MQP GFP 1001

Test Selection and Prioritization for EMC

A Major Qualifying Project Report:

Submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

By:

Eric Prouty

Tin Myo Win

Xianjing Hu

Date: January 8th, 2012

Approved:

Professor Gary F. Pollice, Major Advisor

1. Test selection
2. EMC
3. Proof of Concept

Table of Contents

List of Illustrations

Abstract

Currently, testers at EMC manually select test cases to ensure fixes to bugs and features work as expected and do not affect previously functional code areas. This project investigates automating the test selection process by providing a repeatable procedure for selecting tests, thus improving efficiency and accuracy.

After a comprehensive background research, we designed a solution using gcov, a code coverage tool, in conjunction with GCC to produce code coverage information. The information is then analyzed to show the relationship between source code files and test cases, suggesting tests to be run. We also explored other possibilities, including analyzing EMC’s RCSA spreadsheet, modifying gcov for embedded systems, and developing a profiler for EMC’s embedded system. Our recommendations are based upon the analysis of the pros and cons of each solution.

Acknowledgements

We are very grateful for the sponsorship and cooperation from the managers and developers at EMC. We would like to give special thanks to James Lavallee who initiated this project, to Patrick Cullinane who organized meetings at EMC and constantly provided valuable information, resources, and advices, and to Padraig Bradley and James Grimaldi who contributed very useful suggestions and comments to our project from different perspectives. We would also like to express our sincere thanks to Professor Gary Pollice, whose advice guided us all the way through this project from beginning to end.

1. Introduction

This project develops a proof-of-concept and inspires further investigation into the exploration of systems that perform automated test selection and prioritization. The problem this is intended to solve is selecting the proper tests to exercise changes that have been made to a code base since the previous tested version. Given a code base and test repository, when a change is made to that code base how do you determine which tests contained within the repository to run?

This is an important problem that faces many companies as it directly influences the company's ability to provide reliable software. Even with a comprehensive testing repository, there is often not enough time to perform a full regression test in time for critical patch releases. This is the situation faced by EMC who must deliver patches to customers who face critical data losses or outages. The software must be fixed and delivered quickly in order to maintain good customer relations. When a patch, or EPACK as it is known within the organization, is prepared for a customer it must be tested to ensure that the problem has been fixed and that it has not introduced further complications.

This could be accomplished by performing a full regression test which ensures that all known problems have been accounted for and that the system is still running up to the current specifications. This approach would cause a delay in the patch reaching a customer, which in the case of data storage can mean sustained outages and a loss of customer satisfaction. Another approach would be to run a random assortment of tests over the given amount of time before the patch needs to be delivered to the customer. This approach takes into consideration the time constraint but does not necessarily test the changed area. While throwing as many tests as possible at the problem in the given time at least ensures that most of the systems functionality is still working, it does not guarantee confidence in the patch being a successful solution.

Our solution recommends the creation of a system that ties the test repository to the code base under test. This provides a clear link between the changed code and the tests that would most likely exercise it, and improves confidence in the patch solution to the customer’s problem. In order to come to this recommendation our team performed research upon the current practices being used and the associated articles detailing similar work. Using this information a proof of concept using the code coverage tool gcov was developed and tested. This proved that it was possible to create a concrete relation between a test and the code under test on an operating system. We then investigated how to introduce such a tool into the current workflow at EMC. This also prompted further study into other options.

Our team has identified three possible solutions to the problem that could be implemented by EMC, the remainder of this document will report upon our methodology and how we came to such conclusions. It will also detail the three solutions that we have recommended and explain their use. The final portion of this document will then go into details regarding future work that could be completed and our recommendations for how this proof of concept could be moved forward into a working implementation.

2. Background

2.1 About EMC Corporation

“EMC Corporation (EMC) is a global leader in enabling businesses and service providers to transform their operations and deliver Information Technology as a service.” **(EMC at a**

Glance) Being well-known for its product lines from enterprise storage arrays to security products, EMC is accelerating to cloud computing, helping businesses to store, manage, protect and analyze their massive quantities of data in a more flexible and cost-efficient way. EMC customers are diverse, not only in size, from startups to the Fortune Global 500, but also in industry and geographic location. EMC works with organizations across the world, in every industry, in the public and private sectors.

EMC history starts with the name of the company, EMC. “E” and “M” stands for the initials of the two founders, former Intel executive, Richard Egan and his roommate Roger Marino. Founded in 1979, EMC started as a supplier of add-on memory to the growing minicomputer industry. In 1990, EMC entered into the mainframe-storage market, 80% of which was taken by International Business Machines Corporation (IBM) at that time. By 1995, EMC became the market leader with 41% market share due to its sophisticated and well-served data storage systems. From 1995 to 2000, the company made swift transformations, increasing the complexity of management and production line. EMC started to aim at different market segments offering multiple product lines rather than carrying a single storage system. During the five-year period, EMC became a global player with presence in all corners of the globe, instead of a US-based vendor, selling storage services to virtually all types of IT buyers. (Rao, 2003)

Today, EMC provides a wide range of product categories that address storage, information security, data warehousing, enterprise content management and cloud computing. With nearly 50,000 employees, EMC is a serving more than 400,000 customers and partners across more than 80 different counties. Moreover, it is managing five data centers, with storage of 10 petabytes (1 petabytes = 2^{52} bytes) of information. Starting from 2004, EMC began its journey into Cloud infrastructure to continue as a leader in the market.

2.2 Current EMC Systems and Tools

The following chart illustrates the relationship between the tools and platforms EMC is using to perform its development and testing cycle, and how the customers, developers, and testers communicate with each other. For a detailed introduction of EPACK, OPT, CAT and Syren, refer to the glossary at the end of this document.

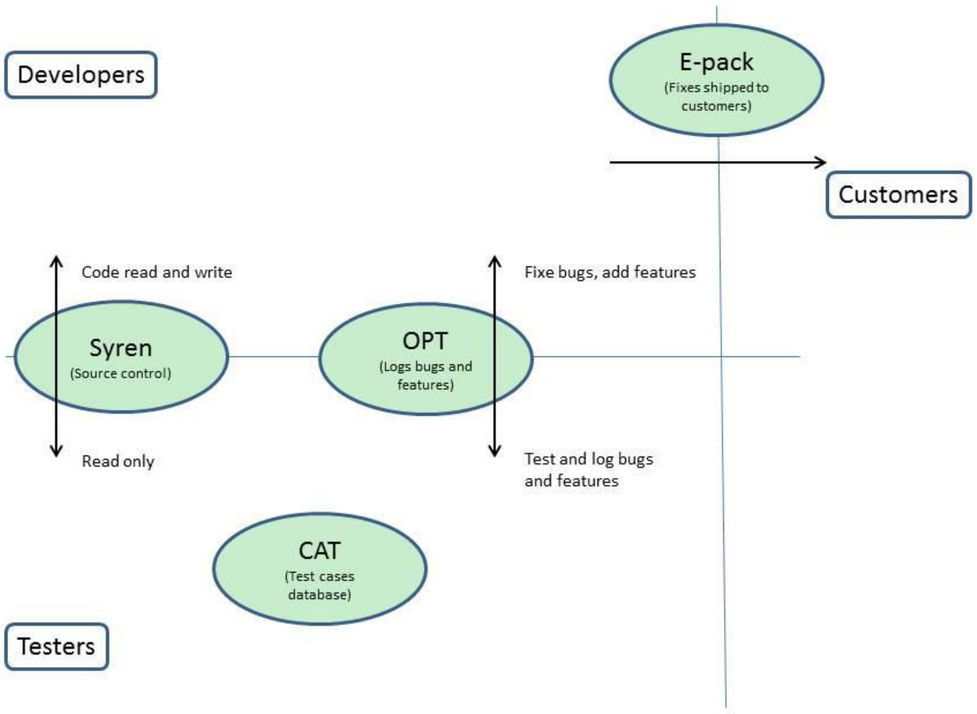


Figure 1. Relationship between the tools used by EMC

An EPACK is a group of fixes, as explain in the glossary section. The customers report problems, and the developers generate fixes for the problems. After the problems are solved and tested, the developers put the required fixes into an EPACK and then ship it to the customers.

