Buggaboo - A Machine Learning Method for Determining the Bugginess of Concurrent Program Executions (or: CSE573 Final Project Report)

Brandon Lucia

December 11, 2008

1 Introduction

The advent of multi-processor computers has provided programmers with a substrate for programs which are faster and potentially more efficient. Programmers need to write concurrent programs to harness the performance available on these systems, but at the same time, concurrent programming is difficult, error prone, and largely inaccessible to many programmers. Writers of parallel code are faced with a spectrum of bugs peculiar to concurrent software, such as Atomicity Violations, Ordering Violations, Deadlocks, and Data-races, to name just a few. These bugs are subtle, difficult to reproduce, and many times difficult to even detect at all, in spite of their potentially catastrophic results. Prior work in concurrency error detection [2–4] has focused on a number of bugs, the results of which are non-obvious to human observers, but can cause crashes much later in an execution or compromise system security. One example of such a bug occurs in the widely deployed MySQL database server; In the logging subsystem there is a bug which if executed causes a series of database actions that have occurred for a window of time to not be logged in the database access log. This is a major hole in the security of the system, as an intruder could exploit this bug to "cover their tracks". The subtlety and importance of this class of bugs lead to the need for solutions to two problems: (1) Detecting that an execution is in fact buggy when by the output it is not obvious that this is the case, and (2) Determining where in the source code the bug which may have manifested itself occurred. In this work, I focus on a solution to the first of these problems, leaving the second as an extension which could be built upon this work, but is nonetheless the subject of future work. To solve the problem of determining the bugginess of an execution of a concurrent program, I leverage the similarity in inter- and intra-thread shared memory communication patterns. In specific, I devise a method by which a set of execution traces known to have been either buggy or non-buggy are used to create a classifier, based on their patterns of shared memory communication, and this classifier is used to determine the bugginess of previously unseen executions.

2 Background

In this section, I will discuss shared memory communication in concurrent programs, and describe exactly the features of communication that will be leveraged in this work.

2.1 Representing Shared Memory Communication using Read/Write Sets

Concurrent programs written to target systems using shared memory do not perform explicit inter- and intra-thread communication, as in message passing programs. Instead, communication between threads of execution is implicitly defined by sequences of accesses to shared memory. For the purpose of this work, a *concurrent program* can be considered as a set of possible memory operations on shared memory state. Each operation is either a Read, or a Write, and each corresponds to some line of code in a high-level language, written by a programmer — these are *static* memory operation. A *concurrent execution* is the sequence of memory operations from the concurrent program which actually occurs when the program is run; *i.e.* a list of *dynamic instances* of the static operations in the program. Because an execution is a sequence of dynamic instances of the program operations, one static operation may occur many times in an execution (*i.e.:* a loop) possibly in any thread. Each static operation is identified by a Program Counter (PC), and each dynamic instance of a given PC is identified by a tuple of a unique identifier for the thread which executed it and its PC, (*e.g.*: (T1,0x808089a)). Examining a trace of the operations in a concurrent execution, it is possible to assemble, for possible combination of static Read and thread the set of Writes from which the Read read data. For instance, suppose one thread, $T1$ is in a loop which repeatedly executes a read at PC $0x01$, which reads some memory location M. Then, another thread, $T2$ is executing a long sequence of code which performs several different operations on M at PCs $0x02$, $0x03$, and $0x04$. At each iteration of the loop in $T1$, when the read at $0x01$ is encountered, it "asks" which dynamic write was the last to write to location M, and it adds that write to its *Write Set.* In one execution, it might be that in the first iteration of the loop in $T1$, the last write to M before the read from M at $0x01$ in $T1$ was the write identified by the tuple ($T2$, $0x02$), in the second iteration, ($T2$,$0x03$), and in the third iteration ($T2$,$0x04$). At the completion of these operations, the Write Set for ($T1$,$0x01$) is the following: $WS_{(T1,0x01)} = \{(T2, 0x02), (T2, 0x03), (T2, 0x04)\}$. By just evaluating a trace of all memory operations which occurred in an execution across all threads, it is possible to compute these sets off-line. In a similar way to that in which the Write Set of every Read is computed, the *Read Set* of every Write can be computed — this is the set of all Reads which read data from a particular Write operation.

