1-1-2017

Parallel, Cross-Platform Unit Testing for Real-Time Embedded Systems

Tosapon Pankumhang
University of Denver

PARALLEL, CROSS-PLATFORM UNIT TESTING FOR REAL-TIME EMBEDDED SYSTEMS

A Dissertation

Presented to

the Faculty of the Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

by

Tosapon Pankumhang

August 2017

Advisor: Matthew J. Rutherford

Author: Tosapon Pankumhang
Title: PARALLEL, CROSS-PLATFORM UNIT TESTING FOR REAL-TIME EMBEDDED SYSTEMS
Advisor: Matthew J. Rutherford
Degree Date: August 2017

ABSTRACT

Embedded systems are used in a wide variety of applications (e.g., automotive, agricultural, home security, industrial, medical, military, and aerospace) due to their small size, low-energy consumption, and the ability to control real-time peripheral devices precisely. These systems, however, are different from each other in many aspects: processors, memory size, develop applications/OS, hardware interfaces, and software loading methods. Unit testing is a fundamental part of software development and the lowest level of software testing, as it tests individual or groups of functions, methods, and classes, to increase confidence that the developed software satisfies both software specifications and user requirements. Although hundreds of unit testing frameworks exist, none of them address the diverse properties of real-time embedded platforms. This inspires us to introduce XEUnit, a cross-platform unit testing framework for real-time embedded systems. XEUnit provides scalability to the framework by supporting parallel execution on multiple embedded platforms simultaneously.

To address the time constraints in real-time embedded systems, we evaluate the impact of runtime overhead from traditional instrumentation through a case study of time-sensitive algorithms. Then, we introduce iterative instrumentation, which is a code coverage technique without runtime overhead, along with a case study demonstrating the effectiveness of this technique. Although iterative instrumentation can measure code coverage effectively in time-sensitive applications, the total execution cost of this approach is much higher than traditional instrumentation due to the execution of multiple variants of the system under test. This leads to scalability and performance issues especially in large applications. To solve these issues, there are two approaches we use: reducing the number of variants and executing them simultaneously.

To reduce the number of variants, we present cluster iterative instrumentation, a graph clustering technique that can reduce the number of nodes in a control flow graph resulting in lower execution time. We also provide a case study of node coverage of control software to show the efficiency of cluster iterative instrumentation compared to iterative instrumentation. In addition to reducing the number of variants, the other method is to execute multiple variants at the same time. Because all executions are independent from each other, we can use parallel execution on multiple embedded platforms. Thus, we design and implement a parallel unit testing framework for real-time embedded system along with a case study comparing the execution times from different numbers of embedded platforms (executing nodes).

Our final contribution is a cross-platform unit testing framework using the concepts of runtime adapters and a runtime protocol that enables testers to run code across different embedded platforms. We also demonstrate the effectiveness of this framework by testing black-box test cases on seven different embedded platforms. Overall, our results indicate that cluster iterative instrumentation with parallel unit testing can address the scalability and performance issues, and the case studies demonstrate that XEUnit can effectively test the same code on a variety of embedded platforms.

ACKNOWLEDGEMENTS

I would like to thank everyone who has supported and motivated me to complete this dissertation.

First of all, I would like to thank Dr. Matthew J. Rutherford, my advisor, since he has helped me develop as a researcher, problem solver, writer, and presenter for six years at University of Denver. During my PhD program, Dr. Rutherford is always available for me to discuss both academic and family topics. In particular, I would like to thank him for giving me the excellent opportunity to work with Arrow Electronics, and I hope that our collaboration still continues after my graduation here.

In addition, I would like to thank Prof. Anneliese Andrews, my preliminary exam committee, for her valuable guidance especially for the software testing aspects, and I would like to thank all defense committee members: Prof. Alvaro Arias, Dr. Chris Gauthier Dickey, and Dr. Rinku Dewri for their time, support, and feedback. I also would like to thank Susan Bolton, Meredith Corley, and other staff in my department for assisting me to process several documents.

Next, I would like to thank Bin Wang, my manager at Arrow Electronics, including all big data members for their generous support while working with them as a big data intern.

Lastly, I would like to thank my parents for their love and support, and I want to special thank to my beloved wife, Nuananong, and my lovely son, Nawapon, for being the inspiration throughout my education.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Introduction

Software testing is an important part of software development, as it increases the quality of the developed software for stakeholders by reducing errors and evaluating both software verification (i.e. software specifications) and validation (i.e. user requirements) [1]. In this area, unit testing is the lowest level of software testing, since it tests individual or groups of functions, methods, or classes. This has inspired the invention of a number of unit testing frameworks, many of which are based on the xUnit framework (e.g., JUnit, AceUnit, CUnit, CppUnit, Check, embUnit, and Google Test) [2-10]. The xUnit family is an object-oriented framework using a test runner to execute test cases or sets of test cases grouped into a test suite. According to the unit testing survey conducted by Runeson [11], there are additional features unit testing should provide, such as automated testing, repeatable testing, white-box testing (e.g., code coverage), real-time testing, and even cross-platform testing.

Cross-platform testing can be achieved in two ways. The first method is to write a single version with intermediate code and run it on a virtual machine or write scripting languages (e.g., JavaScript, Python, and Ruby) supported by most platforms [12]. For instance, Java code is compiled into bytecode (i.e., intermediate code), and it is executed on a Java Virtual Machine running on top of a host operating system. The other way is to create tools in multiple versions for particular platforms. For example, current cross-platform testing tools (e.g., Mono, Portable. NET, and Microsoft. NET) are able to develop and test code across desktop, mobile, and web applications [12-14] using different versions of code (one for each platform). Although these tools can test applications for many systems, they are not suitable for testing real-time embedded applications due to the heterogeneity and resource constraints (both time and memory) in real-time embedded systems.

Real-time embedded systems are typically used to interact with and control sub-systems or peripheral devices (e.g., sensors, actuators, and other digital/analog devices) within strict time bounds imposed by the peripherals or system goals [15]. However, these embedded systems are different from each other in many aspects. Existing embedded platforms use diverse processors produced by various vendors (e.g., ARM, Atmel, Intel, Microchip, Qualcomm, TI, and XMOS) [16-17]. In addition, embedded software must be developed on different operating system platforms as determined by the development tools (e.g., ARM mbed IDE, Atmel Studio, Microchip MPLAB X IDE, XMOS xTimeComposer, and TI Cloud9) provided by those vendors [18-27]. To load and execute code on particular platforms, non-Operating System (OS) embedded platforms (e.g., ARM, Atmel, Microchip, and XMOS) use their own uploader applications, while OS platforms (e.g., TI Beaglebone, Intel Edison, Qualcomm DragonBoard, and Raspberry Pi) use remote access to execute code on their native real-time operating systems through network protocols [23-27]. Lastly, non-OS embedded platforms usually have small memory footprints, which may not be sufficient to include both the code needed for a general-purpose unit testing framework and the code for the system under test (SUT) in the executable code.

In addition to the diversity of real-time embedded platforms, resource constraints (i.e., time and memory constraints) are the other problems for unit testing frameworks. Time constraints are critical for testing code on these real-time platforms. Hard real-time systems could fail or behave differently from the expected results, if the timing of real-time components is incorrect. Although it is possible to simulate timing or virtual devices, the simulation could be very slow depending on the processing speed and workloads of the system during the test. In particular, if the system under test needs to interact with hardware peripheral devices, it is more difficult to simulate. Thus, real-time applications could be tested on simulator for staging software, but they need to run on actual hardware (including real-time OS and built-in resources for OS platforms) for the production version especially for hard real-time software.

Instrumentation [28-30] is a traditional technique for measuring white-box testing (e.g., code coverage) by inserting additional instructions into the original code to collect relevant data during the test. This technique, however, increases runtime overhead to the SUT possibly resulting in unexpected results. Although researchers try to reduce runtime overhead of instrumentation with various techniques (e.g., runtime monitoring, tracing, and optimizing), none of them are suitable for both black-box and white-box testing in real-time embedded systems [28-34]. Runtime monitoring only supports for white-box testing, while tracing and optimizing may cause runtime overhead that could result in missed timing deadlines.

Memory constraints are also the other issue in embedded platforms, because these platforms usually have limited memory size. In our study, the non-OS platforms have memory sizes between 16KB-128KB [18-21, 23], which are not sufficient for loading code generated by most unit testing frameworks. Boydens et al. discuss two approaches to the use of a unit testing framework for these systems [35]. The first method is to run a testing framework on a host PC with virtual drivers for embedded platforms. The other approach is to use wrapper functions as common interfaces for a program on the host to interact with a program using real drivers on the embedded platform. Although the second method should be appropriate for embedded systems that have small memory footprints, this approach is not designed for cross-platform testing and it does not address the time-sensitive properties of the real-time embedded systems.