Testers use CAT as their test cases database. Tests are drawn from the database to test specific functionalities; if a test does not exist to cover the functionality then new test cases are created and added into CAT.

Developers and testers communicate through the problem tracking platform OPT. Testers log the bugs and new features into OPT. Developers take over the bugs and new features description, write code to fix the bugs and add new functionalities as required, then return the cases to the testers; the testers re-test the problems, close the case if problems solved, and put it into OPT again if not.

Sometimes testers also need to look into Syren to see the check-ins made by the developers. Syren is the version control system used by EMC.

2.3 The Workflow

The following workflow diagram illustrates the overall workflow from a customer reporting a bug through to the EPACK containing fixes being shipped back to the customer. The workflow can be broken into the following steps:

1. Customer issue is recreated and logged into OPT.
2. In OPT, the customer issue enters into debug and fix cycle. The developers work on fixing the issue, assign it back to testers when it’s done, and testers retest the issue.
3. When testing a certain fix, the testers look into the CAT test database to see whether there are existing test cases to test the fix. If not, new test cases will be created and added.
4. The tester tests the fix against the test cases selected from CAT. If the fix passes the test, the code will be checked in. If the fix fails the test, the tester will re-assign it to the developers, and the cycle continues until the problem is solved.
5. After all fixes pass all test cases needed to test them, the fixes will be put into an EPACK.
6. To make sure that the selected fixes resolved the issue raised by the customers, and the

fixes do not break anything else, the testers will then need to select tests to test the entire EPACK.

7. When the testing in Step 6 is finished and passed, the EPACK will be ready to be shipped to the customers.

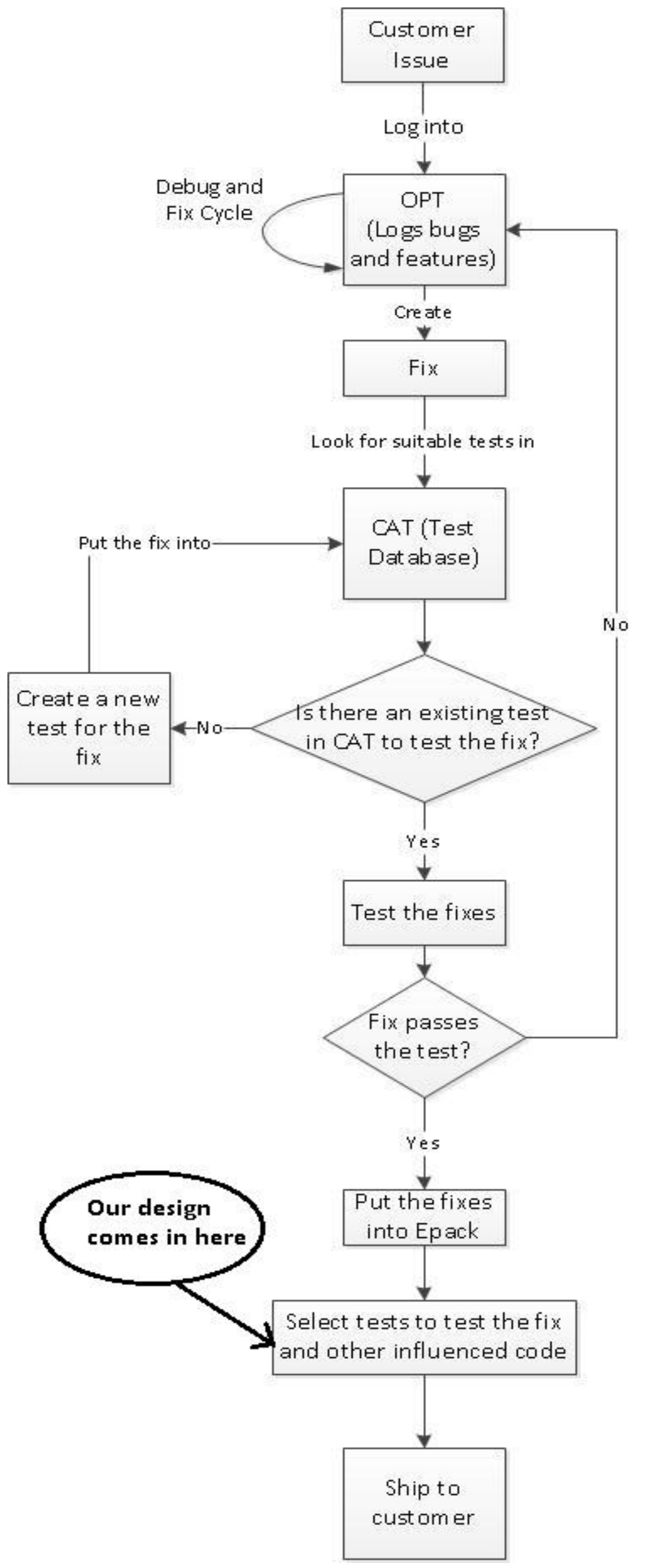


Figure 2. Existing development cycle at EMC showing where the test selection process occurs

3. Problem Statement

3.1 A Better Solution

Our design falls into Step 6 in the workflow. Specifically, when the fixes are ready in the EPACK, what test cases the testers should choose to test the fix and other influenced code. Currently, the test cases are selected manually from the test cases database. This process is time-consuming and inaccurate because it's done completely according to the judgment of the testers. It remains for us to design a system to automate this process, i.e. to find a way to select tests associated with a fix (including tests that test against the fix and against the influenced code). We also need to prioritize the tests so that under tight time constraint, the most effective test cases can be carried out first. "Effectiveness" has two meanings: tests that expose the most problems or tests that exercise most code.

3.2 Problem Statement

The problem presented to our group is the necessity of selecting the proper tests to exercise changes that have been made to the code base in a given EPACK. These tests must be selected from a repository of tests while ensuring that it provides a solid coverage of the affected code. This ensures that any changes being made interact correctly with each other and do not affect the rest of the systems functionality. Along with ensuring that the system is functioning properly, these tests must also be selected on a given time constraints, as deadlines are provided for the release of each fix. This problem affects EMC and impacts them by forcing additional time to be used performing free run testing which may miss problem areas. A successful solution to this problem would be a system that generates an organized list of tests based on the given criteria.

This would provide a system that is systematic and requires minimal effort by the user thus removing the necessity of employees to perform testing based upon their judgment, and introducing a method of selection using concrete heuristics.

4. Methodology

4.1 Approach

To design a Test Prioritization and Selection system for EMC, a certain code coverage system is needed for EMC's code base. During the first few weeks of the project, team members went to EMC weekly to collect information related to EMC's systems and development environment and to design a code coverage system especially for EMC's system. The team finally agreed to use a code-based approach, utilizing the test prioritization techniques as mentioned below. Since EMC's code base is mainly written in the C language, research on existing open-source C code coverage tools such as GCT (General Coverage Tool), gcov, and gprof took place. Eventually gcov, a test coverage tool to be used in conjunction with the GCC compiler, was chosen to design a code coverage system for EMC's code base. Initially, the gcov tool was tested on an open-source C project with existing test cases, since team members did not have access to EMC's source code. Then, the team instrumented the Ubuntu Linux Kernel with gcov to test our final design on an operating system.

4.2 Existing solutions

The main purpose of the project is to produce proof-of-concepts for various approaches to the test prioritization and selection problem rather than to implement a final product. Therefore, research on various similar approaches took place in the initial stage of the project. The approaches researched included source code based test prioritization methods and a binary matching approach.

4.2.1 Source code based approach

The main concept of a code-based approach is to use source code of the system to select proper test cases based on the latest changes to the source code. Code-based test prioritization techniques are dependent on information relating the tests of a test suite to various elements of a system's code. For instance, a particular code-based approach can utilize information about the number of lines of code or the name of functions executed by a test. Then the information about executed code elements is recorded for each test in a database. Different types of test prioritization heuristics are then applied to analyze the information recorded and prioritize the tests.

For test selection and prioritization techniques, there are several test prioritization heuristics available. Two example test prioritization techniques are 'total statement coverage prioritization', which works at statement level and 'total function coverage prioritization', which works at functional level. Both types of heuristic work by prioritizing test cases in terms of the total number of statements or functions they cover. For instance, if we use 'total statement coverage prioritization' heuristic, the test that covers the largest number of statements in the source code will be run first.

