A Performance Profile and Test Tool for Development of Embedded Software Using Various Report Views*

Yongyun Cho and Chae-Woo Yoo

School of Computing, Soongsil University,
1-1 Sangdo-dong, Dongjak-gu, Seoul 156-743, Korea
yycho@ss.ssu.ac.kr, cwyoo@comp.ssu.ac.kr

Abstract. In this paper, we suggest a performance profiling and testing tool that offers developers convenient environments to profile or test an embedded software's performance and to analyze the results through various graphic report views. Because the suggested tool does not need any additional hardware, it is efficient in aspects of costs and management for profiling or testing an embedded software's performance. The tool consists of a code analyzer, a test suite generator, and a report generator. The code analyzer expresses a software's structure as a parse tree and decides a position that additional codes for profiling must be inserted into. The test suite generator offers a test script wizard for users to easily make a test driver. The report generator converts a string-type result to an XML-based class instance in order to raise reusability for the result. To offer various report views we divide the instance into two sections, which are for result data and for visual views. Therefore, users can get various report views by associating the two sections according to their intention.

1 Introduction

Because an embedded system generally offers less computing resources than a general-purpose computer system does, developers make every effort to improve the quality of their embedded software and make it to always have a good performance in resource usage[3, 4]. To do this, developers occasionally use embedded software evaluation tools to increase development efficiency for embedded softwares. With a software evaluation tool, developers know whether the developed software is efficiently optimized for embedded system's restricted resources. Because embedded software is commonly developed in the cross-platform, in which the test execution occurs on a target-side but the result analysis in a host-side existing embedded software evaluation tools are based in the environments. Some of them need an additional hardware to profile and test embedded software. An additional hardware is more or less profitable in a view of times, but may be financial burden on middle-sized embedded software developers. Many embedded

* This work was supported by the Soongsil University Research Fund.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part II, LNCS 3992, pp. 510–517, 2006.

software vendors include profile and test tools in their products. However, because many of them produce the testing results that occasionally are text-based strings, to analyze the string data to find where to be revised becomes often very tiresome and time-consuming work. Of course, many embedded software vendors support profiling or testing through graphical result views. However, they do not offer various report views enough to increase the analysis efficiency.

In this paper, we suggest a graphic tool based on pure software without any additional hardware for profiling and testing embedded software's performance. The tool includes a code analyzer, a test suite generator, and a report generator. The code analyzer inserts profile codes into a target source through parsing and generates an execution file including profiling codes. The test suite generator makes test scripts and generates test drivers after parsing the scripts. In this paper, we design an XML-based test script DTD to easily make a test script. To generate various report views, the report generator uses a result converter to represent string-typed profile or test results to object instances through XML-based classes. The XML-based class consists of two parts. One is to represent graphical views, and the other is to describe result data. Because the two parts are separated, users can get several graphical views by associating one result data with various graphical values according to the users' preference.

2 Related Work

2.1 Existing Profile and Test Tools for Embedded Softwares

Telelogic's Tau TTCN Suite is a system to test telecom and datacom equipment ranging from built-in communication chips to huge switches and intelligent network services. It includes various tools such as script editors, compilers and simulators, but it is not suitable for testing embedded software because it is a test tool for telecommunication vendors. It also is very expensive because it is mostly additional hardware equipment to test telecom tools. AstonLinux's CodeMaker is an IDE (Integrated Development Environment) to develop embedded software based on Linux in Windows. It supports remote debugging and source-level debugging, but it doesn't offer any function to test and analyze embedded software's performance because it is only an IDE for specific RTOS/chip vendor. Rational's TestRealTime is a target-based performance evaluation system for real-time software. It offers various result views so that users can easily analyze real-time software's performance. It can also execute various performance tests ranging from memory usage, memory leak, cpu usage to code coverage. However, it is somewhat difficult for developers to understand the meaning of the result view at a glance because it is not a performance system for embedded software based in the cross-platform environment. Additionally, it offers a somewhat difficult script language for users to make a test script. Because the script language is a perfectly new script language, users must spend a lot of time to study how to use for making a test script. To solve the problems, in this paper we design an XML-based script language that common users can intuitively understand and easily use.

3 The Proposed Performance Evaluation Tool

3.1 A System Architecture

In this paper, we suggest a tool for profiling and testing embedded software's performance that consists of pure software without additional hardware equipment and offers such various performance tests as memory, code coverage, code trace and function performance [3, 5, 6]. The evaluation tool offers users graphical report views that they can easily and intuitively analyze the test result. Figure 1 is the proposed architecture for a performance evaluation tool.