With the diverse and resource-constrained properties of real-time embedded systems, we have found no appropriate unit testing frameworks for such systems. These challenges inspire us to introduce a cross-platform unit testing framework for real-time embedded systems. Our framework will enable testers to use a single framework for testing code on different embedded platforms by using “runtime adapters” and a “runtime protocol.” In addition to the framework, we also present a new code coverage technique called “iterative instrumentation” [36] that solves the time-constrained issue in real-time embedded systems. The iterative instrumentation adapts the concept of weak mutation [37-38], a softer definition of the traditional mutation testing or strong mutation [39], for measuring code coverage without runtime overhead. Despite solving the time constraints effectively, the iterative instrumentation technique still has the drawback of expensive execution cost inheriting from mutation analysis. We, therefore, reduce this cost by introducing “cluster iterative instrumentation” that can reduce the number of nodes from the original version resulting in lower execution time. Lastly, we provide the efficiency to our unit testing framework by applying the parallel unit testing concept inspired by the Mateo and Usaola study [40] to the framework.

1.2 Research Questions

As mentioned above, our main research questions focus on a cross-platform unit testing framework suitable for testing code on real-time embedded systems, which have diverse and resource constrained properties. Since this framework needs to be both effective and efficient, there are five main questions for our research.

1) How does the runtime overhead from instrumentation impact the test execution?
2) How can we effectively test time-constrained applications without affecting the test execution?
3) How can we efficiently test code for real-time embedded applications?
4) How can we execute code on memory-constrained platforms?
5) How can we run the same code across different embedded platforms?

To answer the research questions. First, we characterize the impact of runtime overhead from different code coverage criteria using traditional instrumentation techniques through time-constrained applications. To provide an effective technique suitable for time-sensitive applications, we introduce iterative instrumentation for measuring code coverage without runtime overhead. To improve the efficiency of the iterative instrumentation approach, we present the cluster iterative instrumentation technique to reduce the number of variants executed for each test case. Next, to provide scalability to our testing framework, we apply parallel execution to our unit testing framework to execute multiple variants on parallel-embedded platforms. Finally, we present the concepts of runtime adapters and runtime protocol that can execute code on memory-constrained embedded systems and also enable the cross-platform unit testing framework.

1.3 Research Contributions

According to the research questions, we have made several contributions in this dissertation as follows.

1) We characterize the impact of runtime overhead from different code coverage types using traditional instrumentation through a case study of time-sensitive applications.
2) We introduce a time-sensitive code coverage technique called iterative instrumentation that can measure code coverage without runtime overhead. We also show the effectiveness of iterative instrumentation by comparing the code coverage results between this approach and traditional instrumentation.
3) We improve the performance of iterative instrumentation by introducing the cluster iterative instrumentation concept consisting of the cluster rules and cluster algorithm for transforming a control flow graph into a cluster graph. We evaluate the efficiency of cluster iterative instrumentation by comparing the execution times of node coverage testing between this technique and the iterative instrumentation technique.
4) We tackle the performance and scalability issues of cluster iterative instrumentation by applying parallel computing within our unit testing framework. We evaluate the performance of parallel unit testing by comparing

the execution times of both iterative instrumentation and cluster iterative instrumentation on different numbers of parallel-embedded platforms.

5) We invent a cross-platform unit testing framework using runtime adapters and a runtime protocol, which enables testers to test code on any embedded platforms. We demonstrate the effectiveness of our framework by comparing the black-box testing results on seven different embedded platforms.

1.4 Dissertation Organization

In this paper, we discuss all related work and background in Chapter 2. Chapter 3 shows the impact of runtime overhead from traditional instrumentation through the test results on time-sensitive applications (i.e. heuristic pathfinders) [41-43]. Then, we introduce iterative instrumentation, which is a code coverage technique without runtime overhead, along with a case study demonstrating the effectiveness of this technique. Chapter 4 presents cluster iterative instrumentation that can improve the performance of iterative instrumentation with another case study comparing the execution times between these techniques. Chapter 5 proposes parallel unit testing that solves the scalability issue of iterative instrumentation with a case study comparing the execution times from different numbers of embedded platforms (computing nodes). Chapter 6 describes a cross-platform unit testing framework by using the concepts of runtime adapters and a runtime protocol that enable testers to run code across different embedded platforms. This chapter also provides a case study showing the effectiveness of our cross-platform unit testing framework by running the same code on seven different embedded platforms. The last chapter concludes the results and contributions of this paper, and it also discusses possible improvements as our future work.

CHAPTER 2: LITERATURE REVIEW

Based on our research questions, we review the related papers to our topics: instrumentation techniques, parallel unit testing, embedded platforms, and unit testing frameworks as follows.

2.1 Traditional Instrumentation Vs. Iterative Instrumentation

This section discussed two different approaches: traditional instrumentation and iterative instrumentation, used for code coverage testing.

2.1.1 Traditional Instrumentation techniques

Instrumentation is widely used for code coverage testing due to its effectiveness for virtually any type of coverage criterion. With the addition of the instrumentation code, this approach causes inevitable overheads in both memory consumption and execution time. To solve this problem, a number of researchers [28-30, 44-48] introduce techniques to improve the performance of instrumented code. Most reduce the overhead dynamically, whereas some optimize the instrumented code statically.

Dynamic instrumentation [28, 44-47] modifies instrumented code at runtime. Tikir and Hollingsworth [28] utilize the information from a dominator tree and the on-demand instrumented points to reduce both preprocessing time and testing time. The first technique utilizes a dominator tree, which all test paths to a dominated node are always passed its dominators (i.e., previous nodes in the paths), to execute only leaf nodes. With the property of the dominator tree from root to node n, if node n is executed, the other nodes from root to node n can be assumed that they are always executed. Thus, only the leaves in the tree are actually executed, while the others in the paths are omitted. Although the dominator tree can reduce a number of nodes to execute, there are some nodes uncovered by this technique need to be separately executed by the test cases. The other technique uses on-demand instrumented points that only the called functions are instrumented and all executed instrumented points are removed before the next test at runtime. This technique can reduce the preprocessing time to generate graphs for the uncalled functions and reduce the execution time of the removed instrumented points.

Kumar et al. [44] optimize instrumented code by combining similar function calls of the instrumented points into a single instruction, eliminating unnecessary context switch of each function call (i.e., store and load related registers), and inserting the smallest workload (function) during execution. Santelices and Harrold [45] enhance a runtime monitor to reduce the cost of definition-use (def-use) coverage that a test case can satisfy this by reaching both the definition of a variable and the use of that variable along the test path in the correct order (i.e., the definition node and the use node respectively). Because the cost of def-use coverage is much more expensive than branch coverage, they try to use branch coverage data to estimate def-use coverage rather than directly evaluating it. This approach consists of both static and dynamic processes: evaluating the possibility using branch coverage data for referring def-use coverage and monitoring branch/def-use coverage at runtime. If the order of the def-use coverage is correct (i.e., the definition is reached before the use statement), it is possible to measure the def-use coverage using the branch coverage data. However, some def-use coverage cannot be estimated by the branch coverage (i.e., the order of the def-use coverage is not guaranteed), so this still needs the def-use coverage monitoring directly resulting in an expensive cost. Misurda et al. [46] use on-demand-driven instrumentation by modifying and testing the code and allowing a tester to adjust the instrumented code during the test. Chilakamarri and Elbaum [47] use coverage profiling to indicate unnecessary instrumented probes and utilize Java Collector to remove those probes at runtime.

Static instrumentation transforms source code into instrumented code before executing. Laurenzano et al. [48] statically improve instrumented binary code. Their technique relocates and transforms the binary code for each function to allow a testing program to access the called functions efficiently. It also applies two additional techniques: instrumentation snippet and instrumentation in-lining to reduce runtime overhead. The instrumentation snippet inserts a small, fast assembly code instead of a large, slow instrumentation function, while instrumentation in-lining replaces a block of code at each function call location.

Although these advanced instrumentation techniques are able to reduce runtime overhead of the traditional instrumentation techniques, they still may induce timing issues for time-sensitive operations as instrumentation has additional execution time from the remaining instrumented code. Fischmeister and Lam [30] reduce instrumented code not to exceed the deadline of real-time embedded systems. However, this approach cannot guarantee the minimized version will always fit to all hard real-time deadlines. If the instrumented code is large, the reduced code may still be too large for the available time. Another solution is to extend the deadline of the testing code to ensure the runtime overhead will not interfere the execution of testing code. However, in the general case, it is not possible to extend deadlines arbitrarily as many timing deadlines are driven by requirements of interacting with external components and/or their timeouts and timing constraints.

