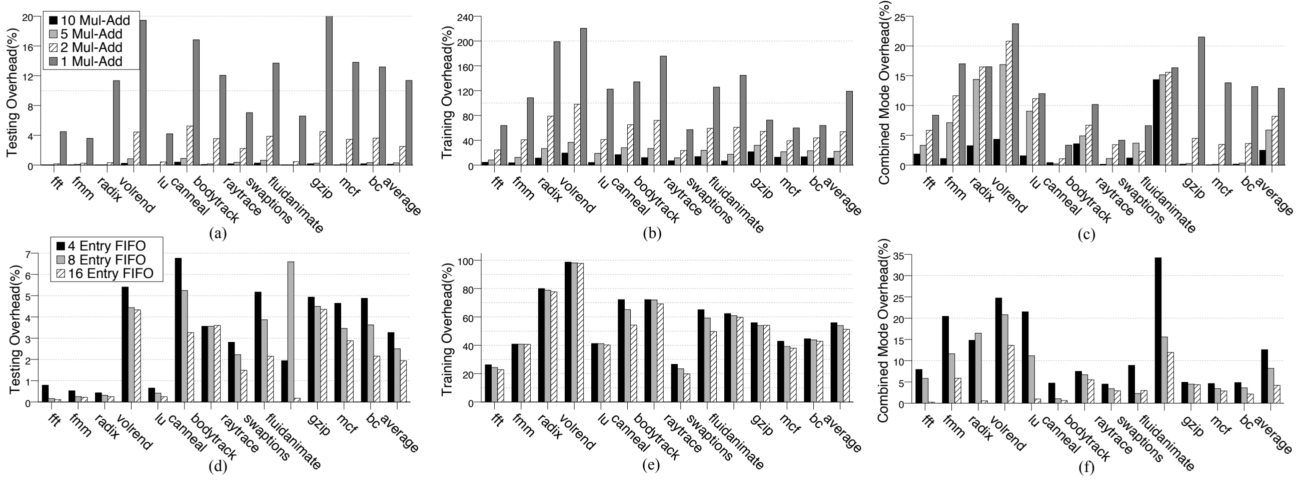


Figure 8. (a)-(c) show execution overhead for different number of multiply-add units. (d)-(e) show the same for input FIFO size.

dependence appears multiple times. As a result, misprediction rate which treats every occurrence of a dependence separately, would be too distorted to provide any insight. We further evaluate ACT's diagnosis ability in the new code by injecting unsynchronized writes in some of the programs to induce failures. ACT diagnoses all of them (Table VI). The offline processing filters 86% dependence sequences and produces the maximum rank of 23. So, ACT has a reasonable diagnosis ability even for never-before-seen code. We can expect the ranking to be similar to that of Table V after learning the new code few more times.

E. Execution Overhead

We vary the number of multiply-add units as 1, 2, 5, and 10. To get the lower and upper bound of execution overhead, we use a misprediction threshold of 10% and 0% respectively. This causes a program to run entirely in online testing mode (Figure 8(a)) or training mode (Figure 8(b)). In both cases, the overhead decreases as we increase the number of multiply-add units. For our default 2 multiply-add unit configuration, average testing overhead is 2.5% whereas average training overhead is 54%. For single multiply-add unit configuration, these numbers are 11% and 119% respectively. For the default misprediction threshold of 5%, ACT switches between online testing and training mode. In this combined mode (Figure 8(c)), average overhead is 8.2% for our default 2 multiply-add unit configuration. This is the overhead we can expect in the steady state (i.e., when ACT runs mostly in online testing mode). We experiment how the execution overhead changes with the input FIFO size of a neural network. Figure 8(d), (e), and (f) show the testing, training and combined mode overhead respectively. Larger buffer decreases the overhead. The average testing overhead decreases form 3.3% to 2.5% to 1.9% as buffer size increases from 4 to 8 and then, to 16. For training, the numbers are 56%, 54%, and 51% respectively. For combined mode, the numbers are 13%, 8.2%, and 4.2% respectively. Our default size is 8. We also implement ACT completely in software using PIN. The overhead ranges from 104X to 2091X. This justifies the necessity of neural hardware.