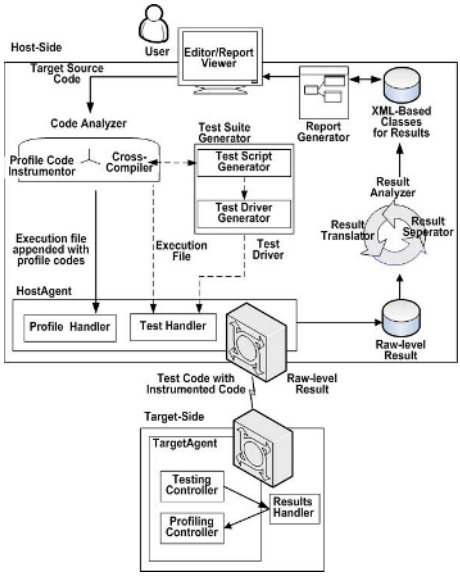


Fig. 1. The Architecture of The Proposed Performance Evaluation Tool

In Fig. 1, the proposed evaluation tool is composed of a GUI, host/target-side agents, a code analyzer, a result analyzer, and a report viewer. The code analyzer consists of the code corrector to insert additional code into source code and a cross-compiler to create target-executable file for the source code. The evaluation tool is a client/server model based in host-target architecture. Because an embedded system offers insufficient memory and an inconvenient user interface, the suggested tool places an agent not only on the host-side to offer users convenient GUI but also on the target-side to execute performance testing of the software in the target board. The agents keep a communication connection to deliver source files and test result to each other. First, the host-side agent transfers an inputted source to the target-side agent through a LAN cable or wireless network. Then, the target-side agent executes a testing process, gains results from the test events, and sends the results to its host-side counterpart.

Consequently, the host-side agent stores the string-typed result received from target-side one into the result DB.

3.2 Profiling for Embedded Softwares

Generally, embedded software must use minimum process and memory resources. To make embedded software to meet those requirements, the suggested tool tests software's performance for 4 items, which are trace, memory, performance, and code coverage profile [4, 5]. Through trace profiling, users can trace what functions are executed according to the software's execution process and find what functions are unnecessarily called. Through memory profiling, users can know about memory allocation/de-allocation, memory leaks, and code sections frequently to use memory. Users can use performance profiling to estimate how much time it takes to execute the whole or part of embedded software and it confirms whether it becomes optimized in embedded system. Code coverage profiling offers users information about used or unused code sections, and frequently or infrequently used code sections. Users can make embedded software more efficient by using information profiled according to the 4 items. In this paper, we classify string-typed profiling results according to the items and converts them into instances objectified by the classes. The report generator uses the instances to make various report views according to the user's requirements. The instances consist of a section to represent result data and a section to describe graphical elements that construct a report view. For example, when a user wants to know how many memory are used in a test target embedded software through a pie graph or a bar graph, the user can get the information by combining the software's memory profile result with class pre-defined for the graphical elements, pie or bar.

3.3 Testing for Embedded Softwares

To enhance the performance of embedded softwares, testing is very important. The suggested tool supports testing for embedded softwares. Commonly, to test embedded softwares users need a test source code and a test driver [7]. A test driver calls code units that must be tested and collects results of the execution. A user can make a test driver in a program language that a test target program is written in. However, because it needs a lot of times, a user commonly uses a tool that automatically translates a test script that a user makes with a script language to a test driver. With existing tools, a user must use a specific script language which is never easy for common users to understand and use. In this paper, we suggest an XML-based test script language. Because the script language is based on XML, users can easily understand its grammar and learn how to use it. Figure 2 shows the schema of the suggested script language.

In Figure 2, <testDriver> is a test script's root. <test> can occur repeatedly and describes test cases. <element> is to set initial values and expected values which are <var> element's attributes. <run> describes the test execution. It executes <runTest> or a branch routine by using <if>, <then>, and <else> according to the test results. Figure 3 shows an example test script for add() function to add two integer values using the suggested script schema.