2.1.2 Iterative Instrumentation Technique

In traditional mutation, faults are simulated to measure test adequacy by replacing a single statement with a syntactic difference to simulate faulty code; each single-statement variant is called a mutant [39]. If test cases are well designed, a mutant should be detected or “killed” when it produces different output than the original, unmodified code. On the other hand, if the test cases cannot detect that mutant, the mutant survives. The “mutant score” is the proportion of the killed mutants out of all tested mutants.

Weak mutation [37-38] is a softer definition than the traditional mutation or strong mutation. While strong mutation needs to propagate an error (i.e., for a mutant to be killed, it must result in output that is different from the original program), weak mutation only requires the detection of different program states. As weak mutation does not require output propagation, the test can be terminated earlier than when performing strong mutation. Howden [37] verifies program states after the execution point of each mutant. Girgis and Woodward [38] use a runtime monitor to track variable states for determining weak mutation detection and collecting data flow coverage simultaneously.

Due to the runtime overhead in the instrumentation techniques discussed in section 2.1.1, we adapt the concept of weak mutation for measuring code coverage without additional runtime overhead. Although mutation is mainly used for assessing test suite adequacy, it has been shown that weak mutation subsumes other test adequacy criteria so the technique can also be applied for code coverage when each mutant represents a unique location reached during the test [29, 51-52]. Since a mutation technique only alters one statement at a time, its execution time before that location is reached is identical to the original code.

Although adapted from weak mutation, iterative instrumentation is different in two important ways. First, we use a single operator (i.e., an exit statement) rather than a set of syntactic differences. This always terminates the execution after reaching a mutation statement. Second, the use of the exit operator means that state-change checking is not performed in our technique, as it is unnecessary for code coverage measurement. Additionally, the goal of iterative instrumentation is to measure code coverage instead of being a test adequacy technique (i.e., mutation analysis qualifies a test suite as being adequate when a sufficient number of mutants are killed). We call our technique "iterative instrumentation" to avoid confusion with mutation analysis and to differentiate with traditional instrumentation. By keeping track of which program variants exit for each test case, we are able to determine which test requirements (i.e., coverage locations) are met, and therefore derive overall coverage. The details of iterative instrumentation will be described in Chapter 3.

Although iterative instrumentation can evaluate white-box testing (e.g., code coverage) without runtime overhead, this approach could have the efficiency issue due to the execution of multiple variants on each test case (one for each variant). We, therefore, introduce "cluster iterative instrumentation," which is more efficient than the iterative instrumentation concept. The idea of cluster iterative instrumentation is to group the number of nodes never executed by the same test case. The more nodes are grouped in the cluster, the fewer variants are executed. We present cluster iterative instrumentation with cluster rules and algorithm, which can transform a control flow graph into a cluster graph, in Chapter 4.

2.2 Parallel Computing for Mutation Analysis

Although iterative instrumentation and cluster iterative instrumentation can solve the time-constrained issue in real-time embedded systems, the major drawback of these approaches is the expensive execution cost resulting from multiple executions of variants. If the number of variants increases, the total execution time could increase more than linearly.

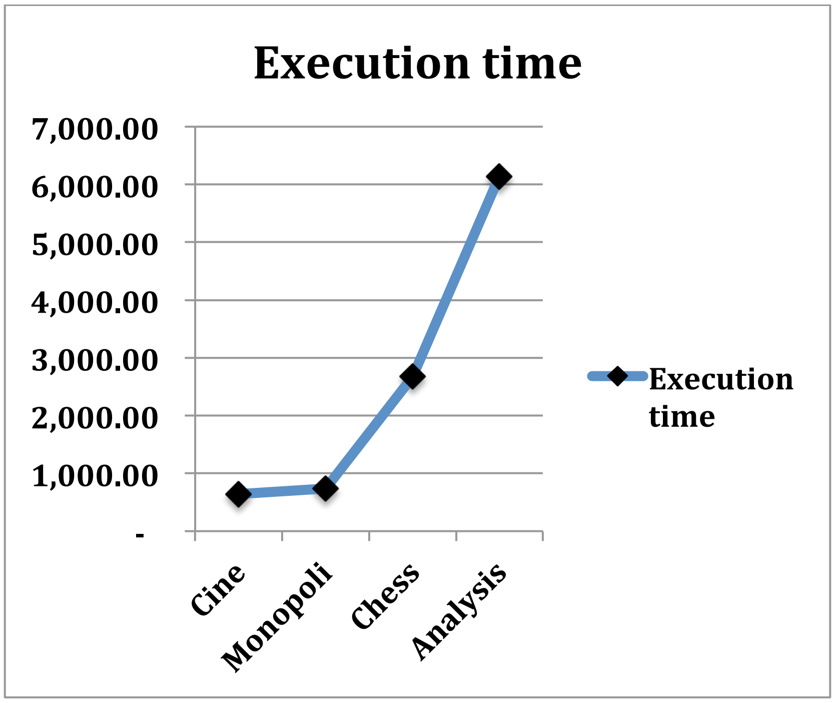


Figure 1. The trend of execution times when the number of mutants increases (from [40])

Mateo and Usaola [40] study the impact of the increasing number of mutants to the total execution time for mutation analysis. Figure 1 shows that the execution time grows exponentially with the number of mutants and test cases. This can lead to scalability and performance issue for mutation analysis when the testing applications are larger. In their work, they try to improve the efficiency of this testing technique by using parallel computing on multiple mutants. In the experiment, they evaluate the execution times when the number of nodes (computing machines) increases.

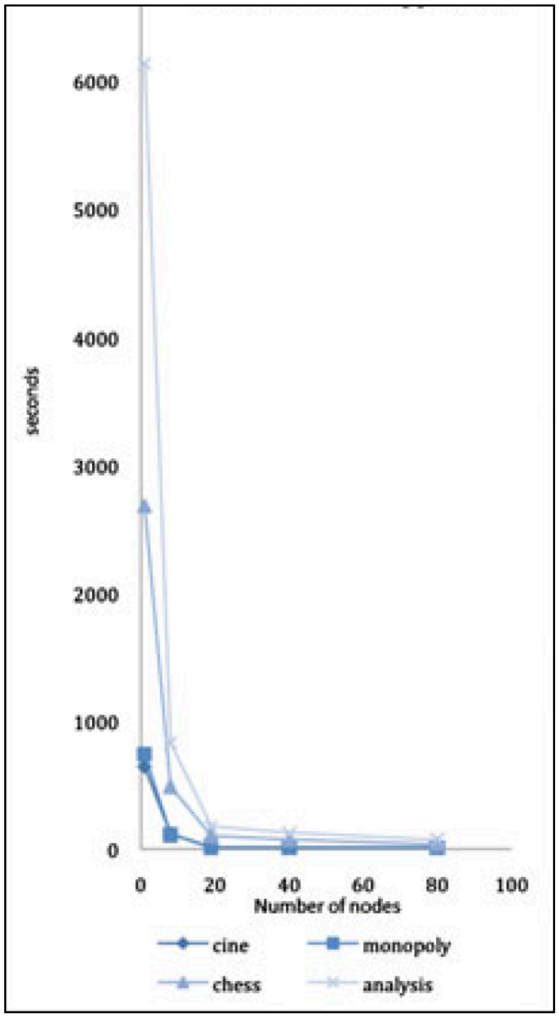


Figure 2. Total times versus number of parallel nodes (from [40])

Figure 2 shows the trend of the execution times when the number of parallel nodes increases. From the graph, the execution times are extremely dropped when the number of executing nodes increases to 20 - 40, and they are not much different after that point. This implies that we are unnecessary to use too many parallel nodes to receive an optimal performance. In this case, using only 20 - 40 nodes is sufficient for reducing the expensive execution cost from mutation analysis.

In addition to the parallel mutation testing, many researchers [58-59, 68] try to use similar approaches to reduce the mutation analysis cost. Krauser and Mathur [58] propose the idea of executing multiple mutants with a vector processor machine (SIMD). However, this study is limited on the scalar variable replacement (SVR) operator, and the mutants are only executed on the same machine. Choi and Mathur [68] and Offutt et al. [59] use a MIMD machine to execute mutants at the same time. Choi and Mathur use dynamic distribution algorithm to send a mutant to execute whenever a processing unit is available, while Offutt et al. use different distribution algorithms (FIFO, random, and uniform distribution) to select mutants to execute. According to these works, the MIMD machines can improve the performance of the mutation analysis, but the transmission time is the bottleneck of the system due to the slow network used in the tests.

Base on our research in parallel computing for mutation analysis, parallelism can improve the performance of the mutation analysis by executing multiple mutants at the same time. Because the generated mutants are parallelizable, these mutants can be executed on multiple machines simultaneously. Although none of these researchers use parallel computing for embedded systems, we can apply the concept of parallel computing using embedded platforms instead of personal computers (PCs). As the costs of most embedded platforms are very low compared to the PCs, it is possible to use a cluster of embedded platforms for testing variants in our unit testing framework. Thus, we design and implement a parallel unit testing framework for real-time embedded system in Chapter 5.