Each of the two heuristics mentioned above have their own benefits and drawbacks. 'Statement coverage prioritization' is more comprehensive in term of performance but more expensive in terms of cost of capturing the trace information, compared to 'function coverage prioritization’. Since the number of statements in a program is typically much larger than the number of functions in a program, the process of collecting statement-level traces is more expensive and intrusive than the process of collecting function level traces. However, experiments in granularity effects show that most techniques with finer granularity (i.e. statement coverage in this case) have a higher weighted average of the percentage of faults detected over the life of a test suite. (Sebastian Elbaum, 2002)

The foregoing test prioritization techniques represent a wide spectrum of approaches, varying along several dimensions. The first dimension is granularity, which we previously mentioned, in terms of function-level and statement-level. Granularity affects the relative costs of techniques in terms of computation and storage, but also affects the relative effectiveness of those techniques. The second dimension involves whether or not a test prioritization technique relies on feedback about coverage attained so far in testing to focus on elements not yet covered. For instance, ‘additional function coverage prioritization’ technique, which is similar to ‘total function coverage prioritization’ technique, incorporates feedback into total function coverage prioritization, prioritizing test cases (greedily) according to the total number of additional functions they cover. The third dimension involves whether or not a technique utilizes information from modified version of a program. For example, techniques that depend on fault-exposing-potential estimation attempt to factor in the potential effects of modifications in general. Several empirical experiments have compared various techniques. (Gregg Rothermal, 2001)

4.2.2 Binary matching approach:

Echelon, a system developed at Carnegie-Mellon University and implemented at Microsoft, utilizes a binary matching system to compute the differences at a basic block granularity between versions of the program in binary form. As shown in the following figure, the system takes two versions, a new image and an old image, of the program in binary form. Along with the test coverage information of previous executions, detailing which tests cover which parts of the program. The system then analyses the impacted blocks likely to be covered by existing tests. Finally, the system outputs a prioritized list of tests.

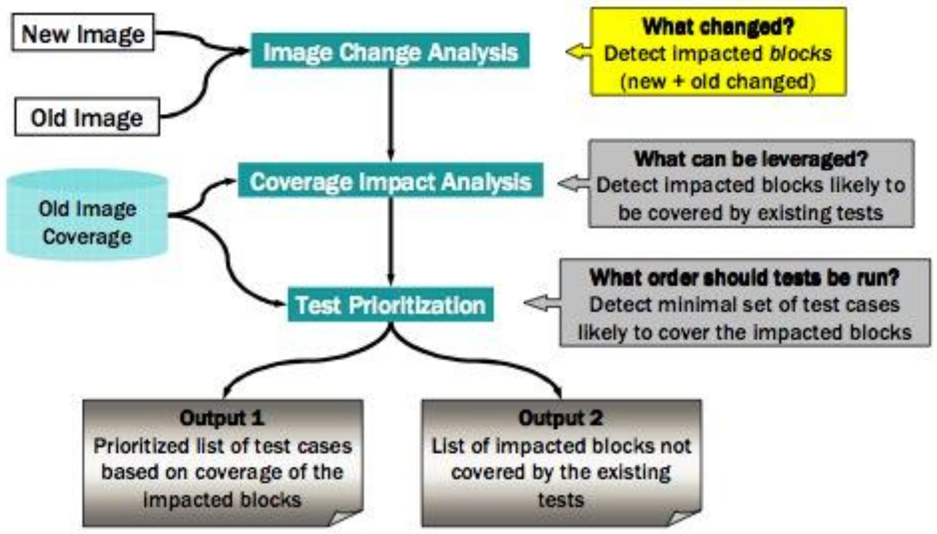

New Image
Old Image
Image Change Analysis
What changed? Detect impacted *blocks* (new + old changed)
Old Image Coverage
Coverage Impact Analysis
What can be leveraged? Detect impacted blocks likely to be covered by existing tests
Test Prioritization
What order should tests be run? Detect minimal set of test cases likely to cover the impacted blocks
Output 1 Prioritized list of test cases based on coverage of the impacted blocks
Output 2 List of impacted blocks not covered by the existing tests

Figure 3. Echelon's work flow

4.3 The gcov tool used by our proof-of-concept

In each stage of the project, we created a proof-of-concept using a tool or technology openly available. After researching existing code coverage tools, team members agreed to use gcov, which is an open source C code coverage tool under GNU tool set, to design a C code coverage system.

4.3.1 gcov

Overview:

Using the gcov coverage tool, the following basic performance statistics can be recorded:

1. How often each line of code is executed?
2. What lines of code are executed?
3. How much computing time each section of code uses?

The code base should be compiled without optimization in order to use gcov on it because the optimization, by combining some lines of code into one function, causes inaccurate results. Gcov accumulates statistics by line so it works best with a programming style that places only one statement on each line.

How gcov works:

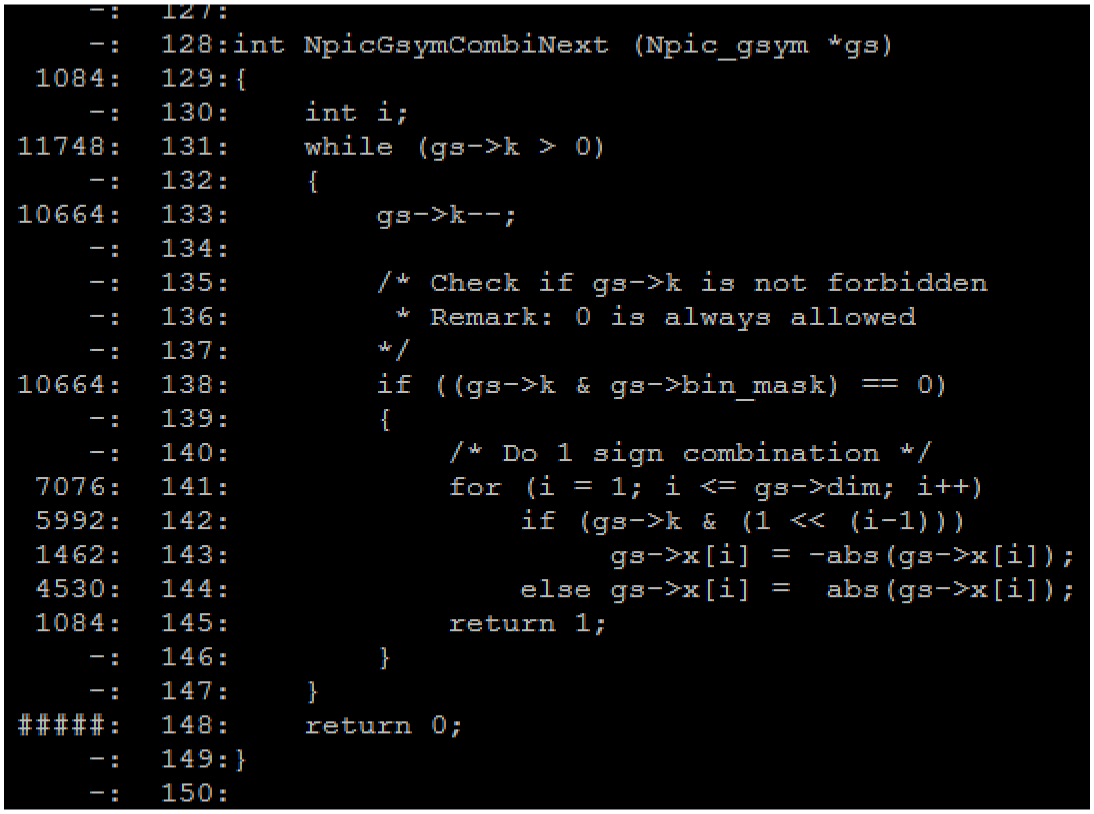
The gcov framework has three phases as explained below:

1. Compilation phase ("gcc -O0 -o hello -fprofile-arcs -ftest-coverage sourcename.c")
   a. This instructs the GCC compiler to add the necessary instrumentation to the object that gcov requires.
2. Data collection and extraction phase ("./sourcename" is the collecting binary)
3. Reporting phase ("gcov -a sourcename.c")

During the compilation phase, a .gcno file is generated. The file contains information to reconstruct the basic block graphs and assign source line numbers to blocks. In the second phase, the .gcda file is generated when a program containing object files built with the GCC -fprofile-arcs option is executed. A separate .gcda file is created for each object file compiled with this option that is executed. It contains arc transition counts, and some summary information. During the reporting phase, the .gcda and .gcno files are read out by gcov. Then the basic block data is calculated from arc data and written into a human-readable report (sourcename.c.gcov).