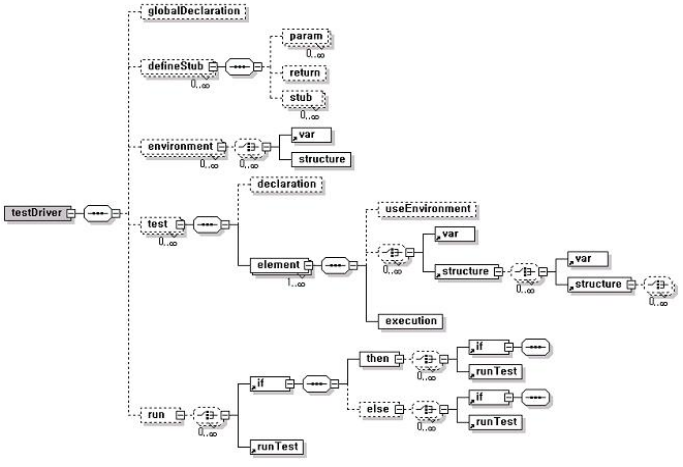


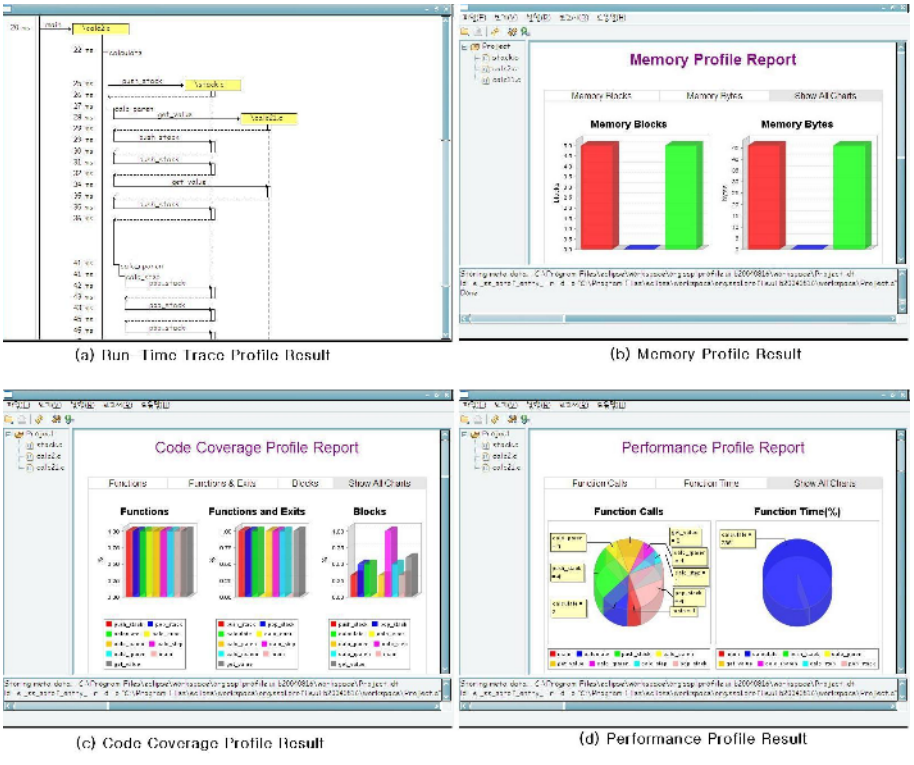
Fig. 2. The Suggested XML-Based Script Language's Schema

Fig. 3. A Sample Test Script for add(x, y) function

4 Experiments and Results

The suggested evaluation tool is implemented in the Java 2 platform and we use an HRP-SC2410 (Ami) launched strong ARM chip and embedded Linux. We use a calculator program in C language as input source code. Its code size is about 520 lines and consists of three modules. We will profile them depending on the 4 items, which are trace, memory, performance, and code coverage profile, and execute unit testing for them. As results for the experiments, we will show various graphical report views for the profiling and testing process.

Figure 4 shows various report views generated by the suggested tool after profiling the sample C program. Each report view's visual element is not static, because the suggested tool divides profile results with visual elements for construction of report views. Therefore, users can make various report views according to their intention. Figure 4(a) shows a trace view that is a style of UML



(a) Run Time Trace Profile Result (b) Memory Profile Result

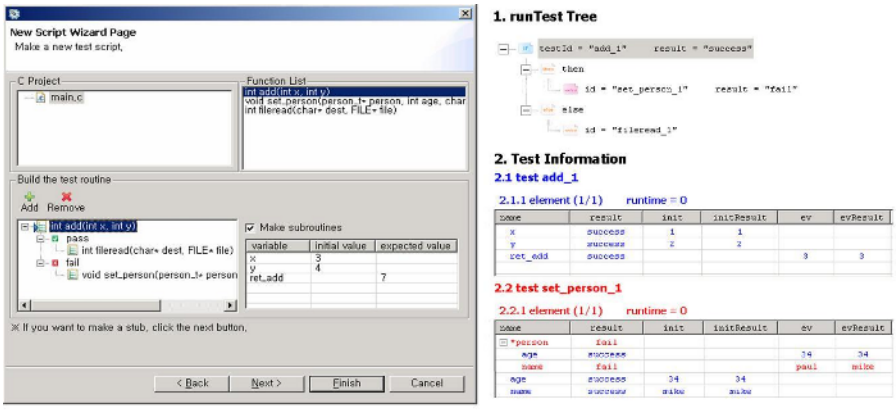
(c) Code Coverage Profile Result (d) Performance Profile Result