2.3 The Diversity of Embedded Platforms

Since various vendors have recently produced a large variety of embedded platforms, there are many different characteristics causing problems for testing code across these platforms.

Table 1. Comparison of five major characteristics on eight different embedded platforms

Embedded Platforms	Processors	Memory	Develop tools	Hardware interfaces	Loading methods
ARM mbed LPC1768	ARM Cortex-M3 100MHz	64 KB	ARM mbed IDE	mini USB	copy and reset
Atmel AVR AT32UC3L	AVR AT32UC3L064 50MHz	16 KB	Atmel Studio	USB to JTAG	atprogram
XMOS XK-1A	XMOS xCore 80MHz	64 KB	xTime Composer	USB to JTAG	xrun
Microchip PIC32MX79	PIC32MX795F512L 80MHz	128 KB	MPLAB IDE	mini USB	mphidflash
Beaglebone Black	ARM Cortex-A8 1GHz	512 MB	N/A	Virtual Ethernet	SCP, SSH
Raspberry Pi 3	ARM Cortex A53 1.2GHz	1 GB	N/A	WIFI, Ethernet	SCP, SSH
Intel Edison	Intel Atom 500MHz	1 GB	N/A	WIFI	SCP, SSH
Qualcomm DragonBoard 410c	ARM Cortex A53 1.2GHz	1 GB	N/A	WIFI, Ethernet	SCP, SSH

Table 1 compares five major characteristics: processors, memory size, developing tools, hardware interfaces, and software loading methods, on eight different embedded platforms. The first four rows are non-OS platforms (i.e., ARM mbed LPC1768 [18], Atmel AVR AT32UC3L [19], XMOS XK-1A [20], and Microchip PIC32MX79 [21]), while the last four rows (i.e., Raspberry PI 3 [22], TI Beaglebone Black [23], Intel Edison [24], and Qualcomm DragonBoard 410c [25]) are OS-based platforms or OS platforms.

Processors used in embedded platforms are varied as different vendors usually use their own architectures. For example, in the second column of Table 1, there are seven types of processors used by eight embedded platforms. Because these processors often need specific compilers to compile source code into their native code, developers have to use several compiler tool chains to cross-compile one application into different versions in order to run on different hardware architectures.

Low available memory is another problem for most testing frameworks (e.g., the xUnit frameworks), as they assume the platform has sufficient memory to run the framework and the code for the SUT. In the third column of Table 1, the OS embedded platforms have between 512MB – 1GB available memory so there is no issue for running large programs. However, the non-OS platforms usually have limited memory footprints between 16KB – 128KB and cannot load large executable code generated by most testing frameworks to run on them.

Developing tools are also different, because developers may prefer to use their favorite tools and the technologies used in their tools could require specific operating systems, e.g. tools based on Microsoft .NET typically run on Windows systems. In the fourth column of Table 1, non-OS platforms use different tools for developing code for particular platforms, while OS platforms do not provide specific tools to developers. For example, Atmel Studio only supports Windows systems as it is based on Microsoft Visual Studio [19], while xTimeComposer and MPLAB IDE supports on all popular operating systems (i.e., Windows, OS X, and Linux) [20-21].

Hardware interfaces on particular embedded platforms are also varied depending on the manufacturer designs. In the fifth column of Table 1, most non-OS platforms basically provide USB and JTAG interfaces, while OS platforms provide network interfaces: virtual Ethernet (Beaglebone Black), WIFI (Intel Edison), and WIFI/Ethernet (DragonBoard and Raspberry Pi). Developers can use network protocols to communicate with OS platforms over the network interfaces. However, they may need additional adapters (i.e., programmers or debuggers) to connect and load code to some non-OS platforms; for example, Atmel AVR and XMOS require JTAG programmers with their development boards [19-20].

Loading methods for embedded platforms can be classified into two groups: non-OS platforms and OS platforms. The former group usually uses uploader applications provided by vendors to upload executable code to run on the platforms, while the later group mostly running applications on Linux-based real-time operating systems can remotely load code to the systems through network protocols. For instance, in the sixth column of Table 1, most non-OS platforms use different uploader applications (e.g., atprogram, mphidflash, and xrun) to flash and execute code on their platforms. On the other hand, all OS platforms simply use SCP to copy executable code to the target systems and use SSH to remotely execute that code on the targets.

With various processors, memory size, developing tools, hardware interfaces, and loading methods, embedded applications are usually developed and tested on their own tools provided by the vendors. As a result, developers and testers are limited to use particular tools for developing and testing code on different embedded platforms, which hinders switching between embedded platforms and increases training time for new developers.

2.4 Unit Testing Frameworks for Real-time Embedded Systems

Runeson analyzes unit testing in practice by interviewing practitioners from 50 companies. In this survey, the majority of individuals require automated and repeatable unit testing tools, but these features may cause the test results on the tools to be larger while it is testing repeatedly. More importantly, they expect testing tools to evaluate the structure of the code (white-box testing), but most tools only verify the functionality of the unit (black-box testing). Additionally, since many tools are designed for general-purpose application systems, they may not address the timing of real-time components, which is critical for them to function properly. With the issues in unit testing frameworks, many embedded system developers prefer to use their own tools rather than modern unit testing tools [11].

Most recent unit testing frameworks are based on xUnit family [2-10]. Originally invented by K. Beck for Smalltalk called SUnit, these tools are widely used across multiple languages. Our study focuses on unit testing frameworks for C, as this is the common language for virtually all embedded platforms. Because xUnit family is an object-oriented framework, most unit testing tools for C are written in C++ or Java [5-8] (e.g., CppUnit, Google Test, AceUnit, and Check) and some of them are written in C [9-10] (e.g., CUnit and embUnit). Additionally, since most unit testing tools based on xUnit frameworks can only run on OS platforms, testers are forced to test code on a PC with these tools instead of the embedded platforms that may need to interact with real devices (e.g. sensors, actuators, and compasses) during the test.

While most tools usually test code on a host computer, only AceUnit, embUnit, and Boydens et. al's work aim to test code on memory-constrained systems especially non-OS embedded platforms having limited memory sizes [5, 10, 35]. Ace Unit and embUnit [5, 10] limit the functionality of their frameworks to generate an executable file fit to the limited memory footprints of memory-constrained platforms, whereas Boydens et. al [35] use two-tier architecture consisting of a host unit and a test unit. The host unit runs on a PC having sufficient resources, while the test unit runs on the real embedded platform; these modules communicate through a serial communication during the test. However, none of these frameworks addresses the timing of real-time software. Therefore, the runtime overhead increased to the testing code may lead to incorrect test results or may cause the software to behave differently during testing than in production for time-sensitive systems as discussed in [36].

Following our research on recent unit testing frameworks for C, we cannot find an appropriate one for real-time embedded systems. This inspires us to introduce XEUnit, a cross-platform unit testing framework for real-time embedded systems, along with cluster iterative instrumentation used for measuring code coverage in time-sensitive applications introduced in Chapter 5. The details of XEUnit are discussed in Chapter 6.

CHAPTER 3: ITERATIVE INSTRUMENTATION

In software testing, the quality of test suites is commonly estimated through measuring the proportion of test adequacy criteria (i.e., code coverage) satisfied by the tests. Code coverage is a white-box testing technique defining the degree of test adequacy criteria (e.g., coverage percentage), to ensure the developer has an adequate set of test cases. It is assumed that the higher the coverage, the lower the chance of latent bugs being released. In structural testing, there are a number of well-known test adequacy criteria, such as node, edge, logic, function, data flow, and path coverage [51, 53]. Each node in a control-flow graph is a single statement or a block of statements, and each edge is a pair of nodes of a conditional statement. Logic coverage relates to clauses in conditional expressions, and each conditional expression may have multiple clauses connected by logical operators. Among these criteria, most code coverage tools [2] (e.g., Clover, EMMA, and Gcov) support node, edge, and function coverage, while some tools (e.g., Semantic Design) evaluate more complicated criteria like logic and path coverage. In practice, the functionality of these code coverage tools is based on instrumentation.

In order to determine which coverage point is executed at runtime, the code is instrumented with additional logic that tracks the dynamic program flow. As instrumentation inserts instructions into the original code, it introduces runtime overhead during execution. In most cases the effect of this is negligible, so most unit testing frameworks do not address this overhead. However, in time-sensitive systems, it can impact the performance of the applications or alter the timing of executing code. This runtime overhead may lead to inconsistent test results, as the tests behave differently (in time) when run against non-instrumented code. Specifically, if the actual time it takes for a section of code to execute is used to determine subsequent behavior, then changing the runtime performance may significantly alter the path a test case follows when run without the extra instrumentation code. For example, a servo motor will stop when a PWM pulse width is equal to 1.5 ms. With instrumented code, the runtime overhead can change the timing of a servo driver resulting in a minor movement of the servo instead of a neutral position.