Gcov outputs:

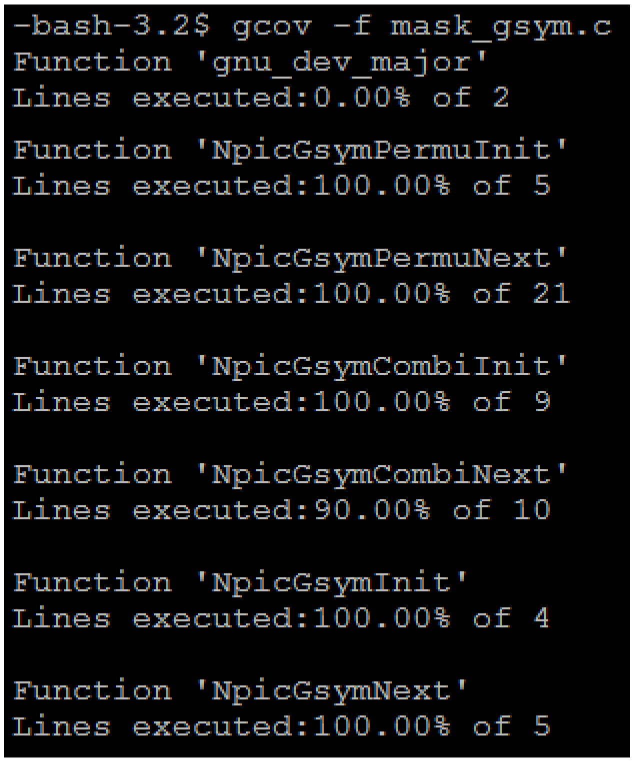
The gcov tool was tested on an open source C project. The figure below shows a screen shot of gcov's output with the number of times a line is executed in a source file. The '-' symbol denotes lines containing no code, and '#####' denotes an unexecuted lines.



```
    -:  127:
    -:  128:int NpicGsymCombiNext (Npic_gsym *gs)
 1084:  129:{
    -:  130:    int i;
11748:  131:    while (gs->k > 0)
    -:  132:    {
10664:  133:        gs->k--;
    -:  134:
    -:  135:        /* Check if gs->k is not forbidden
    -:  136:         * Remark: 0 is always allowed
    -:  137:        */
10664:  138:        if ((gs->k & gs->bin_mask) == 0)
    -:  139:        {
    -:  140:            /* Do 1 sign combination */
 7076:  141:            for (i = 1; i <= gs->dim; i++)
 5992:  142:                if (gs->k & (1 << (i-1)))
 1462:  143:                     gs->x[i] = -abs(gs->x[i]);
 4530:  144:                else gs->x[i] =  abs(gs->x[i]);
 1084:  145:            return 1;
    -:  146:        }
    -:  147:    }
#####:  148:    return 0;
    -:  149:}
    -:  150:
```

Figure 4: Gcov output with count information

The format, or the type, of the gcov output can also be changed by including optional flags when gcov is run on the source file. For example, using the –f flag, we can output a function level summary for each source file as shown in the following figure. The summary includes the percentages of the function executed by a test.



```
-bash-3.2$ gcov -f mask_gsym.c
Function 'gnu_dev_major'
Lines executed:0.00% of 2

Function 'NpicGsymPermuInit'
Lines executed:100.00% of 5

Function 'NpicGsymPermuNext'
Lines executed:100.00% of 21

Function 'NpicGsymCombiInit'
Lines executed:100.00% of 9

Function 'NpicGsymCombiNext'
Lines executed:90.00% of 10

Function 'NpicGsymInit'
Lines executed:100.00% of 4

Function 'NpicGsymNext'
Lines executed:100.00% of 5
```

Figure 5: Gcov output with function summary for a source file

4.4 Designing a C code coverage system using gcov

A C code coverage system was designed using the gcov tool with the following steps:

- Instrumented the operating system by setting compiler flags to utilize gcov tool
- Created a script to parse source code for function information.
- Created script to run tests and parse coverage information.
- Stored in database and correlated using SQL queries.
- The design was tested on the Linux operating system.

The following figure further explains the workflow of the design:

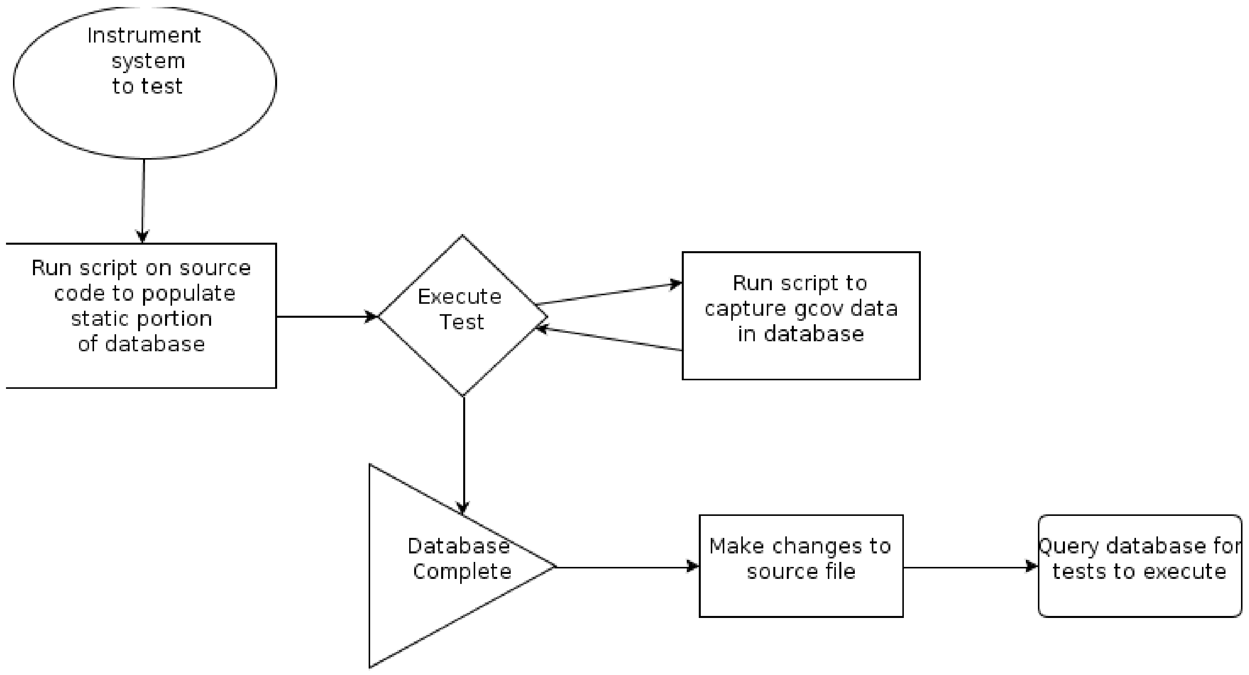
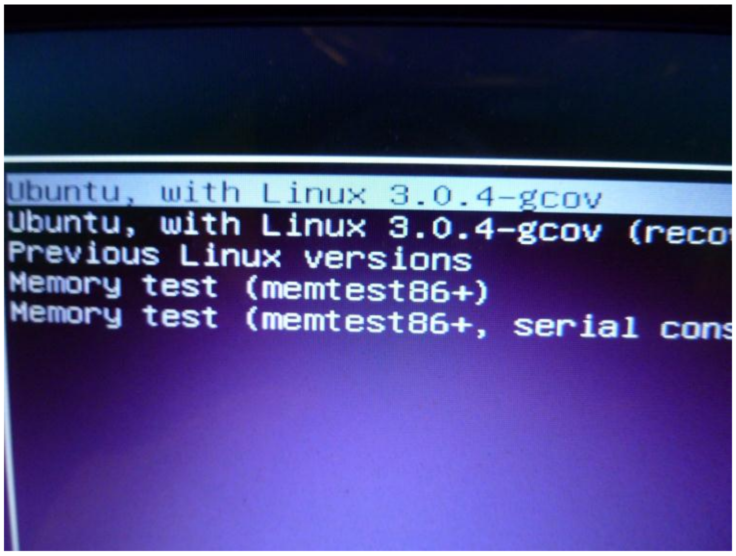

Instrument system to test
Run script on source code to populate static portion of database
Execute Test
Run script to capture gcov data in database
Database Complete
Make changes to source file
Query database for tests to execute

Figure 6: Design of C code coverage system using gcov tool

4.4.1 Instrumenting an operating system

Before compilation, the operating system is configured first by including the gcov coverage profiling option. This can be done using the available configuration tools available with the Linux kernel. The option to include coverage information is a standard part of the current kernel. The following figure shows a screen shot of the grub boot loader about to launch a version of Linux 3.0.4 after it had been instrumented.

