Coverage-Based Testing on Embedded Systems

X. Wu^, J. Jenny Li*, D. Weiss* and Y. Lee^

**Avaya Labs Research, 233 Mt. Airy Rd., Basking Ridge, NJ 07920*
{jjli, weiss}@research.avayalabs.com
^Department of Computer Science and Engineering, Arizona State University,
699 South Mill Avenue Tempe, AZ 85281
{xwu22, yhlee}@asu.edu

Abstract

One major issue of code coverage testing is the overhead imposed by program instrumentation, which inserts probes into the program to monitor its execution. In real-time systems, the overhead may alter the program execution behavior or impact its performance due to its strict requirement on timing. Coverage testing is even harder on embedded systems because of their critical and limited memory and CPU resources. This paper describes a case study of a coverage-based testing method for embedded system software focusing on minimizing instrumentation overhead. We ported a code coverage-based test tool to an in-house embedded system, IP phone. In our initial experiments, we found that this tool didn't affect the behavior of the program under test.

1. Introduction

Code coverage testing technology attempts to quantify progress of program testing. Some academic study on small programs showed that software reliability improves as testing coverage increases [1]. Even though large industrial trials and theoretical study of correlation between coverage and defect detection are still required, intuitively, low testing coverage is not acceptable. No developer would deliver software with only single digit testing coverage.

We developed an automatic coverage testing tool suite to facilitate the usage of coverage testing. eXVantage, short for eXtreme Visual-Aid Novel Testing and Generation, which aims to help developer to improve testing productivity. It also helps project managers in keeping track of each developer's progress. Figure 1 shows the workflow of our coverage testing tool suite.

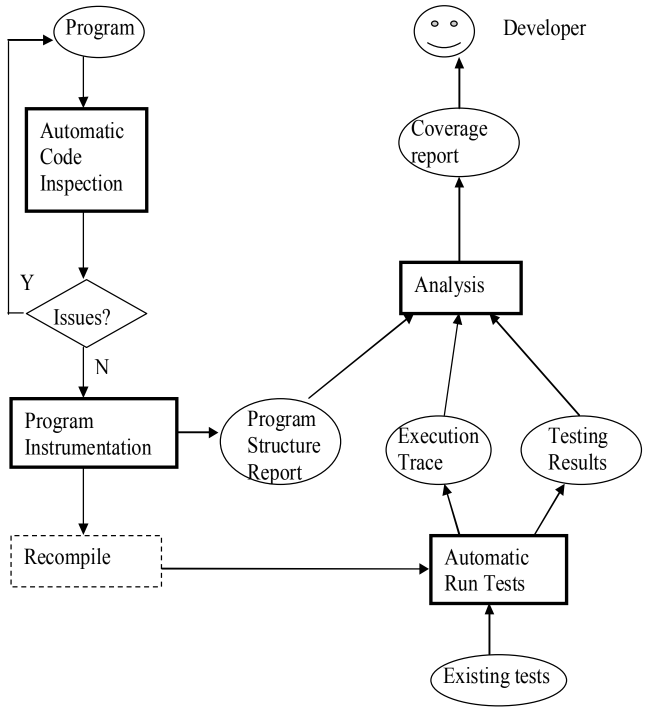

Figure 1. Workflow of Automatic Unit Testing Framework

The tool suite includes four major components: 1) automatic code inspector, 2) program instrumentation module which also includes program structure recovery function, 3) automatic test execution platform, and 4) analysis module. Each component corresponds to one tool in the accompanying tool suite. The component shown in the dashed box, recompile, may not be necessary if the bytecode or object code is used in the program structure analysis and instrumentation. It is not part of the tool suite component because it uses the original program compiler without any modification or using any new tool.