To characterize the impact of runtime overhead from traditional instrumentation, Case Study 1 evaluates the test results of time-sensitive applications between instrumented code and original code especially when the types of coverage criteria are different. To eliminate the runtime overhead caused by traditional instrumentation techniques, we introduce iterative instrumentation. Next, we evaluate the effectiveness of iterative instrumentation through Case Study 2 which compares the code coverage results between the traditional instrumentation and iterative instrumentation techniques. Finally, we discuss the trade-off analysis of our technique and possible improvements.

3.1 Case Study 1: The Impact of Runtime Overhead to Time-sensitive Software

To demonstrate the impact of runtime overhead from traditional instrumentation techniques, we compare the test results between instrumented code and normal code on three heuristic pathfinder algorithms: A-star (A*), weighted A-star (WA*), and anytime weighted A-star (AWA*). These search algorithms are widely used in embedded applications (e.g., an autonomous robot can use a heuristic pathfinder algorithm to rapidly find a shortest path in order to move to the destination).

3.1.1 Heuristic Pathfinders

Although there are many search algorithms to find optimal solutions, some techniques may not find any solution within a limited time. Heuristic search allows a trade-off between the quality of a solution and the search time by returning a sub-optimal solution sooner than traditional versions. In this section, we compare the performance among three heuristic pathfinders: A-star (A*), weighted A-star (WA*), and anytime weighted A-star (AWA*) to find the shortest path from a start node to a goal in a grid maze. Additionally, we evaluate the effect of instrumentation code coverage to the quality of results when testing these algorithms in various time constraints.

3.1.1.1 A-star (A)*

A* [41] uses a best-first search to find the shortest path from a start node to a goal. The cost from the start node to the goal called f(n) is computed from two parts: an actual cost from the start node to a current node called g(n) and the estimated cost from the current node to the goal called h(n). In our case, the Manhattan distance heuristic (distance between two points in the grid) is used to estimate the value of h(n).

A* contains two sets: an open set storing nodes to be explored and a closed set keeping visited nodes. Initially, the close set is empty and the start node is inserted into the open set. For each iteration, the algorithm removes a node n having the lowest f(n) from the open set and adds that node into the closed set. Then it explores a non-visited node n’ (a neighbor of n) and calculates f(n’) before inserting n’ into the open set. Note that g(n’) = g(n) + c(n, n’) where c(n, n’) is a real cost from n to n’. The algorithm keeps exploring nodes from the open set until it reaches the goal. As each visited node has a reference to its parent, the shortest path can be backtracked from the visited nodes in the closed set.

3.1.1.2 Weighted A-star (WA)*

WA* [42] attempts to reduce the search space of A* by multiplying a weight value greater than 1 to the estimated cost h(n). Given a high weight, the solution could be found quickly as WA* with weight > 1 is pulled towards the goal with fewer node expansions than A*, however the result may be sub-optimal.

Although WA* and A* are almost identical, there are some differences between these algorithms. First, the heuristic cost h(n) of WA* is multiplied with a weight greater than 1 (w > 1), while the cost h(n) of A* has no weight (w = 1). Additionally, the value of g(n) of WA* could be greater than the optimal value of g(n) of A* as it explores only part of the search space compared to A*.

3.1.1.3 Anytime Weighted A-star (AWA)*

AWA* [43] aims to improve the results of WA* to achieve the optimal solution of A* as long as there is time available. While WA* terminates a program after finding a solution, AWA* keeps searching a better one until an optimal solution is found or the time limit is reached. AWA* can be terminated anytime by timeout or interrupt and returns the best solution found up to that time. In addition to the open set, AWA* uses an inconsistent set to store the latest solution, which is its current best solution. As a better path from a new node found, the new lower cost is updated and the node will be moved from the closed set back into the open set resulting in multiple expansions of that node.

AWA* algorithm initially finds a sub-optimal solution with the highest weight. As long as the weight is greater than 1 and the time is still available, it decreases the weight and re-explores a better solution from the current open set. If there is sufficient time to search until the weight is equal to 1, it will return the same solution as A* or WA* having the weight equal to 1.

3.1.2 Evaluation Setup

In order to evaluate the trade-off between the quality of a solution (i.e., the optimal solution or the shortest path) and the search time (limited time), our evaluation setup is as follows.

- There are four different versions of code: non-instrumented code, node coverage code, edge coverage code, and logic coverage code.
- There are four different search times: 20ms, 25ms, 35ms, and 50ms.
- The A* has no weight or equal to the weight of 1 for the WA*.
- The initial weight of WA* and AWA* is 8. As the search time available, the weight of the AWA* will be decreased by half (i.e., 8, 4, 2, and 1 respectively) after obtaining a solution, while the weight of WA* is fixed.
- The testing platform is XMOS XK-1A that uses a 32-bit 400 MIPS MCU, and the testing times are measured by a real-time clock.

Although there are many types of code coverage criteria, we select three of them: node, edge, and logic coverage (i.e., logic coverage is also know as Correlated Active

Clause Coverage or CACC) [69], as they are sufficient to evaluate the impact of runtime overhead to time-sensitive applications. Node and edge coverage are supported by most code coverage tools [2], and the CACC is one of the logic coverage that a minor clause must cause a predicate to be true for one value of a major clause and false for the other value. Because CACC is similar to Modified Condition/Decision Coverage (MCDC) required by the US Federal Aviation Administration (FAA) for safety-critical applications [70-71], we select this logic coverage criterion in our case study.

3.1.3 Test Results

We compare the performance of heuristic pathfinders by setting four different weights: 1, 2, 4, and 8 for WA* and initialize the highest weight of 8 for AWA*. The AWA* weight is decreased by half every iteration (i.e., after a solution obtained) until it is equal to 1. We also test these algorithms in a range of timeouts as the limited time could affect the quality of these heuristic search results.

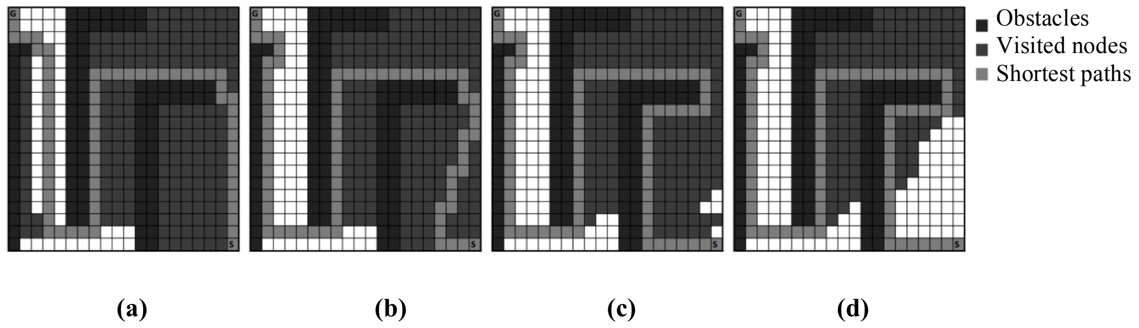


Figure 3. Compare WA* search results from different weights: (a) weight =1, length = 65 (b) weight = 2, length = 73 (c) weight = 4, length = 77 (d) weight = 8, length = 77

The results of WA* depends on its weight, while the results of AWA* depends on both weight and available time to recompute a better solution. If the weight is greater than 1, the results of WA* and AWA* could be sub-optimal solutions compared to what

A* finds. When the weight is equal to 1, they return the same solution as A* (i.e., the optimal solution).

Figure 3 shows the search results of WA* using different weights: (a) weight of 1, (b) weight of 2, (c) weight of 4, and (d) weight of 8. From the results, the lower weight gives a better solution (lower path length) than a higher weight, as it explores more nodes than the high-weight algorithm. However, the trade-off of the low-weight algorithm is the inefficient performance (more searching time). Please note that these results are computed without a time constraint. Therefore, the results will be different when there is insufficient time to find an optimal solution as shown in the next experiment.

