An Investigation into the Cost of Unit Testing on an Embedded System

Wensi Qiu

Research Thesis

M.Sc. Computer Science (Software Engineering)

Department of Computer Science

National University of Ireland, Maynooth

County Kildare, Ireland

A thesis submitted in partial fulfilment

of the requirements for the

M.Sc. Computer Science (Software Engineering)

Supervisors: Dr. Stephen Brown

January 28, 2011

Table of Contents

1 Introduction....5
2 Background....5
2.1 Characteristics of Embedded System....5
2.2 Testing Embedded Systems & Software....6
2.3 The Benefits of Unit Testing in the Embedded Environment....6
2.3.1 Hardware-related Reasons....6
2.3.2 Software-related Reasons....7
2.3.3 Compiler-related Reasons....8
3 Related Work....8
4 Remote Unit Test Model....9
4.1 Overview....9
4.2 The Architecture of the Experiment....9
4.2.1 Host-based Unit Testing....10
4.2.2 Target-based Unit Testing....10
4.3 Remote Test Performance Model....11
4.4 Test Tool....13
5 Experiment Setup....13
5.1 Build/Run Environment....13
5.2 Details of the implementation....14
6 Results....15
6.1 Results of Host-based Unit Testing....15
6.2 Results of Target-based Unit Testing....17
6.2.1 Results from Host1....17
6.2.2 Results from Host2....19
6.3 Operating System Overhead....21
7 Discussion....21
7.1 Experimental Results....21
7.2 Performance Results....23
7.3 Summary....23
8 Conclusions and Future Work....24
REFERENCES....24
APPENDIX – Source Code....25

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of M.Sc. in Computer Science (Software Engineering), is entirely my own work and has not been taken from the work of others – save and to the extent that such has been cited and acknowledged within the text of my work.

Signed: _____ Date: _____

Abstract

The quality of embedded software is important, especially for life-critical and mission-critical embedded systems. And software testing is a key activity to ensure the quality of embedded software. Both system testing and unit testing are vital to test embedded software. Unit testing is probably more important to ensure there are no latent faults. System testing is almost invariably done on a target system, but unit testing is normally done on a host system, as standard test frameworks generally don't support the execution of unit tests on a target system. This paper analyzes the significant quality benefits to run unit testing on a target system. Also, it presents a performance model for evaluating the test execution cost, discusses the experimental results, and identifies the factors which influence the experiment.

Keywords

Embedded system, unit testing, performance

1 Introduction

Embedded systems are the most common form of computer in use today. They can be found in all kinds of electronic devices every day, such as mobile phones, televisions, microwaves, and etc. According to a market research report [1], the global market for embedded system technologies was worth $92.0 billion in 2008. It is expected that it will increase to $112.5 billion in 2013. The market for embedded devices consists of embedded hardware and embedded software. The figures for embedded hardware and embedded software were $89.8 billion and $2.2 billion respectively in 2008. As the market grows steadily, it is expected that embedded hardware will reach 109.6 billion in 2013. And embedded software will amount to $2.9 billion in 2013.

It is important ensure to the quality of embedded systems, especially in life-critical and mission-critical areas, such as aviation, aerospace, and public transportation, where embedded systems are playing an increasingly important role. If the embedded software fails in these areas, the consequences could be disastrous. The June 2009 Washington Metro train collision was a typical example [2]. The accident killed nine people and injured 80. The later investigation found that the accident may have been due to wet leaves on the tracks and a computerized system that wasn't programmed for such a condition. Also, from a financial viewpoint, it is vital to ensure the quality of embedded software. If there are some bugs in an embedded device, it is sometimes very expensive, difficult or even impossible to fix the bugs in a massive market. For example in [3], a large urban school was reported to have significant software problems in its new ERP payroll system. More than one third of employees received incorrect pay checks, which caused overpayments of $53 million and the corresponding lawsuit problem. Thus, it is necessary to consider the embedded software quality as soon as in the development to reduce the unexpected costs.

Software testing is an effective way to enhance the quality and reliability of embedded software and systems. One important criterion for a successful embedded system is whether its software performs well on its hardware platform. Nowadays, testing accounts for a big proportion of resources in software projects. A recent survey gives a qualitative analysis of the practices and preferences in testing different categories of software development [4]. The categorization of respondents was based on safety-criticality, agility, distribution of development, and application domain. One of the noteworthy testing research directions from an industrial perspective seems to be test driven development as indicated by the results of the survey. It shows the fact that software testing is an indispensible procedure to ensure the quality of embedded software in practice.

This paper focuses on the performance cost of unit testing on a target system. To test embedded software at different levels, both unit testing and system testing are important on the host system or on the target system. And unit testing can ensure high level quality of the embedded software. It is more challenging to do unit testing on a target system, as many test frameworks do not support the execution of unit tests on a target system. For hardware, software, and compiler reasons, testing might result in the different code operations on a host system versus on a target system. Therefore, the benefits to grantee the quality of embedded systems by target-based unit testing are worthwhile. This paper proposes a remote unit test model to do unit testing on a target system.

2 Background

2.1 Characteristics of Embedded System

An embedded system consists of a small CPU and I/O devices. Unlike a general-purpose computer, it is dedicated to perform some specific tasks. To reduce costs, many embedded systems are designed to be simple. The O/S, memory and other features of an embedded system are often of lower-level

comparing to that of a general computer. For example, a typical embedded system is embedded within an electronic device, in which its memory size is only a few KB, small or no user input/output (such as screen or keyboard), and does nothing but the required tasks.

2.2 Testing Embedded Systems & Software

It is more difficult to do embedded software testing than embedded system testing. Testing embedded software is more challenging because embedded targets typically don't have the resources to run a full unit test environment (such as CppUnit) and store the test results.

Instead of running tests on a real device, tests can be executed on a host system via simulation or emulation. A simulator can simulate the CPU of an embedded device, thus allowing the 'target binary executable code' to be executed on a host system. For one thing, it can be hard to provide inputs to this code, and to verify the outputs, unless the simulator provides interfaces to support input or output. Also, it is hard to simulate a real device. For example, the AVR Simulator 2 does not provide any extensions to simulate a radio device – generally all you can do is to input 'test vectors' of bit streams into the Input ports, and check the result bit streams out of the output ports. Thus it is difficult, or impossible, to test the hardware-dependent code.

An emulator can typically compile the embedded code into 'host binary executable code' and then be tested on a host system. This allows standard frameworks (such as CppUnit) to be used. But the code is not executed in the hardware environment of an embedded system, and probably won't be executed in the full software environment either (as some or the entire target O/S may be hardware-dependent). An additional issue is that, unless the code is designed with testability in mind, much embedded code contains a mix of hardware-dependent code and hardware-independent code, making even the hardware-independent code difficult to test on a host system.

Due to the disadvantages discussed above, both a simulator and an emulator would not provide a high check of quality and reliability of the correct code operations on a real embedded device. Thus, it is worth running tests on a real embedded device.

2.3 The Benefits of Unit Testing in the Embedded Environment

There are many advantages in host system debugging and testing [5]. The effects of changes can be checked almost instantly, thus reducing costs and yielding high productivity. However, limitations arise as a result of three particular factors: interaction of programs, speed of execution, and host/target hardware differences. A common problem is that unit testing on a host system may provide different results from testing on a target system. It means that tests may pass on a host, but fail on a real target under certain conditions. There are a number of reasons for this problem relating to hardware, software, and compiler as discussed in the following section.