The entire framework can be invoked by a script including five steps: code inspection, instrumentation, compiling, test execution and analysis. The input to the script is the directory of the programs to be tested and analyzed and the output is testing coverage reports. The workflow works as follows:

1) Each program must first go through an automatic code inspector using our style-checker tool. If issues or problems are discovered by the checker, they are highlighted on the program and should be fixed by developers immediately.
2) After the program is clean of static problems, it must go through a program instrumenter that adds hooks to the program for monitoring its execution. In the meantime, the program structure can be recovered from the program.
3) After the program is instrumented, the program can then be recompiled (if the program is the source code) or directly sent to test execution (if the program is bytecode or object code). We use a pattern matching approach to select part or all of the test cases for execution.
4) Testing results and program structures are used by the analysis module to generate various reports. One typical report is program testing code coverage report, which helps user keep track of testing progress. If some parts of code are not covered, a priority report will be generated to show which part of code is to be tested next.

The reports generated by the framework are stored in a data base. Additional tool is included to depict historic trend on program changes and code coverage. Such information is useful for project planning and process control.

One important issue of coverage testing is the overhead of monitoring program execution, in particular, for real-time embedded systems. The probes inserted to the original program at step 2) can cause test execution overhead at step 3). The step 3) is the only step that is platform dependent. Our work on porting the tool suite to embedded systems focuses on step 3) in the tool suite's workflow.

The overhead comes from two sources: CPU requirement for probe execution and memory requirement for storing execution traces. The focus of this paper is on the second issue. Details of the problem and its solutions will be described later.

The rest of the paper is organized as follows. Section two defines in details the porting requirements and memory challenges. Section three describes our solution to the problem. Section four presents our experimental results and section five concludes that our solution enables embedded system code coverage testing.

2. Problem description

The target system of our case study is an embedded system, with MIPS architecture. Due to the real time feature of the software, porting a code coverage testing tool to such systems requires to minimize the overhead caused by the code coverage instrumented probes and monitored trace data storage and transmission.

The target system is based on the VxWorks Real Time Operating System (RTOS). VxWorks is one of the most successful real time operating systems, which provides multitasking, intertask communication, and hardware interrupts handling as the three most important features.

VxWorks has very good performance in terms of both timing and space. Its raw context switch time could be as low as 4 micro seconds, and the VxWorks image we used for this case study has a size of about 4 KB.

Another issue needs to be considered in porting to an embedded system is the memory mechanism. In the original version of eXVantage, shared memory is the way for trace memory buffer maintenance. The VxWorks image in the target system does not have the shared memory component included. Furthermore, VxWorks itself uses a flat memory model, which means every task could access to every single valid memory address without any boundary, including concurrent tasks.

eXVantage was originally developed for Linux platform, and each process has a pointer stored in the shared memory pointing to another piece of memory allocated before the program starts. In this process specific memory, the header starts with 9 bytes, including information like process ID, buffer size, and key value (which will be used during instrumentation). The rest of the memory is the place where trace data are temporarily stored for later retrieval through network connections.

Figure 2 illustrates this memory issue in a diagram. It shows that each process has its own block of shared memory. This block of shared memory doesn't need to be shared among the original processes. But it needs to be shared with the new eXVantage process that handles trace transmission. Since the trace data memories of various original processes are not shared among each other, there is no need for a memory locking mechanism.

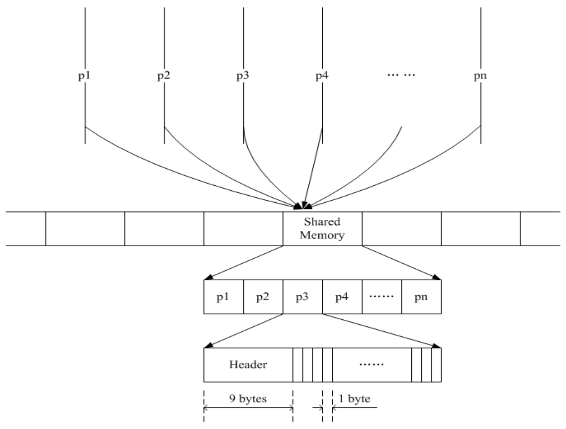

Figure 2. Share memory issue of trace data

The eXVantage VxWorks run time system is composed of three parts: PC Host, the embedded system, and Ethernet connections. Tornado, the VxWorks Integrated Development Environment (IDE) runs on the PC Host, The embedded software application program runs on the embedded system. The PC Host and the embedded system are connected via a serial link. Both of them also have Ethernet connections, as shown in Figure 3.