Table 2. Code coverage results: (a) no coverage (b) node coverage, (c) edge coverage, and (d) logic coverage for A*, WA*, and AWA* with various timeouts: 20, 25, 35, and 50 ms respectively

	20 ms		25 ms		35 ms		50 ms	
	Length	**Time**	**Length**	**Time**	**Length**	**Time**	**Length**	**Time**
A*	65	9.07	65	9.07	65	9.07	65	9.07
WA*	77	6.20	77	6.20	77	6.20	77	6.20
AWA*	77	timeout	73	timeout	65	31.05	65	31.05

(a)

	20 ms		25 ms		35 ms		50 ms	
	Length	**Time**	**Length**	**Time**	**Length**	**Time**	**Length**	**Time**
A*	65	9.17	65	9.17	65	9.17	65	9.17
WA*	77	6.31	77	6.31	77	6.31	77	6.31
AWA*	77	timeout	73	timeout	65	31.55	65	31.55

(b)

	20 ms		25 ms		35 ms		50 ms	
	Length	**Time**	**Length**	**Time**	**Length**	**Time**	**Length**	**Time**
A*	65	9.20	65	9.20	65	9.20	65	9.20
WA*	77	6.34	77	6.34	77	6.34	77	6.34
AWA*	77	timeout	73	timeout	65	31.70	65	31.70

(c)

	20 ms		25 ms		35 ms		50 ms	
	Length	**Time**	**Length**	**Time**	**Length**	**Time**	**Length**	**Time**
A*	-1	timeout	-1	timeout	65	27.32	65	27.32
WA*	77	18.26	77	18.26	77	18.26	77	18.26
AWA*	77	timeout	77	timeout	73	timeout	73	timeout

(d)

As instrumentation code coverage overhead depends on the details of each coverage criterion, we observe the effect of this overhead through different timeouts. We test heuristic pathfinders in a range of timeouts: 20, 25, 35, and 50 ms, and compare the results between non-instrumented code and basic code coverage criteria: node, edge, and logic coverage (i.e. CACC).

Table 2 compares the results of heuristic pathfinders: A*, WA*, and AWA* in a range of timeouts: 20, 25, 35, and 50 ms. The results in (a) are from executions of non-instrumented code (no coverage) and the other results are from code instrumented for the basic instrumented code coverage criteria: (b) node, (c) edge, and (d) logic coverage.

As expected, the results show that instrumented code increases the searching times for all algorithms. For example, when the timeout is 35 ms, A* searching times are 9.07 ms, 9.17 ms, 9.20 ms, and 27.32 ms for no coverage, node coverage, edge coverage, and logic coverage respectively. Note that the length of -1 means the algorithm cannot find a solution within a given time and the "timeout" value of AWA* means the algorithm cannot find an optimal solution (i.e., it only finds a sub-optimal solution). They also show that the more complicated the coverage, the more additional time needed for execution (time logic > time edge > time node). In particular, the impact of logic coverage is much more than node and edge coverage resulting in different search paths as the algorithm executes. For instance, when the timeout is 25 ms the path lengths of AWA* are 73, 73, and 77 for node, edge, and logic coverage respectively.

The search results with different time constraints indicate that instrumentation runtime overhead affects the solution quality of these heuristic pathfinders, since the additional instructions must be executed, and therefore slow down the algorithms. In particular for the AWA*, this impact clearly results in a worse solution being obtained. This further implies that instrumentation can significantly impact the runtime result and alter the execution path when run on time-sensitive systems.

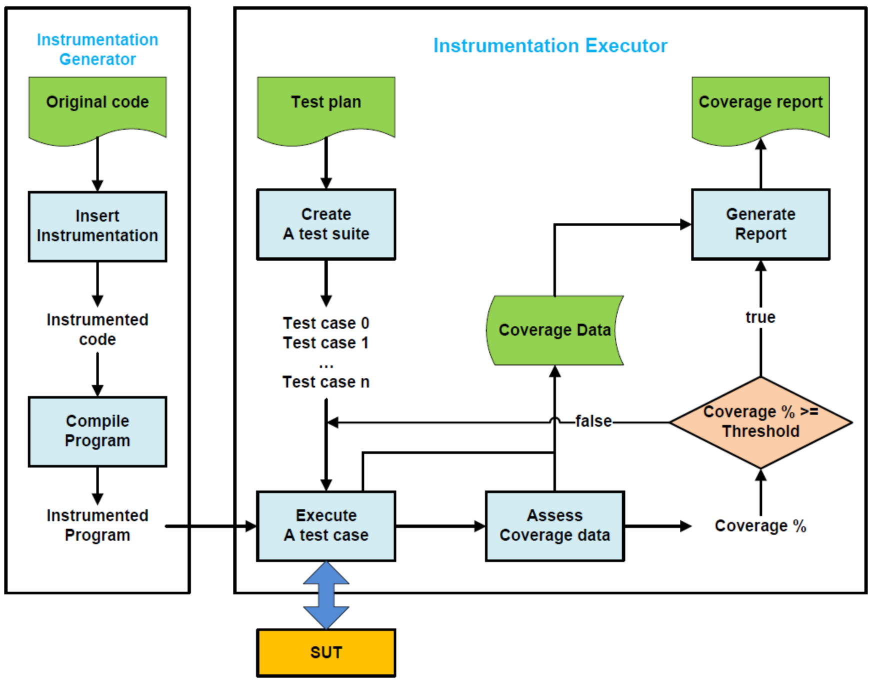
Although there are alternate techniques that can deal with runtime overhead; for example, we can reset the clock measuring the time in real-time applications and subtract the additional time to the original one. This is possible in general software, although the types of instructions that are added for instrumentation often have memory interactions (i.e., loads and stores) and are hard to predict and quantify. However, it is difficult to apply for the real-time hardware, as hardware interactions require precise timing and the embedded application needs to comply with this. In the next section, we introduce iterative instrumentation, a code coverage testing technique without runtime overhead.

3.2 Instrumentation Vs. Iterative Instrumentation

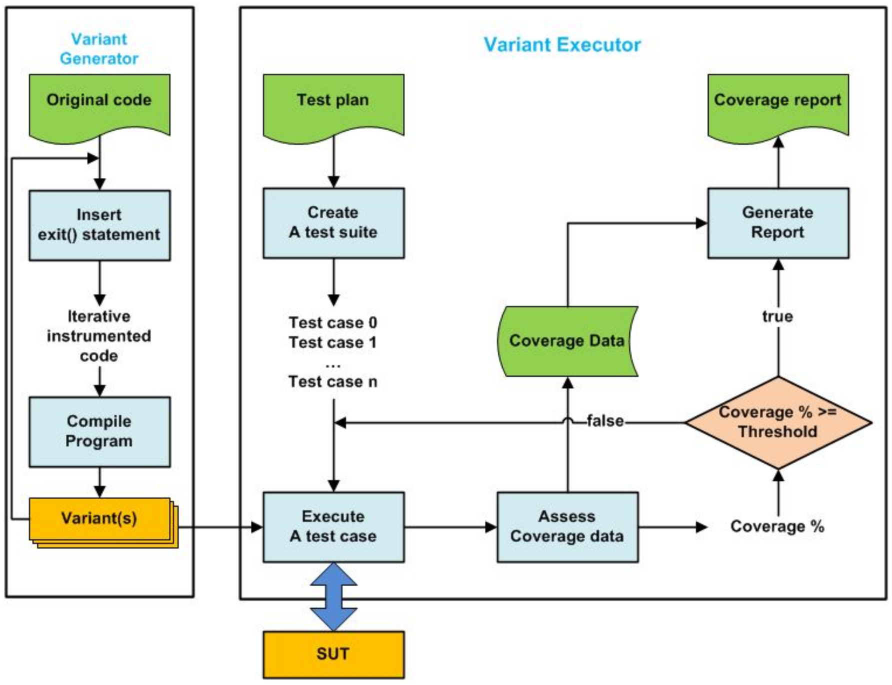
We compare instrumentation and iterative instrumentation techniques for measuring code coverage in time-sensitive software systems. The implementation of these techniques consists of two main components: a generator and an executor.

3.2.1 Instrumentation Code Coverage

Figure 4 (a) shows a diagram of the instrumentation code coverage technique consisting of two main components: an instrumentation generator and an instrumentation executor.



(a)



(b)

Figure 4. Diagrams of (a) the instrumentation code coverage technique and (b) the iterative instrumentation code coverage technique

The instrumentation generator transforms the original code into instrumented code by inserting instructions at certain points as determined by the control flow graph of the original code. Then the instrumented code is compiled into an instrumented executable program.

Next, the instrumentation executor automatically performs the following processes:

1. Initialize a testing configuration for a particular type of coverage criteria: node, edge, or logic coverage, and then generate a test suite according to a given test plan.
2. Execute each test case from a test suite interacting directly with the SUT, store the coverage data into a temporary file and send test results to the assessment process.
3. Assess the test results, evaluate coverage data, and compute a coverage percentage, a proportion of all checked coverage points to the total coverage points times 100. If a threshold, a qualified coverage percentage for this test plan, is reached or all test cases are completed, terminate the testing loop; otherwise, go back to execute the next test case.
4. Summarize the coverage data and generate a report.