2.3.1 Hardware-related Reasons

(1) CPU architecture (endianness, word size, interrupts, speed, and timing)

CPU endianness influences the code operation. Some operations are explicitly endianness dependent, for example, bit operations which are widely used in embedded software. A difference in endianness CPU might result in the code working properly on a host system, but not on a target system. Word-size is another similar issue - typically this relates to the width of the registers used. Again, difference in word-size might cause the same problem - the code works properly on a host system, but not on a

target system. Also, interrupts are the typical hardware features which you can access to on an embedded system, but not on a host system running a desktop O/S. It is impossible to fully test interrupt-related code on a host system. But the incorrect handling of data shared between 'normal' and interrupt level code is a classic source of faults, and needs testing. Finally, speed and timing differences may lead to fault-related timing windows not being seen on a host system, but appearing on a target system.

(2) Memory (architecture, memory management, size, wraparound, and data alignment)

Embedded systems often use special architectures for cost, power, and performance reasons: such as the Harvard Architecture on the AVR ATMega series, with different programs and data spaces. Secondly, the memory management hardware and O/S on a host system can typically provide a large, linear memory space to applications, with program and data in the same address space. The memory management can prevent accesses outside of allowable addresses, and thus prevent "wraparound" of pointers. Yet a target system normally does not have the hardware to support memory management. A typical target system might have say 4KB of data space, and if a pointer accidentally is incremented beyond this, the access will wrap around back to the base of memory. Finally, the different alignment of data to memory may also cause problems. For example, if n-byte data structures must be n-byte address aligned.

(3) Devices

Devices can be simulated (typically at the hardware level) or emulated (typically at the software interface level) to allow unit testing on a host system. However, unless the simulation (or emulation) is based on the design specification of the device (for example, VLSI), a simulator (or an emulator) is unlikely to behave as identically as a real device does. Also, the timing for an emulated device is likely to be substantially different from that of the real hardware. Many devices are very complex; if the manufacturer does not provide a simulator/emulator package, it may not be possible to test the device-related code on a host system.

2.3.2 Software-related Reasons

(1) O/S

Some source code may be O/S dependent, and thus can only be supported on the specified O/S platform. And the lower level of O/S code is O/S dependent. For performance and cost reasons, embedded systems may not have the same neat layering of hardware-dependent and hardware-independent code, which would allow the application to be tested with the hardware-independent code on a host system. Embedded applications may also make direct use of hardware-dependent O/S features which cannot be emulated on a host system, for example interrupts, or I/O.

(2) Libraries

Different software has its own libraries. Some functions supported by the specified library might be hardware-dependent, thus the code might working correctly on a host system but not on a target system.

2.3.3 Compiler-related Reasons

(1) Data Location

The host and target compiler may allocate different addresses for different data structures (for example, due to difference in word size, or different optimisation strategies). This can allow errors to pass undetected in a host system (which may well pack data less densely) that may fail on a target system (with more densely packed data, or with less 'unused' memory space).

(2) MACROS

MACROS are often used with the C/C++ programming languages to provide levels of tailoring not supported by the base language (for example, generics, places where functions can't provide the required functionality, even constants). The definition of certain MACROS may be different between the host native compiler and the host cross-compiler for the target (for example, I/O MACROS may call a stub instead) and thus the actual target MACROS are untested.

These are all reasons why unit testing may give different results on a host system versus on a target system. Unit testing is an effective and general way to test an embedded system, since unit testing on the target system can provide a higher level of confidence in the reliability of code operation on a real device. Also, unit testing ensures the quality of the smallest component of software design – program modules. It partitions a complex system so the system can be tested in relatively independent modules [6], thus protecting code integrity as application evolve and verifying error handling in a much simplified way. At the unit test level, it is more likely to find bugs early in the development, since the system is tested on well-tested units [7].

3 Related Work

Many papers deal with test models and test tools for embedded systems. Vincent Encontre gives a general introduction to testing embedded systems [8]. His article goes through a generic test iteration, from which the granule to be tested is derived (the word granule means the smallest component in his article). Instantiating this iteration, his article addresses how to test complex embedded systems from six different aspects. As mentioned in his article, “these aspects are: software unit testing, software integration testing, software validation testing, system unit testing, system integration testing, and system validation testing”. Also, a couple of issues which explains what makes embedded systems so difficult to develop and test are analyzed. These issues are categorised as follows, “Separation between the application development and execution platforms; A large variety of execution platforms and thus of cross-development environments; A wide range of deployment architectures; Coexistence of various implementation paradigms; Tight resources and timing constraints on the execution platform; Lack of clear design models; Emerging quality and certification standards”. From a general perspective, his article analyzed the difficulty of testing embedded systems and corresponding solutions, which gives a background to this paper.

A new comprehensive embedded software test process model is proposed by Hua-ming Qian and Chun Zheng [9]. The shortcomings of the traditional V model are: the test is thought to a last phase after coding, and its objective is to find defects in the procedure. But the problem hidden in the requirement phase cannot be found until acceptance testing. This new model is put forward to overcome the shortcomings of the V model, combined with the characteristics of embedded system. In the model,

software testing and the performance of the architecture of embedded software systems are paid attention to from the requirement phase, thereby avoiding performance problems later in the software realization. In particular, their paper analyses the difference between general software testing and embedded software testing, and indentifies the characteristics of embedded systems as well as the difficulties of embedded software testing. In this aspect, their paper emphasise the importance of embedded software testing, which is similar to this paper.

A prototype embedded system test framework is presented by Michael Smith et al [10]. Their XPI embedded life cycle applies a test-driven approach, offering a path to solve many issues for embedded systems. Their test framework, which supports remote unit testing on a target system, is similar to that is presented in this paper. That is, a host system provides the standard framework and the test harness, and a target system runs the actual tests. But the performance cost of testing is not addressed in their paper.

An automatic embedded testing tool is developed by Chorng-Shiuh Koong and Hung-Jui Lai [11]. This new testing tool is called ATEMES, which means Automatic Testing Environment for Multi-core Embedded Software. It can automatically generate test cases and test drivers, and support unit test and coverage test. The ATEMES system's architecture layers consist of the host-side auto testing module and the target-side auto testing module. Similar modules can be divided in the architectural view of the experiment proposed in this paper, but the focus of this paper is not automated testing but the performance of unit testing.

4 Remote Unit Test Model

4.1 Overview

Remote unit test execution is an effective way to support unit testing on a target system. Typically the memory of a target system is not enough to run tests or collect test results. In the remote unit test model, the host system provides a standard harness to run the tests and collect the results, and the target system runs the actual tests. The model connects a host system and a target system via a communication link.

The performance of unit testing is the focus of the experiment. Performance is an importance factor in testing. In this test model, performance is most dependent on the basic architecture, and not on inefficient coding. The cost is measured in terms of real time in the experiment.

4.2 The Architecture of the Experiment

The architectures for host-based unit testing and the target-based unit testing are shown in Figure 1 and Figure 2. All the test cases are according to the specifications of the class CarTax, see appendix [12]. And the code in the CarTax example is not related to the embedded system features since the purpose of the experiment is to measure the performance aspect not the test results about the embedded systems.