F. Neural Network Design

We compare ACT neural network with an alternative time multiplexed design [6] that has additional scheduling and queuing latency. On average, ACT has 8.2% overhead wheres the alternate design has 17.2% overhead (Figure 9(a)).

G. Network Traffic Overhead

Figure 9(b) shows traffic overhead caused by piggybacking of last writer in parallel programs. As the number of core increases, overhead increases slightly. The average overhead is $\approx$1.9% for 4 and 8 cores. For 16 cores, it is 2.3%.

H. Effect of False Sharing

We run experiments with cache line size 4, 32, 64, and 128 bytes. Figure 9(c) shows the data. When line size increases from single word (i.e., 4 bytes) to multi words, misprediction rate also increases slightly. Average misprediction rate is 0.4%, 1.2%, 2.0%, and 2.6% for 4, 32, 64, and 128 bytes cache line respectively.

VII. Discussion

There are few issues that we need to address in future to make ACT more practical. *First,* execution overhead, especially when ACT runs in online training mode can be significant (in the worst case, around 100% in volrend in Figure 8(b)). To reduce it, we can think of sampling during online training mode. *Second,* we need to make ACT portable across different parallel programming models. *Third,* while RAW dependence sequence provides quite

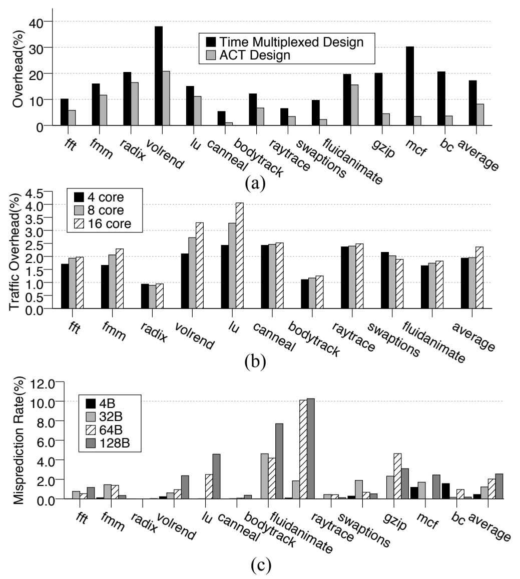


Figure 9. (a) & (b) show design comparison and traffic overhead. (c) shows misprediction rate with different cache line size.

favorable diagnosis ability, it may not work for all bugs. Specially, if the RAW dependences do not differ between successful and failure runs, ACT will not be able to diagnose the failure. Thus, we need to incorporate more invariants to diversify ACT.

VIII. Related Work

Engler et al. [30] proposed the first work to statically determine invariants and check for violations. The invariants are programming rules implied by the code. Daikon [31] starts with a fixed set of possible invariants. It runs the program over test inputs and determines which invariants are always satisfied. DIDUCE [22] dynamically detects value invariants of different expressions and calculates confidence of every invariant violation. Unlike ACT, DIDUCE works for sequential bugs only. AVIO [17] works on communication invariant for concurrency bug detection. AVIO detects single variable atomicity violations. AtomAid [18] uses bulk system [32] to avoid atomicity violations. In order to handle different types of concurrency bugs uniformly, PSet [19] proposes to use an invariant based on inter-thread communication. It is called Predecessor Set. Predecessor set of a memory access instruction includes all other remote memory access instructions upon which this one immediately depends. Bugaboo [20] extends PSet by incorporating a limited form of context information. DefUse [21] uses a slightly different approach where it finds out all definitions upon which a read depends. However, AVIO, AtomAid, PSet and Bugaboo do not learn continuously. Aviso [12] is a continuous learning based scheme that uses scheduling constraints to avoid concurrency bugs. Each constraint is a pair of events (e.g., synchronization operation, signal handler, and shared access). When a program crashes, Aviso determines the scheduling constraints that are likely to cause the failure. Recon [11] is a failure diagnosis technique for concurrency bugs. Unlike ACT, both Aviso and Recon require multiple failure executions to be effective. PBI [10] uses hardware performance counters at production runtime to sample different events. It can diagnose both concurrency and sequential bugs but requires multiple failure runs. CCI [23] instruments program at certain points and collects some events at those points during execution. It samples events to keep the runtime overhead low. It uses a statistical framework to find the root cause of production failures. Like PBI, CCI requires multiple failure runs to be accurate. Moreover, for some applications, its overhead is too large for production run use. There is a large body of work on program slicing [33], [34], [35] to diagnose software failures. Slicing is an effective diagnosis technique. ACT can complement a slicing based technique by providing an initial set of dependences (e.g., contents of the debug buffer).