Fig. 4. Various Report Views for Profile Results

sequence diagram [9, 10]. With the view, developers can analyze a program's execution by tracing functions' call orders. Figure 4(b), (c), and (d) show respectively memory, code coverage, and a performance report view. With Figure 4(b), we can know how much memory software uses and can find whether any freed memory section is called for freeing, or any unallocated memory section is called for freeing. With that, developers can find where memory leaks or illegal memory usages happen. With Figure 4(c), we can know whether functions in source code were executed and how many blocks those functions were executed. Through the information, developers can decide which function must be revised to enhance the entire efficiency of the software. With Figure 4(d), we can know the call times for each function and the function's execution time with its lower functions or without them. We can also find the average execution time of any

function against total execution time. Through the result, developers can know which function is most busy, and they can divide the burden of the function to other function in order to raise the execution efficiency of the software.

Figure 5 shows a test driver wizard for testing and a result view after testing.



(a) Test Driver Wizard for Making Test Driver (b) Test Result View after Testing with Test Driver

Fig. 5. The test driver wizard and the result view

In Figure 5, we set 3 and 4 as initial values for variable x and y. And we set 7 as expected value. After the testing process has finished, if the test result is 7, the next test route will be the *fileread* function. If the result is not 7, the next test route will be the *set_person* function. As Figure 5(b) shows, because the result value after testing is 7, the *fileread* function will be executed as the next step. With the test driver wizard, users can easily make a test driver and find fault positions in a source program by tracing test path.

5 Conclusion

In the development of embedded software, developers have always to consider good efficiency not only for resource usage but also for the development time. In this paper, we suggest a GUI-based tool which easily profiles and tests embedded software's performance and intuitively analyzes the results. For this, we suggested a code analyzer that produces a parse tree as a result of parsing and uses a parse tree handler to insert profile codes into aimed positions of the source code. With the parse tree handler, the code analyzer can decide exact instrumentation points not only in compilation time but also in execution time. Additionally, we also suggested the test suite generator that makes a test script and a test driver. For that, we have designed an XML-based test script DTD for easy understanding and using. By using the DTDs, the report generator converts string-typed results to XML class instances and generates various report views

through which developers can easily understand the meaning of the results and revise the inefficient portions of the source codes. In the experiments with the suggested tool, we profiled the performance of some C source codes for the 4 items mentioned in Section 3.2, and showed the results graphically through the report viewer. We also showed that developers can easily make a test driver by using the test driver wizard. Therefore, through the suggested tool, developers can clearly know what must be fixed in software's source code and can improve development efficiency of embedded software.

References

1. Roper, Marc, Software Testing, London, McGraw-Hill Book Company, 1994.
2. Boris Beizer, Software Testing Techniques 2nd edition, New York: Van Nostrand Reinhold, 1990.
3. Bart Broekman and Edwin Notenboom, Testing Embedded Software, Addisson-wesley, Dec. 2002
4. Dr. Neal Stollon, Rick Leatherman and Bruce Ableidinger,Multi-Core Embedded Debug for Structured ASIC Systems, proceedings of DesignCon 2004, Feb, 2004.
5. David B. Stewart, Gaurav Arora, A Tool for Analyzing and Fine Tuning the Real-Time Properties of an Embedded System. IEEE Trans. Software Eng., Vol.TSE-29, No.4, April 2003, pp.311-326.
6. Ichiro Satoh, A Testing Framework for Mobile Computing Software. IEEE Trans. Software Eng., Vol.TSE-29, No.12, December 2003, pp.1112-1121.
7. Paul Anderson, Thomas W. Reps, Tim Teitelbaum, Design and Implementation of a Fine-Grained Software Inspection Tool. IEEE Trans. Software Eng., Vol.TSE-29, No.8, August 2003, pp.721-733.
8. John Joseph Chilenski and Steven P. Miller, Applicability of Modified Condition/Decision Coverage to Software Testing, Software Engineering Journal, September 1994, Vol. 9, No. 5, pp. 193-200.
9. Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, Eunjee Song, A UML-Based Pattern Specification Technique, IEEE Trans. Software Eng., Vol.TSE-30, No.4, April 2004, pp. 193-206.
10. Ludovic Apvrille, Jean-Pierre Courtiat, Christophe Lohr, Pierre de Saqui-Sannes, TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit. IEEE Trans. Software Eng., Vol.TSE-30, No.7, July 2004, pp. 473-487.