4.2.1 Host-based Unit Testing

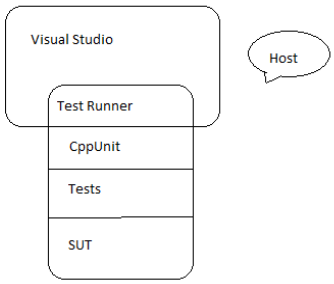

Figure 1 Host-based Unit Testing

The host-based unit testing in Figure 1 is straightforward. On a host system, the CppUnit test runner tests the specified source code within Visual Studio. The Test Case used is shown as follows.

```
Test Case:
void TestCarTax::test1()
{
   CarTax ct1(1999);
   ct1.setCO2(2);
   int actual = ct1.calculateTax(false);
   int expected = 3000;
   CPPUNIT_ASSERT_EQUAL(expected, actual);
}
```

4.2.2 Target-based Unit Testing

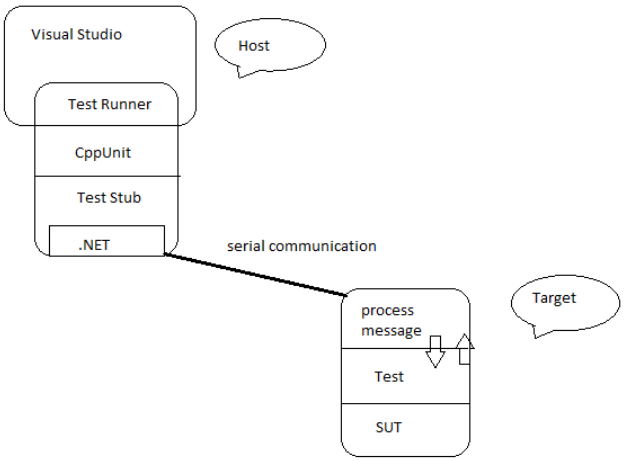

Figure 2 Target-based Unit Testing

Figure 2 shows target-based unit testing. On the host system, the CppUnit test runner is executed within Visual Studio. Then the test stub runs, starting serial communication with the target system. The serial communication is supported by the .NET framework. A simple, low overhead protocol is used to send the test ID and collect test results. On the target system, the test ID is extracted from the message, the required test is run, and then a result message is sent to the host system. When the host system receives the result message, it exacts the test result and records it to a log file. The test code used on the host system and on the target system is as follows.

Test case on the host system:

```
void TestCarTax::test1()
{
   HostMessage::sendMessage("R01\n");   //Send test ID
   bool status = HostMessage::getStatus();    //Get test result
   CPPUNIT_ASSERT(status);       //Assert the test result
}
```

Test case on the target system:

```
bool CarTaxTestCase::test1()
{
   CarTax ct1(1999);
   ct1.setCO2(2);
   int actual = ct1.calculateTax(false);
   int expected = 3000;
   if(expected == actual)
      return true;
   else
      return false;
}
```

NOTE: The difference between the source code on the host system and that on the target system could use a MACRO definition CPPUNIT-ASSERT-EQUAL to hide and keep the source code the same. For clarity, the difference is shown here.

4.3 Remote Test Performance Model

The remote test performance model is shown in Figure 3. In this model, the cost to run target-based unit testing, which is measured in terms of execution time, can be divided into different parts: test time on the host system, communication time, and test time on the target system (includes pre-process message time, post-process message time, and actual test time to run test stub). Two messages are used - one is to send test ID, another is to collect the test result. In this model, no checksum is applied

because the focus of the experiment is not the test results but the performance of unit testing. And with low baud rates and short cables, data corruption is unlikely to happen.

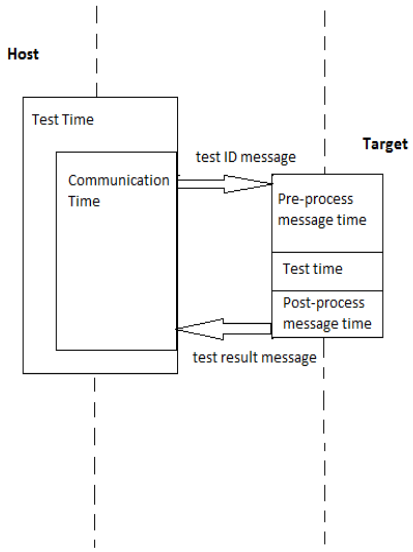

Figure 3 Remote Test Performance Model

The communication takes up a large proportion of the cost (refers to execution time as before). By definition, the communication time is determined by the baud rate, the number of symbols transmitted per second. For RS232, despite the start bit and the stop bit, each baud transmits 8 bits, so the bit rate can be calculated:

bit rate = (baud rate * 8) / (8 + start bit + stop bit)

That is,

bit rate = (baud rate * 8) / 10

The duration time for each data bit = 1/ bit rate

As in the remote test performance model (see Figure 3), there are two messages send between the host system and the target system. To be consistent with the actual experiment, the test ID message is of two-character length, and the result message is of four-character length.

Then the communication time can be calculated:

communication time = the duration time for each data bit * the number of data bits for the messages

The estimated communication time for a four-character message and a two-character message is calculated and listed in Table 1.

Table 1 Estimation of the Communication Time

Baud Rate (baud)	Communication Time (milliseconds)
9600	0.7812
19200	0.3906
28800	0.2604
57600	0.1302
115200	0.0654

4.4 Test Tool

In the experiment, CppUnit [13] [14] was selected as the test framework. CppUnit is primarily targeted at C++ unit testing, and is derived from the widely used framework JUnit. The three open source tools: CuTest, Cunit, and CppUnit are compared in Table 2.

Table 2 Test Tools

Criterion\Tool	CuTest	Cunit	CppUnit
Interface	Console	Automated, Basic, Console, Curses	TexiUi mode and MfcUi mode
Integration with IDE (Visual Studio)	NO	YES	YES

CuTest [15] is small, since it only consists of a single .c and .h file. But the framework is not as structured as that in Cunit and CppUnit. And it does not integrate with any IDE, and is run from the command prompt. Cunit [16] offers a unit testing framework for C programs. It is built as a static library which is linked with the testing code. Compared to these two tools, CppUnit [14] provides a more standard framework for unit testing. It has a more comprehensive library, and it offers many high level features integrated with Visual Studio.

5 Experiment Setup

5.1 Build/Run Environment

Details of the build/run environment are shown in Table 3. Two hosts are used in the experiment to evaluate the costs based on different hardware/OS platforms. Baud rates and compiler optimization levels are set to different values to evaluate the corresponding costs in the experiment as well.

Table 3 Run Environment of the Experiment

	Host1	Host2	Target
Hardware platform	Laptop: Intel(R) Core(TM) Duo CPU @2.10GHZ	Desktop: Intel(R) Core(TM)2 Duo CPU E7500 @2.93GHZ	8-bit AVR Microcontroller
O/S platform	Windows XP	Windows 7	AVR Library
Compiler	Visual Studio compiler	Visual Studio compiler	AVR compiler
Machine code	i386 machine code (intend port COM1)	i386 machine code (intend port COM1)	AVR machine code
Memory size (RAM)	1.99GB	4GB	128 KB of In-System programmable Flash & 4 KB EEPROM & 4KB Internal SRAM
Width	32 bit	32 bit	8 bit
Baud Rate	Up to 128000 baud/second	Up to 128000 baud/second	Up to 28800 baud/second

A High-Resolution timer is used to record the executing time in the program. The QueryPerformanceCounter and QueryPerformanceFrequency approach used in the experiment is a reliable and standard way to record time.

