Invite Research Project

Fall 2009- Final Report

Waseem Ilahi – wki2001@columbia.edu

Professor Gail Kaiser

Chris Murphy

Abstract

There is a limit to how many tests one can perform on a software before deploying it. This limitation makes it possible for some of the bugs in the code to go unchecked. Thus, the system/software is released with those bugs still present and unknown to user and developer. Our research is based on solving this problem in the manner that even after the software has been released to the general public; there can be a way to keep testing it continuously, without disrupting the original order of execution. This functionality is achieved by forking off the original path and performing the test in the child process and the continuing the original path. Doing so makes sure the original path is not mutated. Some virtualization method can be used to make sure the original process isn't affected. Invite is the "system" that implements this continuous testing approach. We call it InVivo testing approach. The original executed these tests whenever it came across them, thus executing many of them various times. This is inefficient, so we need to find a way to avoid this problem. We can specify a certain "state" that a process has, right before it executes a test. These states can be hashed into a data structure that allows very fast iteration through it. So, when a process reaches at the beginning of a test, it can look into the structure and find out if the "test" has been executed before. If it has, then we can simply ignore the test and move on, otherwise we execute the test.

The "state" in this case consists on the variables that the test will depend upon. These variables can be found by the static analysis of the test code. I was given the job to implement this "analyzer/preprocessor" that would figure out these variables and return them.

This paper provides an introduction of the whole project and then the description of the code that I wrote. In this paper, I will try to include a lot of description for my work in Fall 09, so the future project students can use this paper to continue from where I left off.

1. Introduction

The traditional testing is never able to find all the bugs in the program. PSL has been trying to investigate “InVivo Testing Approach” to solve this problem. This method performs the tests continuously, even after the deployment of the software. The functionality that we had so far, for this suite was that it forked the original process and then the test was run on this child thread. This process leads the system to perform a test multiple times, especially in distributed systems. This is a huge performance hit. To avoid this, Mr. Chris Murphy came up with the idea, that we create some sort of a “state” and hash it, so when the process reaches the execution point of the test, it checks the hash and if the test has been performed before, it simply continues without performing the test, otherwise, it performs the test and adds this “state” in the hash for the future. This is a two step process, first is to figure out what the “state” is and next is to use some “data structure” to “hash” these states in it, for fast access. We decided that the “state” will be the values of all the known external variables that a test depends upon, at the beginning of the test. My responsibility this semester was to find these variables. I had to implement a preprocessor that takes a “test/function” and gives back the variables, this test depends upon.

This paper describes the code that I wrote that implements this functionality. Section 2 gives a little insight into the “big picture” of the whole project. Section 3 talks about what I had to implement. Section 4 explains the various parts of the implementation of my code. Section 5 gives the list of files and a little description of what each contains. Section 6 is about future directions. Section 7 will talk about lessons learned and finally Section 8 is references.

2. Big Picture

The whole project, including what Moses is doing……

3. Preprocessor

What I had to do……

4. Checkpoint Function Call and Integration with InVite { ckpt ( ) }

After I was finished with the catching up and understanding the zap & dejaview source code, I started programming for the function call: int ckpt(void);

The function ckpt( ) takes no arguments but returns three kinds of values, negative, zero and positive values.

Return Values

This function returns negative value for a failed attempt at the checkpoint. The failure can occur on many different positions in the execution. Since, the check pointing in the ZAP heavily relies on files, if a file fails to open then we have to return -1 showing that the function did not complete the task in its entirety. Among those files are the counter file, for the checkpoint file name and the log file, named tbd.log. Also, if the do_zap_pod( ) function returns a negative value ckpt( ) also returns a negative value. Therefore, unless everything gets completed successfully, we get a negative return value from ckpt( ).

ckpt( ) returns 0 if the process is the parent process. This means that the current process is the original process that called the ckpt( ) function.

ckpt( ) return a positive (non zero) value, corresponding to the child's Id, if the returning value is the child process. For example, if the session is check pointed and then resumed three times, the function will return 1, 2, 3 respectively for each of the child process and zero for the parent process. The checkpoint in zap is basically a file with all the information in it.

do_zap_pod( )

The do_zap_pod( ) function gets three arguments. The one in the middle is very simple. It is ZAP_ZAP_POD, the command that tells ZAP to take a checkpoint. It is used in the lower level function, to tell them what the caller has asked to do. The other two arguments are structures of the type zap_t and zap_args respectively. Section 6 below tells us where they are defined in the ZAP source, so the user can have a look at them. Now the first of the two structures, zap_t, gets filled by the do_zap_open( ) function, which I will talk about a lot in Section 6. Apparently, this function opens the "zap" device to read/write to it. The second structure is very important, as it holds the flags and the file descriptor for the checkpoint file.