IX. Conclusion

This is the *first* proposal to apply neural hardware for diagnosing software failures. Our proposed scheme, ACT, first determines neural networks' topology and links' weights by analyzing traces. When a program starts execution, the hardware implemented neural networks are initialized with the topology and weights. The networks are used to test and learn RAW dependence sequences alternatively online. When a failure occurs, postprocessing of invalid dependences accurately pinpoint the root cause without requiring to reproduce the failure. We provided a detailed design of ACT in a typical multiprocessor system using a three stage pipelined neural network. Our evaluation of ACT on a variety of parallel and serial programs showed that ACT can diagnose failures from all 16 bugs. ACT has an average execution overhead of 8.2%.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We also thank the members of PALab for the discussions and comments. Special thanks go to Tongping Liu for his feedback on the initial draft of the paper. This work is supported by the startup package provided by University of Texas at San Antonio and NSF under Grant No. 1319983.

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software." https://www.jbs.cam.ac.uk/media/2013/, 2013.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, March 2008.

[3] P. Godefroid and N. Nagappan, "Concurrency at Microsoft - An Exploratory Survey," in *Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[4] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous real-world inputs can open up alternative accelerator designs," in *ISCA*, June 2013.

[5] K. Fan, M. Kudlur, G. S. Dasika, and S. A. Mahlke, "Bridging the computation gap between programmable processors and hardwired accelerators," in *HPCA*, February 2009.

[6] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, December 2012.

[7] D. A. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in *ISCA*, January 2001.

[8] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *CICC*, September 2011.

[9] Qualcomm, "Zeroth processor." http://www.qualcomm.com/media/blog/2013/10/10/introducing-qualcomm-zeroth-processors-brain-inspired-computing.

[10] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu, "Production-run software failure diagnosis via hardware performance counters," ASPLOS, March 2013.

[11] B. Lucia, B. P. Wood, and L. Ceze, "Isolating and Understanding Concurrency Errors Using Reconstructed Execution Fragments," in *PLDI*, June 2011.

[12] B. Lucia and L. Ceze, "Cooperative Empirical Failure Avoidance for Multithreaded Programs," in *ASPLOS*, March 2013.

[13] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: Effective sampling for lightweight data-race detection," PLDI, June 2009.

[14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," PLDI, June 2005.

[15] 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.

[16] 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.

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

[18] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-Aid: Detecting and Surviving Atomicity Violations," in *ISCA*, June 2008.

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

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

[21] 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.

[22] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *ICSE*, June 2002.

[23] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and sampling strategies for cooperative concurrency bug isolation," OOPSLA, October 2010.

[24] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC Simulator," January 2005. http://sesc.sourceforge.net.

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

[26] 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.

[27] OpenCV, "Open Source Compute Vision." http://opencv.org/.

[28] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jrapture: A capture/replay tool for observation-based testing," ISSTA, August 2000.

[29] NetBSD Documentation, "How lazy FPU context switch works." http://www.netbsd.org/docs/kernel/lazyfpu.html, 2011.

[30] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *SOSP*, October 2001.

[31] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, 2007.

[32] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk enforcement of sequential consistency," in *ISCA*, June 2007.

[33] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, "Using likely invariants for automated software fault localization," in *ASPLOS*, March 2013.

[34] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," ASE, November 2005.

[35] S. Tallam, C. Tian, and R. Gupta, "Dynamic slicing of multithreaded programs for race detection.," in *ICSM*, pp. 97–106, IEEE Computer Society, September 2008.