5.2 Details of the implementation

As the time to run one test is too small to measure, each test is repeated multiple times and the average test execution time calculated. For host-based unit testing, each test is repeated 1000 times in each execution. And for target-based unit testing, each test is repeated 100 times in an execution.

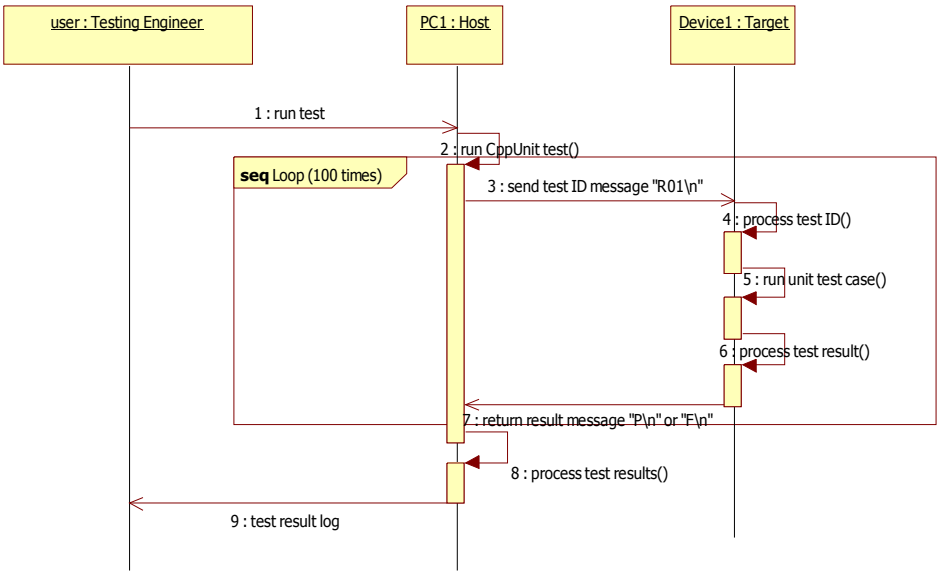

Figure 4 Sequence Diagram of Target-based Unit Testing

NOTE: Message "R01\n" means to run test 01, and Message "P\n" or "F\n" indicates whether the test passed or failed.

Figure 4 describes the following implementation steps.

Step 1: To start up, the program sets up a CppUnit test fixture, and records the clock time before running the unit tests.

Step 2: To get the average cost, the same test is repeated 100 times in each execution. Each time, it triggers a serial communication, and sends a test ID to the target system. Then the host system waits to receive the result from the target system, and processes the test result by Step 4.

Step 3: To run a unit test on the target system, after receiving the test ID, the target system runs the corresponding test, and sends the result to the host system.

Step 4: To process the test result, the host system records the result into a log file. Then it repeats the test by Step 2 again. After the test is repeated 100 times, the real time to run the program is also recorded into the log file.

6 Results

The experimental results are presented in this section. The cost is measured in terms of the real time.

6.1 Results of Host-based Unit Testing

Table 4 Cost to Run Host-based Unit Test on Host 1 (Sample Size: 20)

Average cost to run a unit test 1000 times	**Average cost to run a unit test**
557 milliseconds	0.557 milliseconds
NOTE: The compiler optimization is set as disabled in Visual Studio 2008, and the average data is based on the sample as shown in Figure 5.	

As shown in Table 4, the average cost to run a unit test is about 0.557 milliseconds on host1.

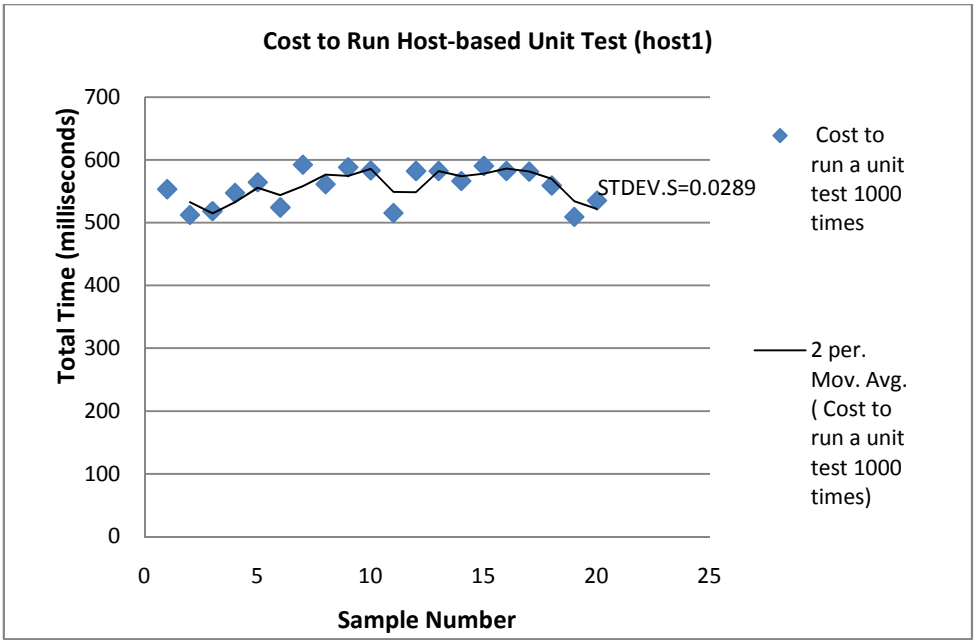

Figure 5 A Sample of Host-based Unit Test Cost on Host1 (Sample Size: 20)

NOTE: STDEV.S means the standard deviation of the sample

Figure 5 shows the sample results about the cost to run a unit test on host1, varying from 0.509 milliseconds to 0.590 milliseconds, and the standard deviation is 0.0289.

Table 5 Cost to Run Host-based Unit Test on Host2 (Sample Size: 20)

Average cost to run a unit test 1000 times	**Average cost to run a unit test**
290 milliseconds	0.290 milliseconds
Note: The compiler optimization is set as disabled in Visual Studio 2008, and the average data is based on a sample of 20 as shown in Figure 6.	

As shown in Table 5, the average cost to run a host-based unit test on host2 is approximately 0.29 milliseconds.