Now, in ckpt( ) I first set the zap_args' flags as ZAP_POD_CONT | ZAP_MIG_INPOD, which means to let the session keep running after the checkpoint has been taken and the checkpoint is being taken from inside the pod. Other than that, I set a few variables necessary to take the complete checkpoint. Then after the name for the checkpoint has been established, I open it and assign its file descriptor to the fd field of zap_args. I get the zname and zid fields of zap_args from the name and zid fields of the zinfo structure (see Section 6 for information). Now we have all three arguments ready, so we pass them along to do_zap_pod( ), which does its thing and returns the result to us.

Logfile and checkpoint Name

When we run the command **dejaview listckpt session_name**, the dejaview manager calls the getAvailableCheckoints(self) function inside the session.py file. The said function calls another function called __concat_tbd_checkpoints( ) and then it returns the list of checkpoints on the current session. Now the __concat_tbd_checkpoints( ) function looks into the /dejaview/fs/cache/sess.”id”/checkpoints/ directory and sees if it has a tbd.log file. This file is suppose to be a log file, that has the checkpoint information listed in it. __concat_tbd_checkpoints( ) then reads the file entry by entry and then matches each entry to a predefined expression. And stores the information, it extracts from the entries it reads from the file, inside a list. Then it assigns these values to the appropriate variable. Now it checks if the entry that it just read already exists in the system, if it does then it continues, else it assigns all the values to a dictionary entry named cpentry. One of the keys of this dictionary is the checkpoint file name. Now after setting the keys and the values, it appends the cpentry to the list of checkpoints for the session. This continues for all the entries in the tbd.log file. Then the __concat_tbd_checkpoints( ) saves the session udates and returns.

Because of the above stated facts we need to populate the tbd.log file after do_zap_pod( ) returns a positive (non zero) value. This is because we need to update the log file only for the parent process. So after the the do_zap_pod( ) returns, we check, if the return value was zero then we update the log file, with the information we get from zinfo_t as well as the time the checkpoint was taken and the checkpoint number that we get from the counter file that I mentioned earlier.

Now, the __concat_tbd_checkpoints( ) function, when enters the checkpoint name in cpentry, it names it tbd. “#”, so we have to make sure that we name the checkpoint file the same way and that our tbd.log entry is consistant with the checkpoint file’s name.

Child ID

This brings us to the final important part inside the ckpt( ) function, which is also related to the get_childID( ) function. When do_zap_pod( ) returns, we check to see if the value was negative and if were, we simple return that value, but if it was not then we check to see if it were zero, and if it is zero then we know that the current process is the child process, so we return the return value of the get_childID( ) function.

The get_childID( ) function opens up the file named “ChildIdentifier” and reads the value stored in it. This value is the child Id of the current process (child). The value gets updated when a session is branched by dejaview branch command. So the unique child id is relative to the time the child was branched and the time it was resumed. You can branch two children and resume them in the opposite order, but they one branched first will have the id =1 and the second will have 2. More detailed explanation; we consult the Section 7 of this report, where I talk more about session.py, which holds the code more this feature.

Integration with InVite

For integration of this function ckpt( ) with Invite, I merely had to replace the fork command inside the precompile.pl with ckpt( ). For more details on this also, look at the Section 7, paragraph three, where I talk about the changes made to InVite code.

The code is commented itself and very self explanatory, what is important to mention here is the steps to take for the whole process. I do have various scripts written that test the function call in various ways, but I will write a step by step process in testing the function. You can see the usage for the scripts in the Section number 7.