Write Sets (or Store Sets) have been used in prior work [1] for the purpose of supplementing prediction techniques in single threaded programs, but has not been used for the purpose of bug determination or detection. In this work I leverage the fact that when executed for a sufficiently long time, the write and read sets for a give Read or Write tend to be the same across runs. These sets, however, tend to differ between runs if one run being considered was buggy, and the other was non-buggy. This is the cornerstone of the technique used in Buggaboo.

3 Buggaboo Determines the Bugginess of an Execution

In this section, I describe the method by which I use the write and read sets computed across a variety of runs which are labeled as buggy or non-buggy to determine the bugginess of never-before-seen executions. In this work, I use the vector of Write sets or Read sets computed for an execution as the vector of features for that execution. The value of each feature is the size of the set. Inference of the bugginess of an execution is performed using these feature vectors.

3.1 Computing the Distance Between Executions in Read/Write Set Space

An execution, E maps to a set of Write Sets, $R_{(Ti,0xj)}$, and $W_{(Ti,0xj)}$ for all threads Ti and all relevant PCs $0xj$. Given two executions E_1 and E_2, it is possible to determine the "distance" in Write Set space between them. This distance is exactly the sum of the differences in size of each Write Set in each execution. In a similar way, the distance between the two executions can be computed in the space of Read Sets as well.

3.2 Using Distances in Feature Space to Infer Bugginess

Because executions of a similar class (Buggy or NonBuggy) tend to have similar memory access patterns, their Write Sets and Read Sets tend to be similar. To determine the bugginess of never-before-seen executions, I leverage this property and create two types of Nearest-Neighbor Bugginess Classifier. The first classifier simply determines, for a new execution, which labeled execution is nearest it in feature space (either Write Set or Read Set space), and classifies that new run with the class of its nearest neighbor. Extending on this classifier, I also developed a voting-based nearest neighbor classifier which computes the n nearest labeled

	% Correctly Classified			
	Write Sets		Read Sets	
	Near. Neigh.	Voting Near. Neigh.	Near. Neigh.	Voting Near. Neigh.
BankAccount	100	100	100	100
apache-extract	99	98	98	98
mysql-extract	100	99	88	88

Table 1: Results of 100 iteration leave-one-out cross-validation

executions to a new execution, and classifies the new run with the class of the majority of the n nearest neighbors. Each of these two nearest neighbor classifiers can be applied to feature vectors in the space of either Read Sets, or Write Sets, or, though it was not evaluated in this work due to time constraints, some feature which combines Read Sets and Write Sets.

4 Experimental Setup and Evaluation

In this section I discuss the experimental setup and methodology used to evaluate Buggaboo, and present preliminary results for three sets of experiments performed using Buggaboo.

4.1 Experimental Setup

In order to evaluate the ability of Bugaboo to correctly determine the bugginess of program executions, first, a method to collect memory operation traces was developed using the PIN binary instrumentation framework [5]. Then, this tool was used to collect memory access traces from several bug-kernel benchmarks taken from prior research [4]. These bug-kernels are buggy sections of code which were extracted from full applications, or which were constructed based on othe prior literature. Their purpose is to provide an environment in which a buggy section of code executes, and then periodically and nondeterministically the bug actually manifests itself. In this work, I used three such kernels: *mysql-extract* which is an extracted version of a bug in the logging subsystem of MySQL, *apache-extract* which is an extracted version of a bug in the logging subsystem of the Apache httpd web server, and *BankAccount* which is a simple bank account implementation in which there is occasionally a lost update to the bank account due to an atomicity violation.

Once I have collected a memory access trace for a run, a series of scripts computes the Write Set for each Read, and the Read Set for each Write. I computed these sets for 100 executions each of the benchmark programs. Using these sets, I created the two versions of the nearest neighbor classifier described above for each feature set (Write and Read Set features separately). I then performed a 100 iteration leave-one-out cross validation of the classifiers which were created to determine the ability of a classifier created using 99 labeled data points to determine the class of the remaining one data point.