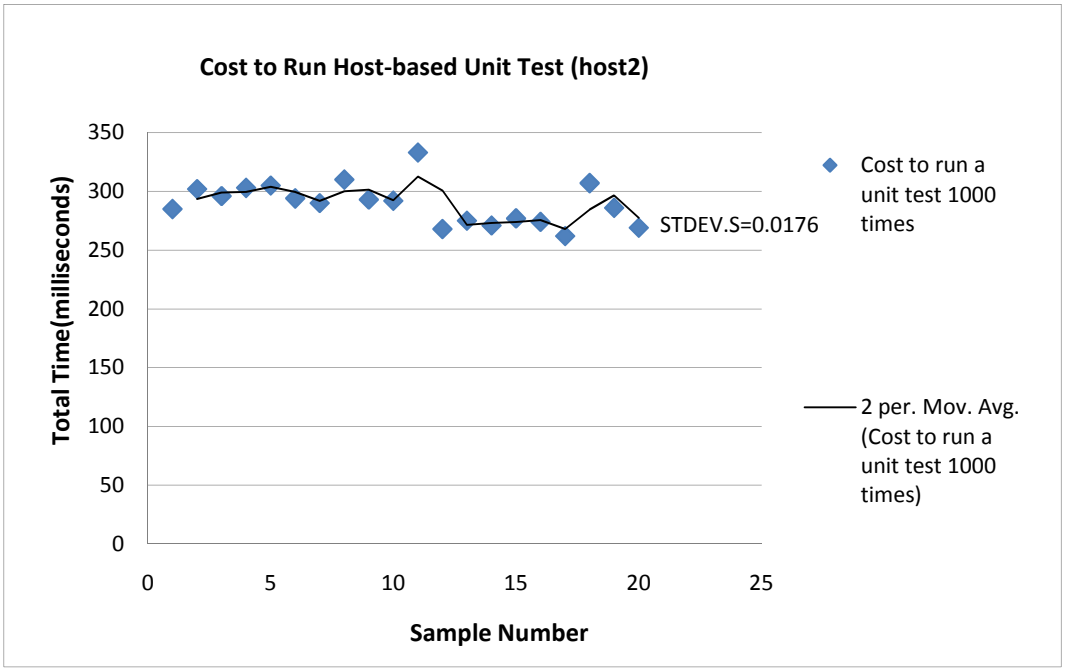

Figure 6 A Sample of Host-based Unit Test Cost on Host2 (Sample Size: 20)

NOTE: STDEV.S means the standard deviation of the sample

And as shown in Figure 6, the standard deviation for the host2 sample (0.0176) is smaller that of the host1 sample (0.0289).

The differences among the costs in each sample are caused by other activities in the host system, such as device interrupts, networking activity, page faults, disk I/O, other applications running. These activities can take up the resources of the host system, and thus affect the experiment results.

Also, host1 (laptop, Windows XP) and host2 (desktop, Windows 7) are used in the experiment to show the experimental results based on different O/S and hardware.

6.2 Results of Target-based Unit Testing

6.2.1 Results from Host1

Table 6 Cost to Run Target-based Unit Test from Host1

Baud Rate	Total Time(100 times)	Average time
9600	34784 milliseconds	347.84 milliseconds
19200	27223 milliseconds	272.23 milliseconds
28800	24620 milliseconds	246.20 milliseconds
NOTE: The compiler optimization modes are set as disabled in Visual Studio 2008 and –o0 in AVR Studio 4 respectively. The average data is calculated based on a sample of 20 as shown in Figure 7.		

Table 6 shows the average costs to run a target-based unit test from host1. The results are about 347.84 milliseconds (baud rate: 9600), 272.23 milliseconds (baud rate: 19200), and 246.20 milliseconds (baud rate: 28800).

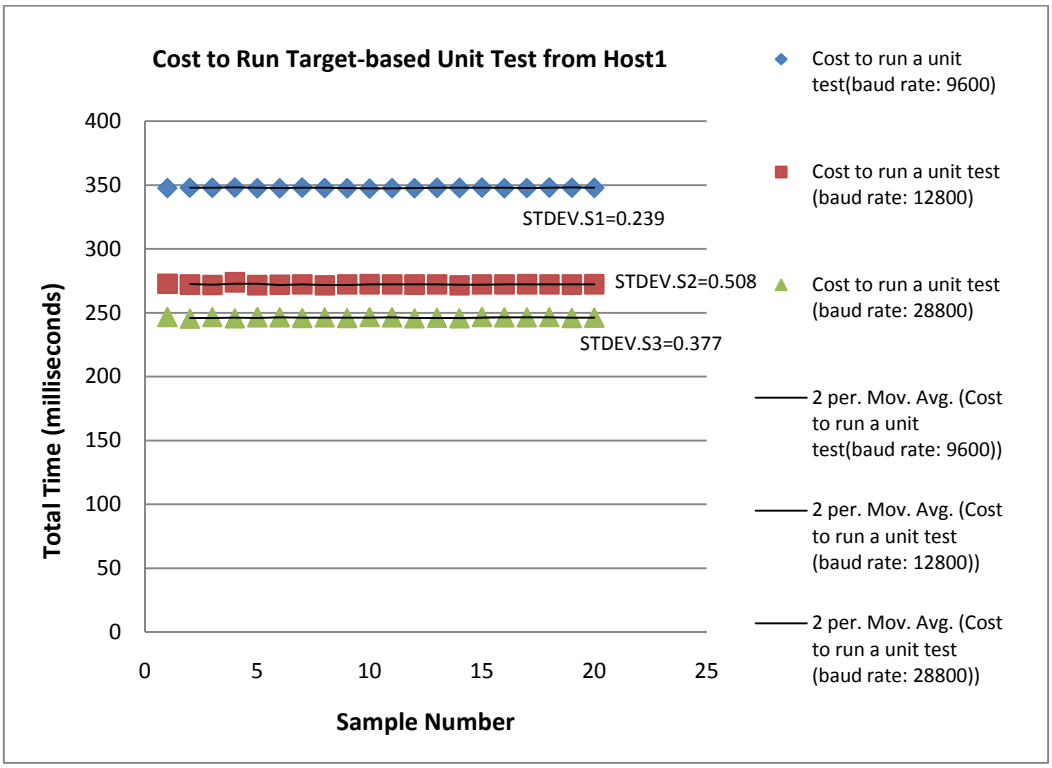

Figure 7 A Sample of Target-based Unit Test Cost from Host1 (Sample Size: 20)

NOTE: STDEV.S1 means the standard deviation for the sample which the baud rate is 9600. STDEV.S2 means the standard deviation for the sample which the baud rate is 19200. STDEV.S3 means the standard deviation for the sample which the baud rate is 28800.

As shown in Figure 7, the cost trend is decreasing moderately as the baud rate goes up. And the standard deviations for each sample group are 0.239 (baud rate: 9600), 0.508 (baud rate: 19200), and 0.377 (baud rate: 28800).

Also, the code execution time on the target system is affected by the target compiler optimization level, which can cause the different costs of unit testing. Table 7 demonstrates the corresponding results when different target compiler optimization levels are set in the experiment.

Table 7 Cost of Target-based Unit Testing from Host1

Baud Rate \ Target Compiler Optimization Level	Average cost to run a unit test (milliseconds)		
	-o0	-os	-o3
9600	347.84	346.88	348.43
19200	272.23	271.66	272.14
28800	246.20	245.37	245.48
NOTE: The compiler optimization mode is set as disabled in Visual Studio 2008. The average data is calculated based on a sample of 60.			

Table 7 shows that the cost of a unit test is generally smaller if the compiler optimization level is set as –os or-o3. The –os or –o3 means the target compiler adopts the specified optimization strategy, while –o0 means no optimization is made.