3.2.2 Iterative Instrumentation Code Coverage

Figure 4 (b) shows a diagram of the iterative instrumentation Code Coverage technique having two components: a variant generator and a variant executor. Although it is similar to the instrumentation code coverage technique, there are some important differences.

Unlike with instrumentation, the variant generator creates a set of variants rather than a single executable program as each variant represents one instrumented point. To generate each variant, we simply replace an individual instrumented point with an exit statement to terminate the execution immediately with a special value returned to the system.

The second component, the variant executor, also has similar processes to the instrumentation executor except the execution process and the assessment process:

1. The first process (create a test suite) is identical to the process in the instrumentation technique.
2. In the execution process, all surviving variants will be executed with a given test case. This process, however, does not store the coverage data into a temporary file, as an executing program could be killed and terminated any time (the assessment process will determine whether the variant is killed or survives from the returned output).
3. The assessment process determines the returned output whether the variant is killed (an exceptional value) or survives (an expected value). Next, it records the coverage data into the temporary file and computes a coverage percentage (this is similar to a mutant score in [51]). Then the system keeps testing the next variant until the threshold has been met or the last variant is tested with all test cases.

4. The remaining processes (evaluate coverage data and generate a report) are similar to the instrumentation technique except a report is specific for the iterative instrumentation technique.

3.3 Case Study 2: Instrumentation vs. Iterative Instrumentation

In this case study, we compare the effectiveness of the instrumentation and iterative instrumentation techniques for assessing code coverage of test cases for a simple proportional-integral-derivative (PID) controller. In addition to the effectiveness, we compare the execution costs of both techniques.

A PID controller uses three parameters or "gains": the proportional (P), the integral (I), and the derivative (D), for evaluating the difference (an error value) of a measured value and a required value (setpoint). It usually works as a feedback control loop by receiving a current error value and minimizing that error value for the controller by sending a new command [65].

As a case study, we implement a PID controller to manage speeds and directions of a small differential-drive ground robot by attempting to maintain a particular directional setpoint, as measured by a compass. After the PID controller receives a current compass reading and a desired compass reading, it calculates an error between those values. Then it produces an adjusted motor command to control the direction of the vehicle to go back to the setpoint.

The test cases of this case study are randomly selected from six partitions determined by the value of the error:

1. error < -359 or error > 359

2. error = 0
3. error = 180
4. 180 < error < 360
5. -360 < error < -180
6. -180 < error < 0 or 0 < error < 180

```
int get_error(int pos, int sp, int max)
{
  int error = INVALID_VALUE;
  int mid = max/2;
  if (pos >= 0 && pos < max && sp >= 0 && sp < max) {
    if ( pos == sp ) {
      error = 0;
    } else if ( (pos-sp) % mid == 0 ) {
      error = mid;
    } else if (pos - sp > mid) {
      error = ((pos-sp) % mid) - mid);
    } else if (pos - sp < -mid) {
      error = (pos - sp) % mid;
    }
  } else {
    error = (pos-sp/ABS(pos-sp)) * INVALID_VALUE;
  }
  return error;
}
```

Figure 5. The original code of get_error() function

Figure 5 shows the non-instrumented source code of the get_error() function written in C. This function calculates an error value between a current position (pos) and a setpoint (sp). The error value will be validated and adjusted before returning to the caller, the get_pid_speed() function, shown in Figure 6.

```
int get_pid_speed(int pos, int sp, t_pid *pid, int max_degree, int min, int max)
{
        float output = 0;
        int speed = INVALID_VALUE;
        int error = INVALID_VALUE;
        int mid_degree = max_degree/2;

        error = get_error(pos, sp, max_degree);

        if (error >= -mid_degree && error <= mid_degree)
        {
                pid->integral = pid->integral + (error * pid->dt);
                pid->delivative = (error - pid->prev_error)/pid->dt;
                output = (pid->Kp*error)+(pid->Ki*pid->integral)+(pid->Kd*pid->delivative);

                pid->prev_error = error;

                if (output >= -pid->eps && output <= pid->eps) {
                        speed = 0;
                } else {
                        if (output > -min && output < 0) {
                                speed = -min;
                        } else if (output > 0 && output < min) {
                                speed = min;
                        } else if (output < -max) {
                                speed = -max;
                        } else if (output > max) {
                                speed = max
                        } else {
                                speed = (int)lroundf(output);
                        }
                }
        } else {
                speed = (error/ABS(error)) * INVALID_SPEED;
        }
        return speed;
}
```

Figure 6. The original code of the get_pid_speed() function

Figure 6 contains the non-instrumented code of the get_pid_speed() function calling the get_error() function to calculate an error value between a current heading position and a desired setpoint. After that, it computes a PID output and adjusts an appropriate command (a speed) from that output.