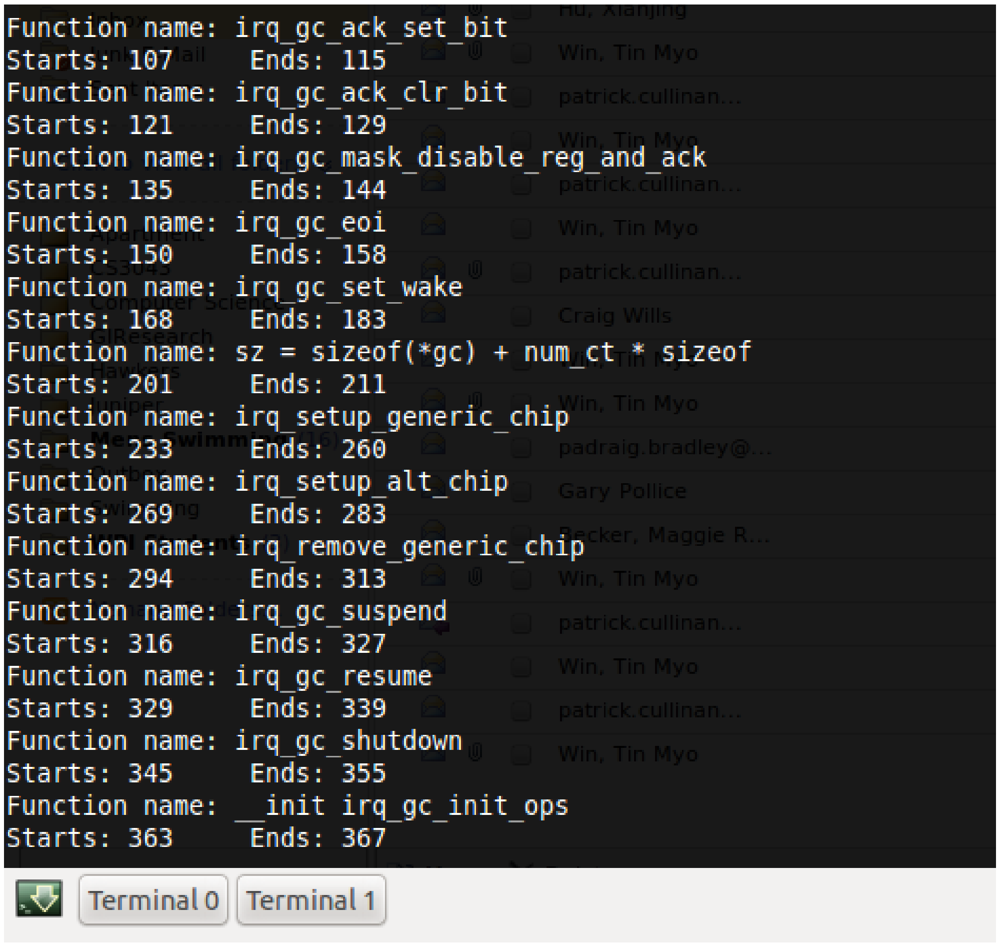


```
Ubuntu, with Linux 3.0.4-gcov
Ubuntu, with Linux 3.0.4-gcov (reco
Previous Linux versions
Memory test (memtest86+)
Memory test (memtest86+, serial cons
```

Figure 7: Instrumented Linux 3.0.4 with gcov

4.4.2 Source code parser

A source code parser was created using Perl to get function information from source files. As shown in the figure below, the Perl script returns the line number of the start-line and end-line of each function in a source file. Then that function information is stored in a database, which is later used to correlate between a changed line and which function it is part of by querying the database.

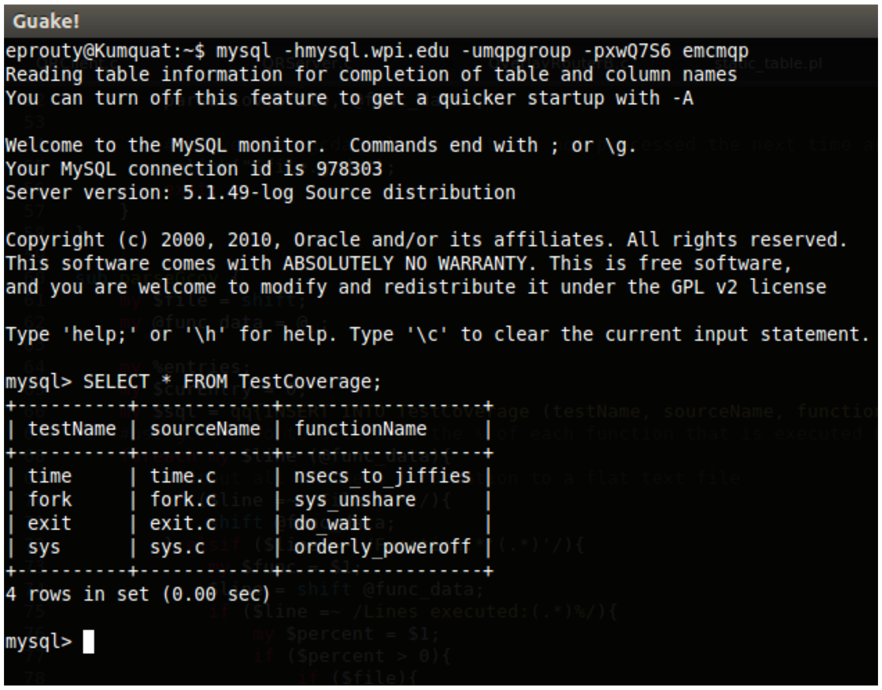


```
Function name: irq_gc_ack_set_bit
Starts: 107     Ends: 115
Function name: irq_gc_ack_clr_bit
Starts: 121     Ends: 129
Function name: irq_gc_mask_disable_reg_and_ack
Starts: 135     Ends: 144
Function name: irq_gc_eoi
Starts: 150     Ends: 158
Function name: irq_gc_set_wake
Starts: 168     Ends: 183
Function name: sz = sizeof(*gc) + num_ct * sizeof
Starts: 201     Ends: 211
Function name: irq_setup_generic_chip
Starts: 233     Ends: 260
Function name: irq_setup_alt_chip
Starts: 269     Ends: 283
Function name: irq_remove_generic_chip
Starts: 294     Ends: 313
Function name: irq_gc_suspend
Starts: 316     Ends: 327
Function name: irq_gc_resume
Starts: 329     Ends: 339
Function name: irq_gc_shutdown
Starts: 345     Ends: 355
Function name: __init irq_gc_init_ops
Starts: 363     Ends: 367
```

Figure 8: Source code parser that gives function information in a source file

4.4.3 Test coverage parser

For test coverage information, the gcov tool is used. It is noted here again that when a test file is executed, gcov creates a .gcda (gcov data files) for each source file the test executed. A Perl script was created to run gcov on test files and then parse the output. Finally the test coverage information was stored in a database. The figure below shows the database which records the names of source files and functions a test case executed:



```
eprouty@Kumquat:~$ mysql -hmysql.wpi.edu -umqpgroup -pxwQ7S6 emcmqp
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 978303
Server version: 5.1.49-log Source distribution

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SELECT * FROM TestCoverage;
+----------+------------+-----------------+
| testName | sourceName | functionName    |
+----------+------------+-----------------+
| time     | time.c     | nsecs_to_jiffies |
| fork     | fork.c     | sys_unshare     |
| exit     | exit.c     | do_wait         |
| sys      | sys.c      | orderly_poweroff |
+----------+------------+-----------------+
4 rows in set (0.00 sec)

mysql> 
```

Figure 9: A sample test coverage database

4.4.4 Test selection

After we have the source and test information stored in a database, appropriate test cases can be queried based on changes made to the code base. We can query the name of functions modified using function information from the source database if we know the line numbers modified from source control system. Then from the test coverage database, a set of test cases that executed the modified functions can be queried.