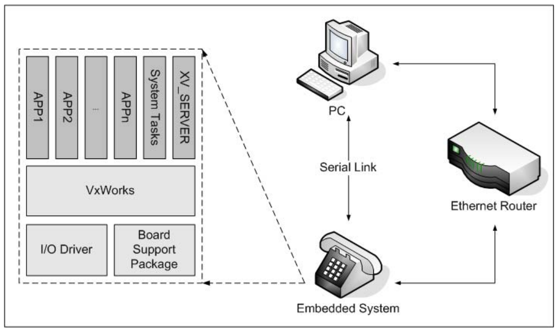

Figure 3. The hardware setting

The program that runs on the embedded system - the telephone in this system - could be divided into three layers. The first layer contains user-defined application tasks $APP_1 – APP_n$, VxWorks system tasks, with the addition of a task, XV_SERVER, created by eXVantage for memory setting and trace data transmission; the second layer is the VxWorks library; and the third layer is the hardware support, which mainly includes I/O Driver and Board Support Package (BSP). The addition of this new XV_SERVER task is part of our solutions that will be explained in the next section.

3. Solutions

Our solution to reducing overhead includes several mechanisms. First we only record the coverage information instead of the full execution trace, thus the number of probes and their CPU time requirements are reduced; second each time a probe is executed we store data in memory rather than directly writing it into files. By this way the CPU time requirement for writing file is eliminated; third we use binary data format for the trace information other than readable characters so that the memory requirement is greatly reduced.

Taken VxWorks shared memory issue into account, we pick up the API “sysMemTop()” which returns the first byte of memory that is neither used by the system nor by the user. The trace memory starts from this point to obtain blocks of memory for each original process. Since only count information of coverage execution is recorded, no execution sequence is required and the total size of the memory can be pre-determined. Figure 4 shows the selection of this memory beginning point. The other arrangement is kept the same as in Figure 2.

eXVantage runtime solution is divided into two layers from the software point of view. The upper layer contains all application tasks plus XV_SERVER task; the lower layer is composed of all related modules of eXVantage that provide supports to the upper layer

1) Application Tasks

Application tasks are created by the users and are instrumented by eXVantage before execution. Such instrumentation is mainly supported by the instrumenter module of eXVantage.

2) XV_SERVER Task

After instrumentation, original application tasks are logically organized into various connected nodes. When any of the probes on each node is executed, a piece of information will be written into the corresponding buffer created and maintained by the XV_SERVER task.

XV_SERVER task’s execution includes the following functions:

a) Setting up the three types of buffers supported in eXVantage, coverage (bit), profiling (counting) and tracing(ordered); and
b) Configuring network communication in order to send traces to trace handler program periodically or reply to its requests.

When the system boots up, XV_SERVER task is assigned priority 100. Periodic transmission will be carried out every 5 seconds to send out execution trace information.

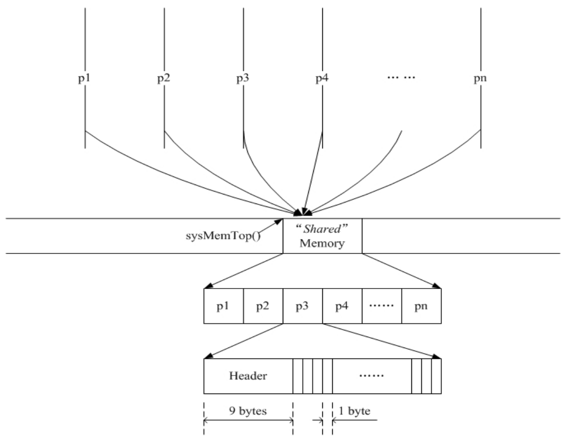

Figure 4. Solution to shared memory issue

Currently only bit buffer has been implemented, counting and ordered buffer will be added in the near future; and the ordered buffer will have to be dumped based on their used buffer size, otherwise there would be data loss.