6.2.2 Results from Host2

Table 8 Cost to Run Target-based Unit Test from Host2

Baud Rate	Total Time(100 times)	Average time
9600	40514 milliseconds	405.14 milliseconds
19200	39686 milliseconds	396.86 milliseconds
28800	39365 milliseconds	393.65 milliseconds
NOTE: The compiler optimization modes are set as disabled in Visual Studio 2008 and –o0 in AVR Studio 4. The average data is calculated based on a sample of 20 as shown in Figure 8.		

Table 8 shows the average costs to run a target-based unit test are about 405.14 milliseconds (baud rate: 9600), 396.86 milliseconds (baud rate: 19200), and 393.65 milliseconds (baud rate: 28800) respectively.

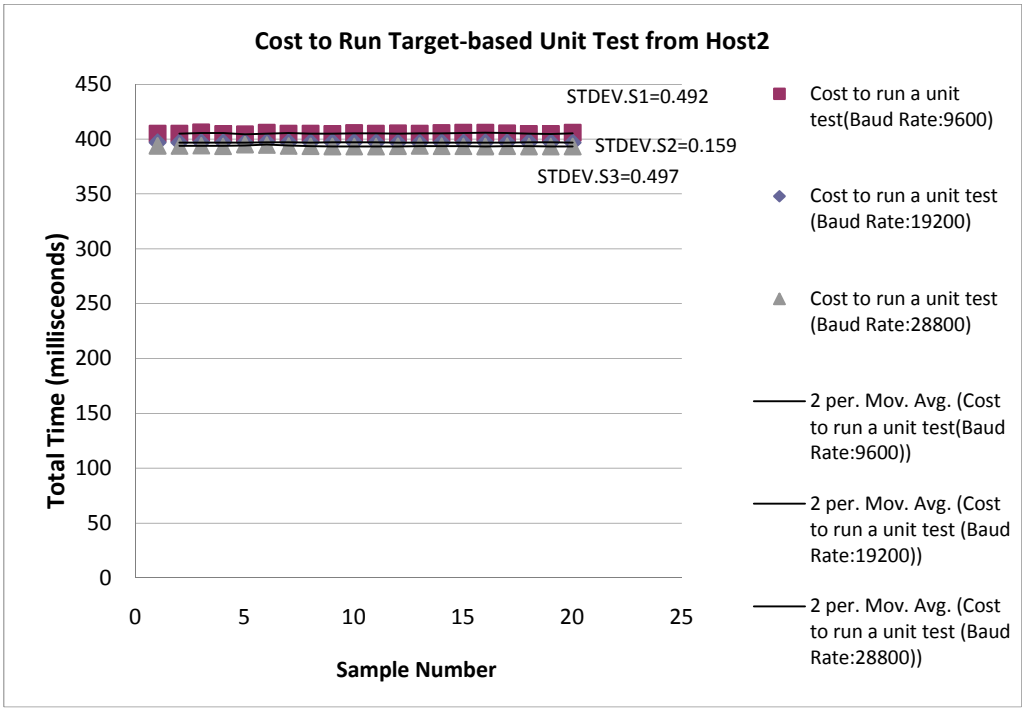

Figure 8 A Sample of Target-based Unit Test Cost from Host2 (Sample Size: 20)

NOTE: STDEV.S1 means the standard deviation for the sample which the baud rate is 9600. STDEV.S2 means the standard deviation for the sample which the baud rate is 19200. STDEV.S3 means the standard deviation for the sample which the baud rate is 28800.

The standard deviations for each sample group from host2 are 0.492 (baud rate: 9600), 0.159 (baud rate: 19200), and 0.497 (baud rate: 28800). And as shown in Figure 7 and Figure 8, the standard deviations of the host2 sample are generally bigger than that of the host1 sample except for the group (baud rate: 19200).

Table 9 Cost to Run Target-based Unit Test from Host2

Baud Rate \ Target Compiler Optimization Level	**Cost to run a unit test (milliseconds)**		
	-o0	**-os**	**-o3**
9600	405.14	405.04	404.96
19200	396.86	396.82	396.81
28800	393.65	393.87	394.43
NOTE: The compiler optimization mode is set as disabled in Visual Studio 2008. The average data is calculated based on a sample of 60.			

Table 9 demonstrates the collected data when the target compiler optimization level is set as different values. The effects of the target compiler optimization level are minor to the costs. And it seems that the costs are slightly reduced when the optimization level is set as –os or –o3.

The experimental results from host1 are quite different from that from host2. The reasons why different results were gathered from the two hosts will be discussed in the next section.

6.3 Operating System Overhead

The cost to run a target-based unit test can be divided into several parts in the remote test performance model (see Figure 3). As expected, the communication time takes up a majority of the cost. The actual time to run a test on the target system is relatively small, which is measured as about 0.087 milliseconds using a hardware timer on the target system.

The differences in the O/S and hardware give different experimental results on target-based unit testing on host1 and on host2. This is due to the communication mechanism used in the experiment is supported by the .NET framework. This communication mechanism behaves differently based on the O/S platform and hardware, thus affecting the communication time. To verify this, the executing time to run a loop-back program, which tries to loop back messages of different lengths via the serial communication, is recorded.

In the target-based unit testing (see Figure 4), there are two messages sent between the host system and the target system. The test ID message is of four-character length. And the test result message is of two-character length. Taking the loop-back program data on host1 for example, when the baud rate is 9600, the average time to send (or receive) a four-character message and a two-character message are approximately 171.45 milliseconds and 170.69 milliseconds respectively. The total time to send the two messages is 342.14 milliseconds. As shown in Table 6, the average cost to run a unit test is 347.84 milliseconds (baud rate: 9600, optimization level: -o0). The time for the target system to process message and run a unit test is relatively short according to the data measured using the hardware timer.

7 Discussion

7.1 Experimental Results

The baud rate represents the transmission speed in the serial port communication, which influences the communication time. Figure 9 and Figure 10 demonstrate the collected data in the experiment when different baud rates are set, with the predicted cost (we refer cost as the real time as in the previous sections).