4.5 Suggested Improvements of the Current Approach

4.5.1 Modify gcov for Embedded Systems

In an environment with a UNIX based file system, gcov produces data files for each source file executed by a test. These data files contain count information such as the number of times a line is executed. Since an embedded system similar to the one used by EMC does not have such a file system, gcov must be modified so that rather than writing out those data files, the count information will be extracted from the target embedded system to a host system, that satisfies those requirements.

Gcov goes through three phases in the process of profiling as follows:

1. compilation phase (“gcc -O0 -o hello -fprofile-arcs -ftest-coverage hello.c”)
2. data collection and extraction phase (“./hello” is the collecting binary)
3. reporting phase (“gcov -a hello.c”)

Gcov needs to be modified in the second phase, which is Data Collection and Extraction phase, where the count information is collected and extracted into gcov data files. On the embedded system, it would be necessary to stop the system manually and make a call to the gcov_exit function. This would then write the data into memory and transfer the data through the network to a host system. For further information a possible approach has been described for an open source, real-time operating system for embedded applications called eCos. (eCos® and eCosPro® Reference Manual)

4.5.2 Use test history to develop a low cost solution

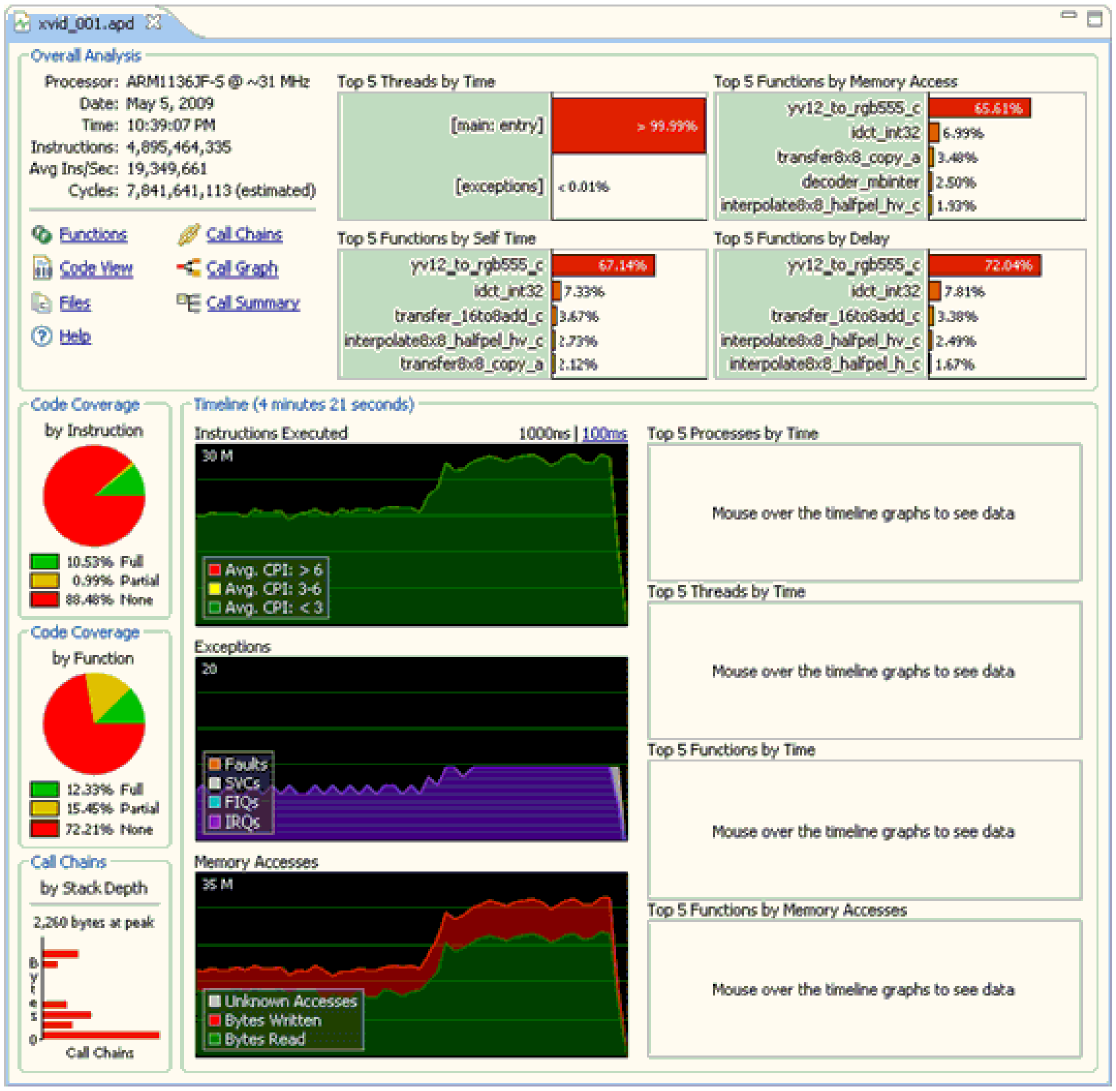
The test coverage information that is being gathered by the gcov tool could be approximated by using information pertaining to the history of tests that have been run. For EMC this comes in the form of their RCSA spreadsheets. These spreadsheets contain information about which tests have passed and failed. As these tests are then fixed they record information regarding their entry into OPT and eventual fixes.

By assuming a correlation between a previously failed test, and the lines of code contained within a fix that was created in order for this test to pass, a database the same as the one previously described could be created. This solution is entirely dependent of the availability of good history regarding this subject. With the proper history and tools in place to mine such information this could be used as a temporary solution while a more elegant one is implemented, or as a low-cost permanent solution, that would become more effective as history becomes more comprehensive over time.

4.5.3 Develop a profiler for EMC's embedded system

Instead of analyzing the RCSA spreadsheet or running the tests with gcov to get coverage information, there exists a more effective, but also more time-consuming way to get coverage information. The approach is to directly build a profiler for EMC's embedded system. This is a feasible approach because during our research, we found that such kind of profiler already exists for the ARM embedded systems.

The ARM profiler provides analysis of the performance of code running on embedded system, while it's processing a real work load for as long as needed. It provides user-friendly interfaces with results displayed in an intuitive way. The following picture shows an analysis summary:



xvid_001.apd

Overall Analysis

Processor: ARM1136JF-S @ ~31 MHz
Date: May 5, 2009
Time: 10:39:07 PM
Instructions: 4,895,464,335
Avg Ins/Sec: 19,349,661
Cycles: 7,841,641,113 (estimated)

Functions · Call Chains · Code View · Call Graph · Files · Call Summary · Help

Top 5 Threads by Time

[main: entry]	> 99.99%
[exceptions]	< 0.01%

Top 5 Functions by Memory Access

yv12_to_rgb555_c	65.61%
idct_int32	6.99%
transfer8x8_copy_a	3.48%
decoder_mbinter	2.50%
interpolate8x8_halfpel_hv_c	1.93%

Top 5 Functions by Self Time

yv12_to_rgb555_c	67.14%
idct_int32	7.33%
transfer_16to8add_c	3.67%
interpolate8x8_halfpel_hv_c	2.73%
transfer8x8_copy_a	2.12%

Top 5 Functions by Delay

yv12_to_rgb555_c	72.04%
idct_int32	7.81%
transfer_16to8add_c	3.38%
interpolate8x8_halfpel_hv_c	2.49%
interpolate8x8_halfpel_h_c	1.67%

Code Coverage by Instruction

🟩	10.53%	Full
🟨	0.99%	Partial
🟥	88.48%	None

Code Coverage by Function

🟩	12.33%	Full
🟨	15.45%	Partial
🟥	72.21%	None

Call Chains by Stack Depth

2,260 bytes at peak

Timeline (4 minutes 21 seconds)

Instructions Executed 1000ms | 100ms

30 M

Avg. CPI: > 6
Avg. CPI: 3-6
Avg. CPI: < 3

Exceptions

20

Faults
SVCs
FIQs
IRQs

Memory Accesses

35 M

Unknown Accesses
Bytes Written
Bytes Read

Top 5 Processes by Time

Mouse over the timeline graphs to see data

Top 5 Threads by Time

Mouse over the timeline graphs to see data

Top 5 Functions by Time

Mouse over the timeline graphs to see data

Top 5 Functions by Memory Accesses

Mouse over the timeline graphs to see data

Figure 10. Analysis Summary (ARM Ltd.)