4.2 Experimental Results

Using even just a simple nearest neighbor classifier, and also using a voting nearest neighbor classifier, Bugaboo was able to very reliably determine the bugginess of the single run left out in each run of the leave-one-out cross-validation for all applications evaluated, for both Read Set features and Write Set features. Table 1 shows these results in detail.

Generally, the Buggaboo is able to determine the bugginess of an execution very reliably, for most experiments, almost 100% of the time. There tends to be little difference between the results obtained using Read Set features and using Write Set features, and also there tends to be little difference between the results of the nearest neighbor and the voting nearest neighbor classifier. Perhaps surprisingly, in some cases the voting nearest neighbor classifier performs worse than, or the same as the voting nearest neighbor classifier. The margin is small, and examining the instances provides little insight into why this is, except that certain instances produce Write Sets for certain “important” Reads (or Read Sets for Writes) which end up leading to incorrect classification. I did not have time to do this, but interesting future work would be to attempt to map the significance of certain reads or writes back to programs space to determine why the lead to incorrect classification in certain cases.

One notably unsuccessful attempt at classification is the case of mysql-extract, when using Read Set features. The reason for this is that the bug in this case occurs when a read in one thread reads data from a write in another thread which should not have been able to be read. As a result, the size of the Write sets in this execution differed, but the size of the read sets did not, because each write still produced data which was read by just one read. This is the reason that the performance when using Write sets is successful, but that using Read Sets is not.

5 Conclusions and Future Work

Buggaboo is a technique by which executions of concurrent programs which may contain subtle bugs can be classified as buggy or not buggy. This classification is done using the computed Read and Write Sets for each Write and Read in the execution, and then creating a Nearest Neighbor classifier which determines the bugginess of executions.

There are many interesting directions in which this work could next be taken. First and foremost, a more thorough and inclusive evaluation of the technique presented herein should be performed. In particular, full programs should be evaluated, not just bug-kernels (though bug-kernels do provide a nice test-bed for a proof of concept). Next, it is an extremely interesting and useful step to take to determine why executions classified as buggy were classified as buggy. This might occur by some evaluation of differences between Write and Read sets which differ between buggy and non-buggy runs. Surely, this would increase the value of this work tremendously, as once developers learned executions were buggy, they could be led to the point in the code which is involved in the bug. Finally, it is interesting to consider how this technique could, with hardware support be used “online”, to constantly be checking for potential bugs, and perhaps even be coupled with a bug avoidance technique similar to that in [4] to provide resiliance to bugs as well as simply detection of bugs.

6 A Tour of the Software And How To Run Buggaboo

Documentation for Buggaboo is, lamentably mostly in the source code. A few key things make up the Buggaboo code. First is the binary instrumentation profiling tool. This tool requires an installation of PIN to work. The provided Makefile will build the pintool “make tool” is run. Then, running pin with the -mt (multithreaded) switch, the tool as the argument to the -t option, and the desired binary, traces can be collected. These traces are, by default, saved to ”nd.data”. For a given trace, the script nondetermometer.pl can be used to digest it and to produce the read or write sets. To determine which will be produced, some lines in the script need to be commented out (change references to Inverse Data Flow Graph to Data Flow Graph). Ok, so I know that whoever is grading this probably won’t end up having PIN installed, nor would they want to dig through my ugly source, but in the interest of completeness, I thought I’d include it. I have some pre-digested datasets included. In the files which match the command line regex ”*Instances”, I have Read Set and Write Set version of 100 instances for each of the benchmark programs. For one instances, the script “find_Nearest_Neighbor.pl” can be used to find the nearest neighbor amongst the remaining instances. As a convenience, CrossValidate.pl has also been included. To run this and see some real results (the ones included in the report), use the command line “perl CrossValidate.pl XInstance/All XInstance/Buggy XInstance/NonBuggy N”. In this command line, XInstance is the instance directory for the instance you’re interested in, and N is the number of voters which should be used in the voting majority nearest neighbor classifier. The results of CrossValidate.pl have all been included in the paper for your convenience, but naturally I’ve included every line of source used in this project for you review.

References

[1] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 142–153, Washington, DC, USA, 1998. IEEE Computer Society.

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

[3] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[4] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, June 2008.

[5] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.