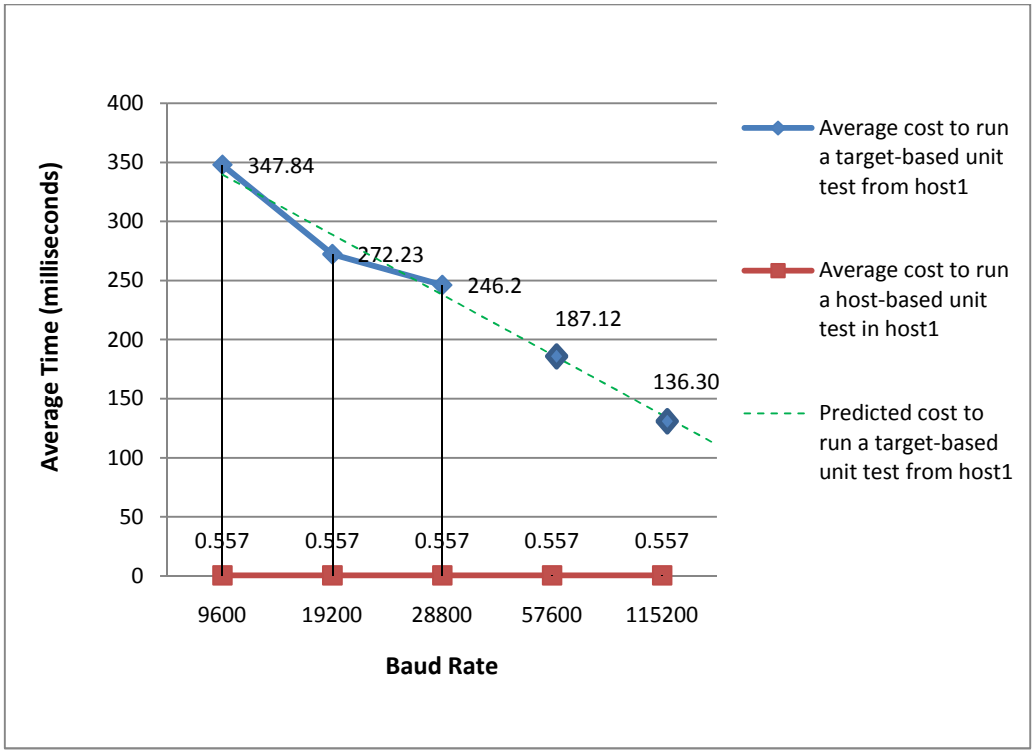

Figure 9 Average Cost of Unit Testing (host1)

NOTE: the compiler optimization levels are set as disable in Visual Studio 2008 and –o0 in AVR Studio 4.

In Figure 9, the trend of the cost from host1 is descending moderately with the baud rate goes up.

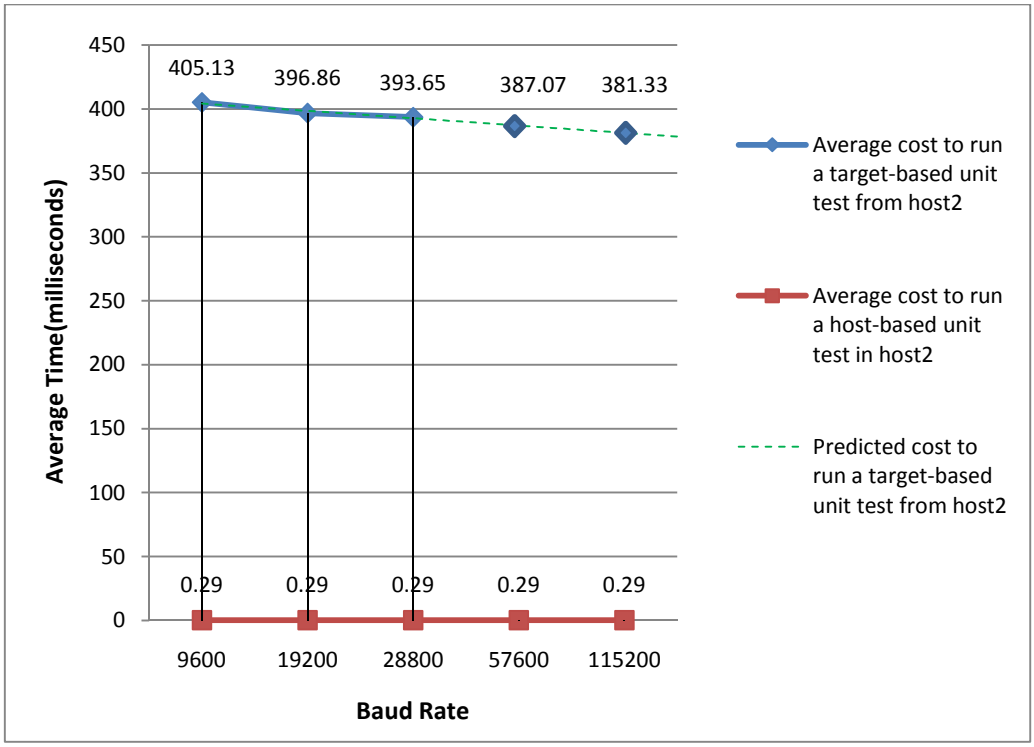

Figure 10 Average Cost of Unit Testing (host2)

NOTE: the compiler optimization levels are set as disable in Visual Studio 2008 and –o0 in AVR Studio 4.

In Figure 10, the trend of the cost from host2 is slightly decreasing at a much slower speed. The predicted cost when the baud rate is set to 57600 or 115200 is also provided. Due to the fact that the target does not support 57600 or 115200 baud, the predicted data cannot be validated in the experiment.

7.2 Performance Results

As the communication time accounts for a large proportion of the cost (execution time), Figure 11 compares the actual results for the communication time with the estimated values. The actual results are based on the data collected from the loop-back program. And the estimated values are calculated according to the relevant theory in section 4.3.

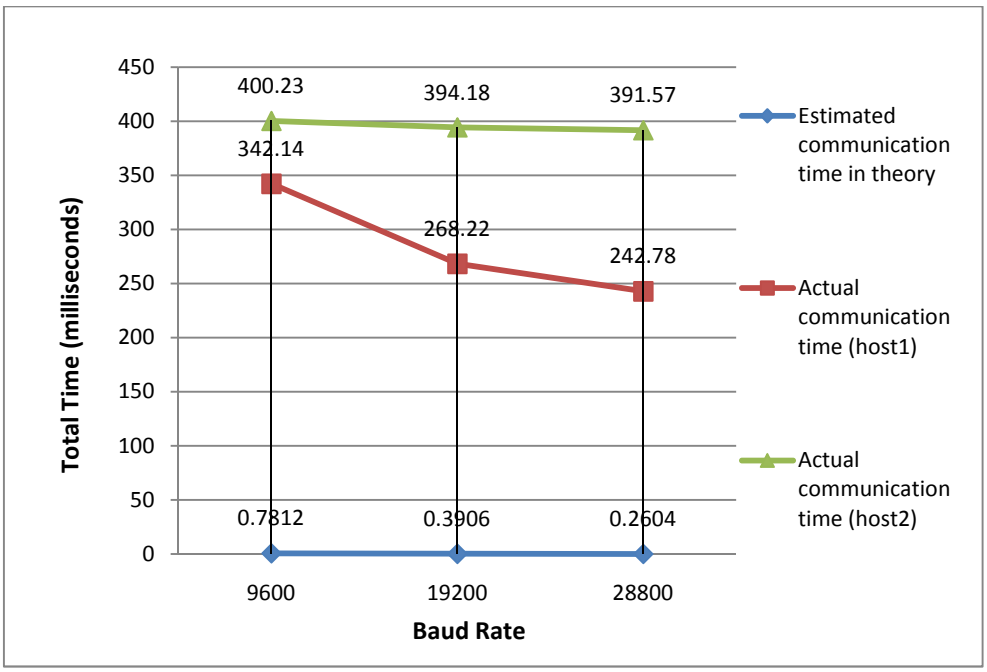

Figure 11 The Actual Results and the Estimated Data about the Communication Time

As shown in Figure 11, the estimated communication time is 0.7812 milliseconds (baud rate: 9600), 0.3906 milliseconds (baud rate: 19200), and 0.2504 milliseconds (baud rate: 28800). While from the loop back program, the actual costs from host1 are 342.14 milliseconds (baud rate: 9600), 268.22 milliseconds (baud rate: 19600), and 242.78 milliseconds (baud rate: 28800). And the actual costs from host 2 are generally more than that from host1. The results are 400.23 milliseconds (baud rate: 9600), 394.18 milliseconds (baud rate: 19600), and 391.57 milliseconds (baud rate: 28800) respectively. Also, the descending trend of the host1 costs is much obvious than that of the host2 costs as the baud rate increases. As discussed in the section 6.3, the gap between the host1 data and the host2 data are due to different O/S and hardware.