Figure 10 shows the top five time-consuming functions with different prioritization benchmarks at the top, and pie charts of code coverage measured by instruction and by function with percentages are shown at the left of the analysis. At the top left, the summary also provides links to other more detailed reports, including the complete list of statistics for every function organized by execution, code coverage and time spent; detailed information on the source code and its derived assembly code, annotated with performance and coverage information. It also generates views that help explore the dynamic call graph and caller-callee relationships.

We suggest that EMC analyze the statistics given by the detailed views, in the same way that we analyze the statistics given by gcov. EMC should end up with a script that takes in the function-level analysis file from the profiler and generate a list of tests to be performed.

The way that the ARM profiler transfers data from the embedded hardware (or hardware simulation) to the user interface on a PC is shown in the following picture:

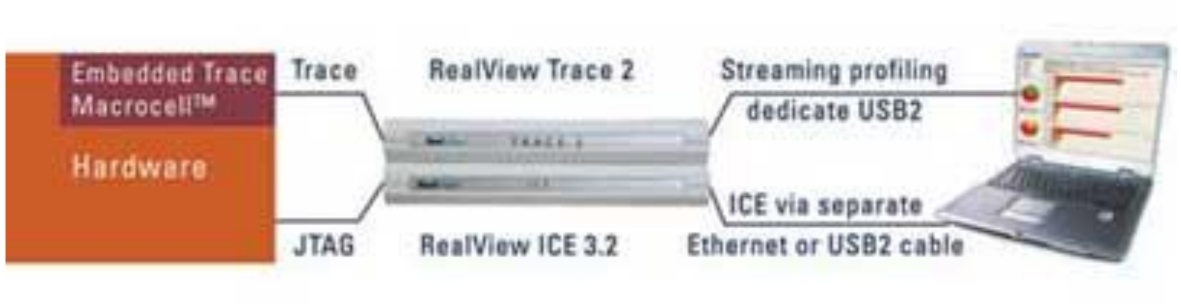

Embedded Trace Macrocell™
Hardware
Trace
JTAG
RealView Trace 2
RealView ICE 3.2
Streaming profiling dedicate USB2
ICE via separate Ethernet or USB2 cable

Figure 11. ARM Profiler Data Transfer Flow (ARM Limited.)

The ARM profiler works by observing code on target hardware using RealView ICE and RealView Trace 2 or by testing code against a Real-Time System Model (RTSM). The former runs on the actual embedded hardware while the latter runs on a simulated model. Data on the embedded hardware is accessed through a controller on the ARM core using the JTAG interface. The RealView ICE product has a run control unit that connects to the target board over a JTAG interface and to a PC using either USB or Ethernet. If EMC is to develop a similar profiler, the data transfer solution adopted by ARM should be of reference value.

5. Results and Analysis

This project produced three recommendations that EMC could follow to solve the problem of selecting and prioritizing tests, along with a proof of concept detailing our primary recommendation. These three solutions run the gamut from the lowest cost, to most effective, with our primary recommendation as a compromise between the two. The lowest cost solution utilizes the available history of tests in order to produce a correlation between failed tests and the changes that were necessary to produce a successful run. This implies that the changed code is in direct relation to the test, and therefore the test can be assumed to cover that piece of code.

The second solution is a compromise between cost and effectiveness. It uses the gcov tool mentioned above. This allows for an implementation utilizing existing tools to reduce cost, while still allowing for comprehensive test coverage. The final recommendation is the creation of a profiler designed for EMC's Symetrix system; this would require the most engineering but would provide the finest grained run-time information possible. This would result in the most accurate representation and would allow more complex heuristics to be used in the test selection process.

5.1 Lowest cost solution

This solution utilizes the fact that EMC maintains records of every test that has failed, and in turn the fix that was then associated with it in a later update. As mentioned above this allows correlation between that piece of code and the test that had failed. This allows for a coverage data base to be built by analyzing the entire available history. The workflow for how this process should occur is included below.

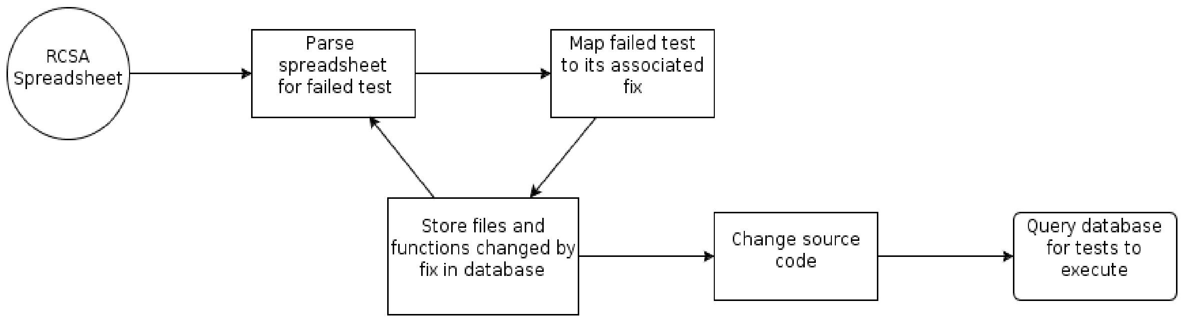

RCSA Spreadsheet
Parse spreadsheet for failed test
Map failed test to its associated fix
Store files and functions changed by fix in database
Change source code
Query database for tests to execute

Figure 12. Workflow suggestion for using RCSA spreadsheet

5.2 Most cost-effective solution

Modifying gcov tool for an embedded system would be recommended as the most cost-effective solution. Using the existing gcov tool, a certain C code coverage system can be implemented at line level accuracy. Then, the line numbers in source file being executed by a test can be recorded. Therefore, by modifying the existing gcov tool, a reliable C code coverage system for EMC system could be implemented. For this project, code coverage information at functional level has been demonstrated by testing the tool in Linux kernel.

5.3 Most informative solution

The solution that will provide the most complete set of information is to create a profiler specifically for EMC's Symetrix system. This would provide accurate run time information about every command that is executed for a specific test. The major drawback of this solution however is the amount of engineering required to implement it. This system would need to be built from scratch and would require a large number on man-hours to complete.

6. Conclusions

This project has provided several feasible solutions to the stated problem. Though there is a lot of work to be done before a fully functional system can be implemented this project has provided the base work to begin the selection and further development of a solution. Throughout the entire process, from gathering information to implementing the prototype, the education and several project experiences received at WPI helped team members greatly. First of all is the communications skill team members learned during their school projects. Understanding EMC's complex systems in a short amount of time required team members to ask intelligent and concise questions and then interpret the answers to design the system collaboratively among team members and project advisor. Secondly team members' familiarity with the Linux operating system helped complete the project since the project included compiling, building, and testing of C projects. Last but not least, working knowledge of the Perl scripting language and relational databases was big help to create a proof-of-concept for the system designed.