Dynamic transmission will be carried out upon the receiving of requests from a remote trace handler, which is also going to be implemented later.

The following is a more detailed description of eXVantage run time software components and its workflow. Figure 5 shows the workflow of eXVantage run time solution on embedded systems which includes four components: APPn, XV_SERVER task, Trace Handler, and Buffer.

The following is the detailed description of the four components.

1) APPn

APPn is the original program under test. They were instrumented with probes that write into the memory buffer during the execution. Every APPn task writes execution information directly into its own buffer when a "logic" node gets executed.

2) Buffer

The buffer temporarily stores the program execution information. There are three types of buffer supported by eXVantage: Bit Buffer, Counting Buffer, and Ordered Buffer.

Bit Buffer shows if the corresponding node is executed or not; Counting Buffer records how many times the node has been executed; and the Ordered Buffer saves the node execution sequence of the whole application program. For code coverage testing, only the first one is needed. The latter two kinds of information have some other usage such as compiler optimization.

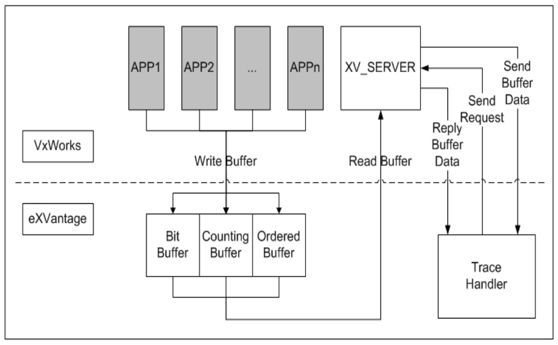

Figure 5. eXVantage VxWorks Run Time Solution

3) XV_SERVER task

XV_SERVER sends bit and counting buffer data to Trace Handler on a remote machine every 5 seconds; sends ordered buffer data to the same place based on its dynamic in-use buffer size, in another word, the size of data in the buffer. Both of the two sending operations have to guarantee no interference to the application tasks' execution.

In the meantime, the XV_SERVER task also handles requests sent from the Trace Handler and responses to them with the trace data as requested

4) Trace Handler

Trace Handler resides on a remote machine. It sends requests to XV_SERVER task for retrieving trace data dynamically. It also receives trace data from XV_SERVER task passively every certain period of time. Received trace data will be written into trace files for further analysis as shown in Figure 1.

4. Experiments

By using a timer with resolution of 1/3600, which is 0.278 milliseconds, we obtained the time cost on several measurements as listed below (Table 1). These measurements show the cost of instrumentation (probe time) and XV_SERVER task (buffer initialization and transmission).

Several other measurements have also been proposed and the work is in progress:

1) When combining the above time cost values with the eXVantage overhead on the regular non-real-time system (<1%), we have confidence that eXVantage on the embedded system with VxWorks RTOS could also achieve an overhead that is at least as low as 1%. However, some more experiments are needed to prove this.

Measurement	Time cost	Standard deviation
Trace buffer initialization	619.80 nanoseconds	0.30 nanoseconds
Execution time per trace data send	788.07 nanoseconds	1.79 nanoseconds
Execution time per probe	115.91 nanoseconds	1.10 nanoseconds

Table 1. The overhead of instrumentation

2) Tornado provides a CPU time measurement tool as one of its components, which could show the CPU time spent on each of the tasks during a specific execution. We are going to use this tool to measure the task's execution time of both un- instrumented and instrumented code, and apply statistics theory to calculate its overhead.
3) Another way to check if our instrumentation affects the original program is to get the execution sequence of the program's both un-instrumented and instrumented version, and compare with each other. This work could be done by another Tornado tool "WindView". WindView is also an instrumentation tool which instruments VxWorks API in order to get the execution sequence.

Initial results on the existing test cases show that the code coverage instrumentation didn't alter the behavior of the original program.