7.3 Summary

According to the collected results, the time cost is decreasing as the baud rate increases. Due to the operating system overhead, the experimental results are different from the two hosts. And the actual communication time, which accounts for the majority of the time cost, is at least hundreds times more than the estimated time calculated in theory.

8 Conclusions and Future Work

This paper identifies a variety of reasons for unit testing on a target system, develops a performance model of remote unit testing, and quantifies the corresponding costs. The experimental results shown in this paper demonstrate the feasibility of remote unit testing on a target system. By using a standard unit testing framework (such as CppUnit), a host-based unit test on the target system can be run with little or no changes. The experimental results show that the cost of unit testing on a target system is at least a thousand times greater than on a host system. If the cost to run a unit test on a host system is known, the trade-off on a target system can be estimated and the decision whether to do the corresponding unit test on a target system can be made accordingly.

As it is analyzed, the major cost of remote unit testing lies in the communication overhead. There are possible ways to alleviate this. To reduce communication overhead, instead of running one test case each time, a message could be sent to invoke an entire suite of tests to be run at a time. Also, the target could overlap the communication and test execution to reduce the overall time cost. An additional problem to be addressed is the unreliability of the communication link to the target system. A checksum and retransmissions could be added to the remote unit testing protocol to ensure the reliability of the communication.

REFERENCES

[1] PRLog. “Global Market for Embedded Systems worth $112.5 Billion in 2013”. Internet: http://www.prlog.org/10225881-global-market-for-embedded-systems-worth-1125-billion-in-2013.html, PRLog, Apr. 28, 2009 [Nov. 25, 2010].

[2] S.Greengard. “Making Automation Work” . *Journal of Communication of the ACM*, vol. 52, NO. 12, Dec, 2009, pp. 18-19.

[3] SoftwareEngineeringReferences.Com. “Some Recent Software Failures Caused by Software Bugs”. Internet: http://www.sereferences.com/software-failure-list.php, SoftwareEngineeringReferences, [Nov. 25, 2010].

[4] A. Causevic, D. Sundmark, and S. Punnekkat. “An Industrial Survey on Contemporary Aspects of Software Testing,” in 3rd Int. Conf. Software Testing, Verification and Validation. Paris, France, IEEE CS, 2010, pp. 393-401.

[5] J.E.Cooling, Software Design for Real-time System. London, UK: Chapman & Hall, 1991, p. 433.

[6] Parasoft. “Techniques for Unit Testing Embedded Systems Software”. Internet: http://www.parasoft.com/jsp/products/article.jsp?articleId=2685, Parasoft, [Nov. 19, 2010].

[7] W. Schmitt. “Automated Unit Testing of Embedded ARM Applications.” Information Quarterly. [Online]. vol. 3, Number 4, p. 29, 2004. Available: www.iqmagazineonline.com/.../Pg29_IQ_Hitex_**Automated**.pdf [Nov. 24, 2010].

[8] V.Encontre. “Testing Embedded Systems: Do You Have the GuTs for It?” Rational Edge. Internet: http://www.ibm.com/developerworks/rational/library/459.html#ibm-pcon , Rational Software, Nov. 23, 2003 [Jan. 14, 2011].

[9] H.-ming Qian and C. Zheng. “An Embedded Software Testing Process Model,” in Int. Conf. on Computational Intelligence and Software Engineering. Wuhan, China, IEEE, 2009, pp. 1-5.

[10] M. Smith, J. Miller, L. Huang, and A. Tran. “A More Agile Approach to Embedded System Development”. *Journal of IEEE Software*, vol. 26, issue 3, May, 2009, pp. 50-57.

[11] C.-S. Koong, H.-J. Lai, C.-H. Chang, W.C. Chu, N.-L. Hsueh, P.-A. Hsiung, C. Shih, and C.-T. Yang. “Supporting Tool for Embedded Software Testing.” In 10th Int. Conf. on Quality Software. Zhangjiajie, China, IEEE CS, 2010, pp. 481-487.

[12] S.Brown, CS608 Software Testing, Topic: “’Examples: Car Tax 2 (OOT)+UML” Department of Computer Science, National University of Ireland, Maynooth, Co.Kildare, Ireland, Feb. 5, 2010.

[13] Sourceforge.Net. “CppUnit Wiki”. Internet: http://cppunit.sourceforge.net/doc/lastest/index.html , Sourceforge, Dec. 2008 [Nov. 12, 2010].

[14] Sourceforge.Net. “CppUnit Documentation”. Internet: http://cppunit.sourceforge.net/doc/lastest/index.html , Sourceforge, [Nov. 12, 2010].

[15] A.Jalis. “CuTest The Cutest C Unit Testing Framework”. Internet: http://cutest.sourceforge.net/ , Sourceforge, [Nov. 12, 2010].

[16] Sourceforge.Net. “CUnit Home”. Internet: http://cunit.sourceforge.net/ , Sourceforge, [Nov. 12, 2010].

APPENDIX – Source Code

Test Class: CarTax

```
// Identifier: ie.nuim.cs.cs608.2009-10.CarTax.java, version 1
// Car tax system based on CO2 pollution levels
class CarTax {
  private boolean valid;
  private int yearOfManufacture;
  private int co2Pollution; // in ugrams
  private int tax;

  // create a new CarTax object
  // if valid year, set valid and year of manufacture=year
  // else set !valid
  // initialise co2Pollution and tax to -1
  CarTax(int year)
  {
    if (year>=1900) {
      valid = true;
      yearOfManufacture = year;
    }
```

```
  else {
    valid = false;
  }
  co2Pollution = -1000;
  tax = -1;
}

// set co2 pollution, reading is in mgrams
// only works if valid year
public void setCO2(int reading)
{
  if (valid)
    co2Pollution = reading * 1000;
}

// return co2 pollution in mgrams
public int getCO2()
{
  return co2Pollution / 1000;
}

// return year of manufacture (-1 if invalid year)
public int getManufactureYear()
{
  if (valid)
    return yearOfManufacture;
  else
    return -1;
}
// Inputs:
//   year of manufacture:int (valid=1900->)
//   co2 levels:int (valid=0..1000) - in mgrams
//   newSystem: boolean
//
```

```
  // Specification
  //
  //   If year of manufacture <=2010,
  //      old system: tax=3000
  //      new system: tax=10*co2
  //   Otherwise (new system must be true)
  //      tax=10*co2
  //
  // Output:
  //   return tax
  //   return -1 on any data errors, or not valid year
  public int calculateTax(boolean newSystem)
  {
    if (valid) {
      if ( (yearOfManufacture<=2010) || ( (yearOfManufacture>2010) && (newSystem) ) )
      {
        if (newSystem)
          tax = 10 * (co2Pollution/1000);
        else
          tax = 3000;
      }
    }
    return tax;
  }

  // return the current tax value, and reset to zero
  public int readAndZeroTax()
  {
    int temp = tax;
    tax = 0;
    return temp;
  }

}
```