Appendix

gcov parser code

```
#!/usr/bin/perl

use Getopt::Long;
use File::Find;
use strict;
use warnings;
use DBI;

my $dbh;
my ($del_statement, $delete_sql);
#get the root of the soure directory that should be parsed

my $sourceBase;
my $testBase;

my $curTest;

setupDatabase();

GetOptions("source=s" => \$sourceBase, "test=s" => \$testBase);
open OUTPUT, ">>output.txt";

find(\&testWanted, $testBase);

sub setupDatabase {
    $dbh = DBI-
>connect("DBI:mysql:database=emcmqp;host=mysql.wpi.edu",
"mqpgroup", "xwQ7S6") or die ("Couldn't connect to DB\n");
    $dbh->{AutoCommit} = 0;    # Use transacted processing
    $dbh->{RaiseError} = 1;    # die on processing errors
    $delete_sql = qq{DELETE from Static};
    $del_statement = $dbh->prepare($delete_sql);
    $del_statement->execute();
}

sub testWanted {
     if (-x $_ and !($_ eq ".")){
          #execute the test
          #system("./$_");
        print OUTPUT "Test: $_\n";
        $curTest = $_;
        #search through the source for the .gcda files to see
which source files were used
        find(\&sourceWanted, $sourceBase);
     }
}

sub sourceWanted {
    #search for .gcda le
    if ($_ =~ /(.+)\.gcda/){
        my $file = $1;
        #when found execute gcov on the associated c file
        my @func_data = `gcov -f $file.c`;
        #parse the .gcov file into a useable format
        parseGcov($file, @func_data);

        #remove the .gcda file so that its not processed the
next time around
        unlink("$file.gcda");
        exit;
    }
}

sub parseGcov {
    my $file = shift;
    my @func_data = @_;

    my %entries;
    my $curEntry = 0;
    my $sql = qq{INSERT INTO TestCoverage (testName, sourceName,
functionName) VALUES (?, ?, ?)};
    #use gcov info to determine the % of each function that is
executed by the test
    foreach my $line (@func_data){
        #output all of the % information to a flat text file
        if ($line =~ /File '.*'/){
            shift @func_data;
        } elsif ($line =~ /Function.*'(.*)'/){
            my $func = $1;
            $line = shift @func_data;
            if ($line =~ /Lines executed:(.*)%/){
                my $percent = $1;
                if ($percent > 0){
                    if ($file){
                        $entries{$curEntry++} = ["$file.c",
$func];
```

```
                        undef $file;
                    }
                }
            }
        }
    }

    eval {
        my $statement = $dbh->prepare($sql);
        while (my ($id, $dataRef) = each %entries) {
            my @data = @$dataRef;
            $statement->execute($curTest, $data[0], $data[1]);
        }
    }
}
```

gcov parser output example in human-readable format

```
Test: test-mask
      Source File: files_gz.c
            Function: NpicReadGZ_Open -> 44.44%
            Function: NpicWriteGZ_Open -> 44.44%
      Source File: mask_comp.c
            Function: NpicCompGeneMask -> 97.16%
      Source File: props.c
            Function: NpicPropKeyIsValid -> 100.00%
            Function: NpicFreeProps -> 100.00%
            Function: NpicSupprProp -> 100.00%
            Function: NpicAddProp -> 50.00%
      Source File: mask_creat.c
            Function: NpicExpandMask -> 100.00%
            Function: NpicMaskInsert_6l -> 100.00%
            Function: NpicMaskInsert_5d -> 38.46%
            Function: NpicMaskInsert_4l -> 91.67%
            Function: NpicMaskInsert_3l -> 100.00%
            Function: NpicMaskInsert_2d -> 91.67%
            Function: NpicPrintMask -> 38.46%
            Function: NpicCopyMask -> 90.91%
            Function: NpicCreateMaskDP -> 90.91%
      Source File: files_nmask.c
            Function: NpicNMASKWriteFile -> 98.48%
            Function: NpicWriteNMASK -> 90.91%
            Function: NpicReadMedialAxisMask -> 56.25%
            Function: NpicWriteMask -> 98.48%
      Source File: mask_gsym.c
            Function: NpicGsymPermuNext -> 100.00%
            Function: NpicGsymCombiNext -> 100.00%
            Function: NpicGsymInit -> 100.00%
```

Source parser code

```
#!/usr/bin/perl

use strict;
use warnings;
use Getopt::Long;
use File::Find;
use DBI;

my $dbh;
my ($del_statement, $delete_sql);
#get the root of the soure directory that should be parsed
my $source_root;
GetOptions("source=s" => \$source_root);

setupDatabase();

find(\&sourceWanted, $source_root);

sub setupDatabase {
    $dbh = DBI-
>connect("DBI:mysql:database=emcmqp;host=mysql.wpi.edu",
"mqpgroup", "xwQ7S6") or die ("Couldn't connect to DB\n");
    $dbh->{AutoCommit} = 0;    # Use transacted processing
    $dbh->{RaiseError} = 1;    # die on processing errors
    $delete_sql = qq{DELETE from Static};
    $del_statement = $dbh->prepare($delete_sql);
    $del_statement->execute();
}

sub sourceWanted {
    if ($_ =~ /(.*)(?:\.c)$/){
        my $file = $1;
        parse("$file.c");
    }
}

sub parse {
    my $file = shift;
    open SOURCE, "$file";

    my ($curLine, $startLine, $endLine, $curEntry, $funcName);
    my %entries;
    $curLine = 1; #line numbers start at 1...
    my $openBraces = undef;
    my $sql = qq{INSERT INTO Static (sourceName, functionName,
startLine, endLine) VALUES (?, ?, ?, ?)};

    while (my $line = <SOURCE>){
        #fucntion definiton... therefore the start of a new
function
        if (!defined($startLine) && $line =~
/(?:(?:int|void|char|short|long|float|double|signed|unsigned|boo
l|complex|imaginary|struct|union|const|restrict|volatile)[\s\*])
+\*?(.*)\s*\(+.*/){
            $startLine = $curLine;
            $funcName = $1;
            print ("Function name: $funcName\n");
        }
        if (defined($startLine)){
            if ($line =~ /{/){
                $openBraces++;
            }
            if ($line =~ /}/){
                $openBraces--;
            }
            if (defined($openBraces) && $openBraces == 0){
                $endLine = $curLine;
                print("Starts: " . $startLine . "\tEnds: " .
$endLine . "\n");
                $entries{++$curEntry} = [$funcName, $startLine,
$endLine];
                undef($startLine);
                undef($endLine);
                undef($openBraces);
            }
        }
        $curLine++;
    }

    eval {
        my $statement = $dbh->prepare($sql);
        while (my ($id, $dataRef) = each %entries) {
            my @data = @$dataRef;
            $statement->execute($file, $data[0], $data[1],
$data[2]);
        }
    }
}
```

Glossary

CAT: Test case database called Common Automation Tool, in which a user can search test cases by ‘Test Name’, ‘Primary Tester’ and ‘Product Information’.

EPACK: A fix/patch which a development team would ship to an EMC customer. Testers choose a set of test cases from the CAT system to test against the EPACK.

gcov: C code coverage profiler, which is run in concert with the GCC compiler.

OPT: The Online Problem Tracking (OPT) system in which testers can list problems/bugs found. The following fields are filled while listing the issue in the tool: Status, Severity, Product Information, Problem Type, Problem Submitter, Responsible Engineer and Responsible Tester.

RCSA: A history report listing the records of regression testing, including which tests have passed of failed, and if necessary their associated record in the OPT system.

SYREN: A source-control tool. Users can use this tool to see code difference between two versions of source code along with other functionalities often provided by version control systems.

Bibliography

eCos® and eCosPro® Reference Manual. (n.d.). Retrieved 12 22, 2011, from Test Coverage: http://www.ecoscentric.com/ecospro/doc/html/ref/gcov.html

EMC at a Glance. (n.d.). Retrieved 11 07, 2011, from EMC: http://www.emc.com/about/emc-at-glance/corporate-profile/index.htm

Gregg Rothermal, R. H. (2001). Prioritizing Test Cases for Regression Testing. *IEEE Transaction on Software Engineering* , 929-947.

(2003). Leading with Knowledge. In M. Rao, *Leading with Knowledge* (pp. 93-99). Tata McGraw-Hill.

Sebastian Elbaum, A. G. (2002). Test Case Prioritization:A Family of Empirical Studies. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 159-181.

ARM Limited. "ARM Profiler Non-Intrusive Performance Analysis ." *ARM.com.*Web. 23 Dec 2011. <http://www.arm.com/products/tools/software-tools/rvds/arm-profiler.php>.

ARM Limited. "ARM Profiler User Guide Version 2.1, 5.3.Analysis Elements." *ARM Infocenter*.Web. 23 Dec 2011. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0414d/index.html>.