5. Related work

As a method of effectively checking the quality of a software test, also quantitatively monitoring the progress of testing, code coverage testing has been an active research topic for more than 10 years. M.R.Lyu et al 0 developed a coverage analysis tool named ATAC (Automatic Test Analysis for C), which could evaluate test set completeness based on data flow coverage measures and allows programmers to create new tests intended to improve coverage by examining code not covered. But ATAC didn't address embedded system's characteristics. W.E.Wong et al [2] proposed another solution, TestingEmbedded, on top of X Suds/ATAC that works for the embedded system environment, but up to now it is restricted to Symbian/OMAP platform only. Y.Y.Cho et al 0 suggested a performance evaluation system for embedded software that it consists of code analyzer, testing agents, data analyzer, and report viewer; code coverage analysis is performed by the code analyzer; report is generated from data analyzer showing the analysis results and displayed by the report viewer. However, they didn't discuss how much overhead could be caused by this system in the original embedded software. H. Thane in his doctoral thesis 0 also discussed about the code coverage technique in the testing phase of distributed real-time system development, defining the complete coverage for both single node and the whole distributed system; but similar to 0, he didn't take the instrumentation overhead into consideration, which is quite an important issue for embedded systems' timing behavior. In our coverage based testing technique, we overcome the drawbacks listed above by designing the testing environment specifically for our embedded system platform, minimizing instrumentation overhead, and applying basic performance analysis to verify the test overhead on the original embedded software.

6. Conclusion

We have successfully ported a code coverage testing tool, eXVantage, from Linux platform to an embedded system with VxWorks RTOS. The resulted code coverage testing runtime solution passed basic test cases we have at hand. Currently it had been handed to our customer for field usage.

We have also carried out some basic measurements, of which the experimental result is pretty good. According to the measurements after our implementation, our solutions together could provide us with a good enough performance for embedded system code coverage testing. When the required resources become available, more detailed and comprehensive measurements will be applied and necessary changes could be made according to the new results.

Several future works could be valuable to consider:

1) Addition of the two other kinds of buffers, counting and ordered buffers, which will require more strict server task and application tasks synchronization to prevent data loss.
2) Applying eXVantage to other software systems. We have implemented and tested our porting on one embedded system. More trials of the porting are being carried out.
3) Implementing a real time system logger. The logger works at the field during program execution to record necessary runtime

information. The information is stored in a remote trace file. With this trace file, exactly the same execution can be "replayed" as many times as users need. This can be quite helpful for debugging and remote diagnosis.

4) Product line development
   a) Language Support
      Currently we have already supported JAVA and C/C++, when there is other programming language come out in the future, we could also try to include it in our eXVantage language support.
   b) Platform Support
      Up to now eXVantage works well on both Linux and VxWorks. Next step we may port it to other platforms, so that eXVantage can be applied to a much wider domain of applications

Besides code coverage testing, the users can also leverage the coverage data collected from the run-time environment to help in debugging. It can also be used to profile the execution count to find system performance stress points. It can also locate bugs by finding the common parts of the failed test cases and subtracting the parts in successful test cases. Overall, porting code coverage testing tool to embedded systems makes the tool's related features usable for embedded systems, thus helping improve the quality of embedded software.

7. References

[1] Michael R. Lyu, J.R Horgan, Saul London, "A coverage analysis tool for the effectiveness of software testing", *IEEE Transactions on Reliability, Volume 43, Issue 4,* Dec. 1994, pp. 527-535

[2] W. Eric Wong, S. Rao, J. Linn, and J. Overturf, "Coverage Testing Embedded Software on Symbian/OMAP" *in Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, July 2006.

[3] Yong-Yoon Cho, Jong-Bae Moon, and Young-Chul Kim, "A system for Performance Evaluation of Embedded Software", *in Transactions on Engineering, Computing and Technology V1*, ISSN 1305-5313, December 2004

[4] Henrik Thane, "Monitoring, Testing and Debugging of Distributed Real-Time Systems", Doctoral Thesis, Royal Institute of Technology, KTH, Mechatronics Laboratory, TRITA-MMK 2000:16, Sweden, 2000

[5] Barry W. Boehm, *Software Engineering Economics*, Prentice Hall Ptr, October 22, 1981