P – T – O →

Demo

These are the steps I take to test the function: (make sure you are a super user for it to work)

I will number the commands that I run:

i) sudo su /* become the super user. */

ii) cd /home/CKPT/ /* move to the directory of the source code. */

iii) make /* makes the target "testckpt" → calls the ckpt( ) to take checkpoint */

iv) touch empty /* Script for creating an empty pod. */

v) chmod 755 empty /* change its permissions. */

vi) dejaview new –s demo1 –nilfs –size=1024 /* create a new session. */

vii) dejaview start –start=/home/CKPT/empty sess1 /* start the session with the empty script.*/

viii) cp testckpt /zap/p0/root/tmp/testckpt /* copy the test to the pod. */

ix) dejaview add_proc –u root sess1 /tmp/testckpt /* add the process to the pod "run it"*/

x) dejaview listckpt sess1 /* see the list of the checkpoints on the session. */

xi) dejaview branch –c 1 sess1 sess2 /* branch off at the point of the checkpoint. */

xii) dejaview resume sess2 /* resume the check pointed session. */

xiii) more /zap/p0/root/tmp/Parent_Output /* see the parent's output. */

xiv) more /zap/p1/root/tmp/Child_Output /* see the child's output. */

You can repeat the xi and xii steps to create as many children as your want. Make sure you keep track of their names or ids to know which is which, and you can check their child ids by repeating the step number xiv but with the appropriate pod number.

To further understand the functionality, read the source code, provided with this final report and also look at the Section 7 below. The material in section 7 is also available as a README with the source.

5. The Source Code:

My goal for the semester was to write a "preprocessor" that takes a "c" file reads it and for each function defined in that file, if finds out the variables that this function depends upon. This functionality is very important for the main project that we were working on, which is to figure out the impact on the performance on the Invite system, if it knows whether it has seen the current state or not and if it has, then it skips the test. My code comes into play, when we talk about a state. We decided to differentiate among the states, based on the values for each variable, the current test depends upon. My code finds out these variables and after the integration, will pass along these variables to the next "preprocessor that creates the resulting code for testing.

I divided the code in multiple files with each file housing the code called on the similar levels. I am listing the files and the functions in each with a little explanation for each.

The List of Files and Directories:

a. The CSInvite directory

CSInvite.h → It is the main header file for all the components. It has the "includes" from the c library. It also contains the macros for the lengths and numbers. Then it defines the data structures for all the variables used by the program. And at the end, it has the prototypes of all the functions used by the program.

CSInvite.c → The main file. It calls all the other functions and does controls the program flow. It contains the main function.

CSInvite_help.c → This file holds the three string helper functions. These functions are called by other functions to do string manipulation.

CSInvite_set_input.c → This file holds the functions that are called by the main function once after the invocation of the program. It sets up the global variables and constants, and the functions with their definitions.

CSInvite_set_func.c → This file sets up the scopes of the statements, plus the re-declared variables for each function, the user asks for.

CSInvite_set_vars.c → This file contains the functions that actually fill the structure that represents the variables the function depends upon. It also contains the function to handle calls to other functions and recursion.

Makefile → makes CSInvite.exe (can be used in Windows or Linux)

b. The Test Directory

This directory contains the test files. These tests were used to check the functionality of the program.

test.h, test.c, test1.c and test2.c → These are the basic test files, test1.c and test2.c have one function definition and test.c calls these functions.

single_test.c → This file contains seven function definitions, these functions test various features the program provides.

Makefile → makes the test.exe, also checks single_test.c for problems.

NOTE: "compile" the test code before using it with the "CSInvite", for the program to work correctly.

6. Future Direction

We have the preprocessor to get the variables the function depends upon; we also have a way of checking whether the state was visited by the test before or not. Next step is integrating both features and taking care of few limitations they each have. We might also want to try and include some external variables to expand the "definition" of state we have. Right now it is pretty limited; we will want to expand that to include external variables as well. We did achieve good numbers while evaluating our technique, so there is definitely some potential in going in this direction.

7. Lessons Learned

First, I had to researh whether there was something already present, like what I had to do. During the extensive research on the subject, I came across a lot of good technologies/tools, one of them being "Frama-C". I learned a lot about the static analyzers. However, none of them did what we were looking for. Therefore, I had to implement the "preprocessor" myself. Since, it is something totally new, I didn't have many references to

consult. Working with “c” gets complicated, but I enjoy working with it. I was able to under the whole syntax of c programs much better than before. I used many new functions provided by c standard library. It was fun, but it was very complex and took a lot of my time.

Nevertheless, I am pleased that I was given the opportunity to work with this project. I give my thanks to my project advisors Mr. Chris Murphy and Dr. Gail Kaiser, for providing me the opportunity to work on such an interesting subject, my contribution made me feel proud and satisfied. I am planning to go far the PhD. I hope to work on similar projects in the future if possible.

8. References:

Google Search Engine [Multiple Forums]

Frama-C Documentation. [http://frama-c.cea.fr/]

Waseem Ilahi (myself) [Summer 09 Final Report]