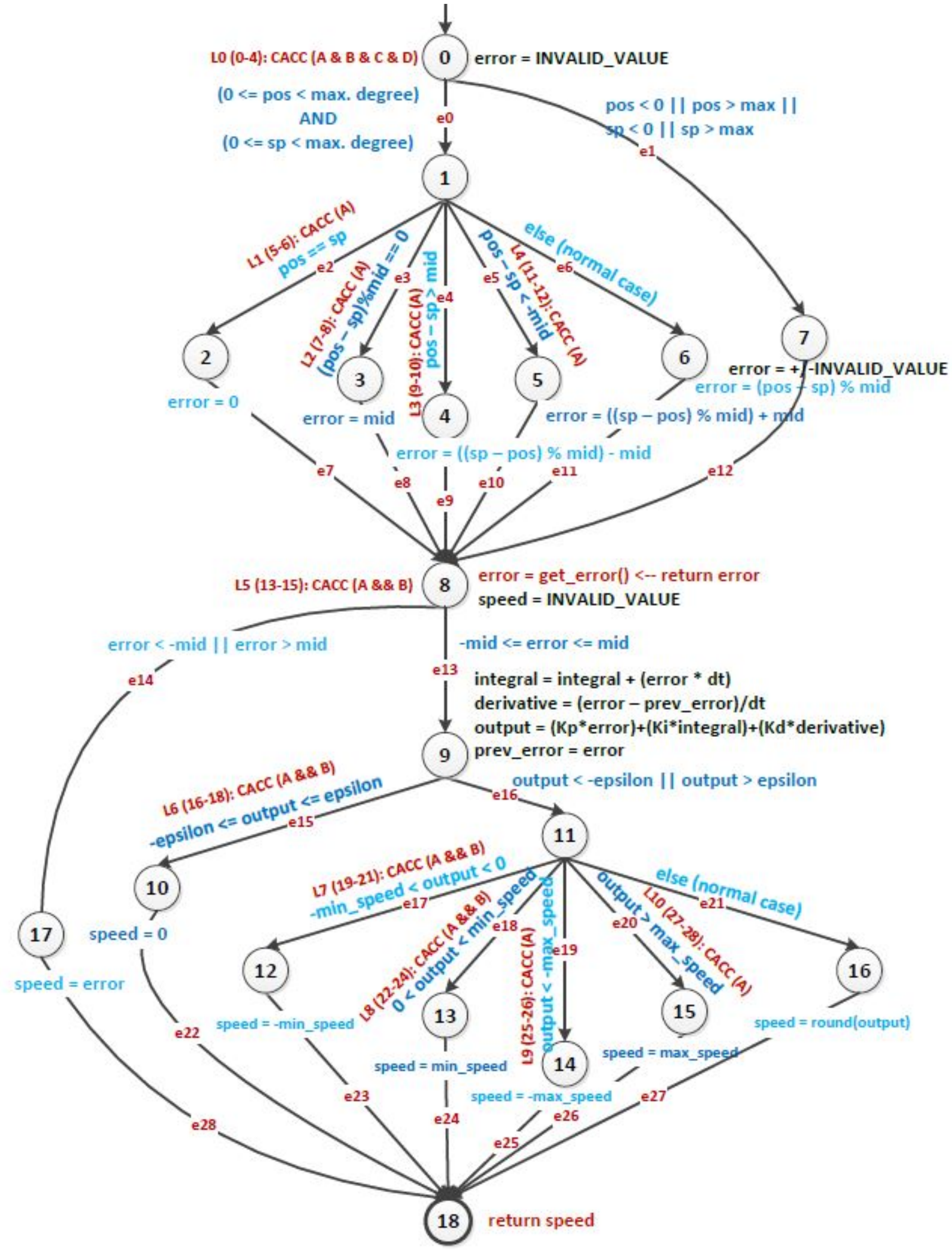


Figure 7. The CFG of the get_error() and get_pid_speed() functions

Figure 7 shows the combined control flow graph (CFG) of the get_error() and get_pid_speed() functions. The CFG contains 19 nodes (n0 - n18), 29 edges (e0 - e28), and 11 logical statements (L0 - L10). This CFG will be used to generate both instrumentation code and iterative instrumentation code for the coverage criteria.

```
if (output >= -pid->eps \
 && output <= pid->eps)  {
  nc[10]++;
  speed = 0;
} else {
  nc[11]++;
  if (output>-min && output<0)
  {
    nc[12]++;
    speed = -min;
  } else if(output>0 && output<min){
    nc[13]++;
    speed = min;
  } else if (output < -max) {
    nc[14]++;
    speed = -max;
  } else if (output > max) {
    nc[15]++;
    speed = max;
  } else {
    nc[16]++;
    speed=(int)lroundf(output);
  }
}
```

```
if (output >= -pid->eps \
 && output <= pid->eps)  {
  //node10: exit(MUT_KILLED);
  speed = 0;
} else {
  //node11: exit(MUT_KILLED);
  if (output>-min && output<0)
  {
    //node12: exit(MUT_KILLED);
    speed = -min;
  } else if (output>0 && output<min) {
    //node13: exit(MUT_KILLED);
    speed = min;
  } else if (output < -max) {
    //node14: exit(MUT_KILLED);
    speed = -max;
  } else if (output > max) {
    //node15: exit(MUT_KILLED);
    speed = max;
  } else {
    //node16: exit(MUT_KILLED);
    speed=(int)lroundf(output);
  }
}
```

Figure 8. Compare (left) the instrumented code and (right) the iterative-instrumented code for node coverage

3.3.1 Node Coverage

From the CFG in Figure 7, each node represents a block of statement (block coverage) except a function call statement.

Figure 8 compares the instrumented code (left) and iterative-instrumented code (right) for node coverage. The instrumentation technique inserts an instrumented point (i.e., nc[i]++ where i is the node identifier) at the beginning of each block, whereas the iterative instrumentation technique places a single exit statement at the same location as the instrumented point.

In the iterative instrumentation technique, all variants are initially marked as a line of comments having no effect to the execution. Then the variant generator removes a comment marker "//node:" and compiles the code to generate a variant. Note that we only show parts of the whole function due to space limitations.

Table 3. Compare node coverage results between (a) the instrumentation testing and (b) the iterative instrumentation testing

tc	sp	pos	exp	act	time	0	1	2	3-15	17	18	coverage
0	78	112	17	17	38	1	1	0	...	0	1	42.11%
1	80	347	-47	-47	32	2	2	0	...	0	2	47.37%
2	460	42	99	99	1	3	2	0	...	1	3	57.89%
...	...	...	...	...	...	...	...	...	...	...	...	...
12	24	536	99	99	1	13	10	1	...	3	13	89.47%
13	38	46	5	5	32	14	11	1	...	3	14	94.74%
14	304	298	-5	-5	26	15	12	1	...	3	15	100.00%

(a)

tc	sp	pos	exp	act	time	0	1	2	3-15	17	18	coverage
0	78	112	17	100	33(722)	1	1	0	...	0	1	42.11%
1	80	347	-47	100	28(384)	x	x	0	...	0	x	47.37%
2	460	42	99	100	1(298)	x	x	0	...	1	x	57.89%
...	...	...	...	...	...	...	...	...	...	...	...	...
12	24	536	99	100	1(2)	x	x	1	...	x	x	89.47%
13	38	46	5	100	19(36)	x	x	x	...	x	x	94.74%
14	304	298	-5	100	22(22)	x	x	x	...	x	x	100.00%

(b)

Table 3 compares the node coverage results from (a) the instrumentation testing and (b) the iterative instrumentation testing. The results are in a tabular format, and the table header contains test case identifiers (tc), setpoints (sp), current positions (pos), expected outputs (exp), actual outputs (act), execution times (time), node identifiers (0, 1, 2, 3, ..., 18), and the coverage percentages (coverage) respectively. Data in each row is a result of each test case. The system keeps executing test cases until a current coverage percentage meets the threshold or the last test case is executed.

There are some differences between these reports. First, the actual values of the iterative version are special values (100) to indicate variants killed, whereas the instrumentation shows normal returned values. Second, in the time column the iterative table shows the average time used for each variant and the total time used for all executed variants (in the parentheses), while the instrumentation data is the actual time. As there are multiple variants executed for each test case, we show the individual time and total time in the same row for the iterative report. Lastly, the iterative coverage data uses a special character 'x' to represent a killed variant, while the instrumentation displays the values of array elements (0, 1, 2, ..., n; n = max. test case identifiers).

As both techniques achieve 100% coverage, it is clear that our technique is just as effective as instrumentation for measuring node coverage. However, instrumentation has less total execution time than the iterative technique as it must execute each test case on multiple variants. On the other hand, the iterative technique has less execution time than instrumentation for each individual execution since it can terminate the program whenever a variant is detected, while the instrumentation executes and exits the program normally. As test case execution is easily parallelizable, we do not see the total (sequential) time for the iterative technique to be a major barrier to use.

```
if (output >= -pid->eps \
 && output <= pid->eps)  {
  ec[15]++;
  speed = 0;
  ec[22]++;
} else {
  ec[16]++;
  if (output > -min && output < 0) {
    ec[17]++;
    speed = -min;
    ec[23]++;
  } else if (output > 0 && output < min) {
    ec[18]++;
    speed = min_speed;
    ec[24]++;
  } else if (output < -max_speed) {
    ec[19]++;
    speed = -max_speed;
    ec[25]++;
  } else if (output > max_speed) {
    ec[20]++;
    speed = max_speed;
    ec[26]++;
  } else {
    ec[21]++;
    speed = (int)lroundf(output);
    ec[27]++;
  }
}
```

```
if (output >= -pid->eps \
 && output <= pid->eps)  {
  //edge15: exit(MUT_KILLED);
  speed = 0;
  //edge22: exit(MUT_KILLED);
} else {
  //edge16: exit(MUT_KILLED);
  if (output > -min && output < 0) {
    //edge17: exit(MUT_KILLED);
    speed = -min_speed;
    //edge23: exit(MUT_KILLED);
  } else if (output > 0 && output < min) {
    //edge18: exit(MUT_KILLED);
    speed = min_speed;
    //edge24: exit(MUT_KILLED);
  } else if (output < -max_speed) {
    //edge19: exit(MUT_KILLED);
    speed = -max_speed;
    //edge25: exit(MUT_KILLED);
  } else if (output > max_speed) {
    //edge20: exit(MUT_KILLED);
    speed = max_speed;
    //edge26: exit(MUT_KILLED);
  } else {
    //edge21: exit(MUT_KILLED);
    speed = (int)lroundf(output);
    //edge27: exit(MUT_KILLED);
  }
}
```

Figure 9. Compare (left) the instrumented code and (right) the iterative-instrumented code for edge coverage

3.3.2 Edge Coverage

The instrumented code for edge coverage is similar to node coverage but having more instrumented points, since each edge consists of two nodes. Therefore, the instrumented code is roughly twice the size of the node (block) coverage. While the node coverage inserts one point at the beginning or at the end of each block, edge coverage inserts two points (at the beginning and at the end of each block).

Figure 9 compares the instrumented code (left) and the iterative-instrumented code (right) for edge coverage. The instrumented code has a pair of instrumented points at the beginning and at the end of each block. Similarly, the iterative version has the same number of variants and locations as the instrumented points. Note that each variant is marked with a line of comments similar to the node coverage and will be active only one line at a time while generating a variant.

Table 4. Compare edge coverage results between (a) the instrumentation testing and (b) the iterative instrumentation testing

tc	sp	pos	exp	act	time	0	1	2	3-25	27	28	coverage
0	78	112	17	17	38	1	0	0	...	1	0	24.14%
1	80	347	-47	-47	32	2	0	0	...	2	0	31.03%
2	460	42	99	99	1	3	1	0	...	3	1	44.83%
...	...	...	...	...	...	...	...	...	...	...	...	...
12	24	536	99	99	1	13	11	8	...	13	11	86.21%
13	38	46	5	5	32	14	12	9	...	14	12	93.10%
14	304	298	-5	-5	26	15	13	10	...	15	13	100.00%

(a)

tc	sp	pos	exp	act	time	0	1	2	3-25	27	28	coverage
0	78	112	17	100	38(725)	1	0	0	...	1	0	24.14%
1	80	347	-47	100	32(381)	x	0	0	...	x	0	31.03%
2	460	42	99	100	1(325)	x	1	0	...	x	1	44.83%
...	...	...	...	...	...	...	...	...	...	...	...	...
12	24	536	99	100	1(2)	x	x	x	...	x	x	86.21%
13	38	46	5	100	28(59)	x	x	x	...	x	x	93.10%
14	304	298	-5	100	22(22)	x	x	x	...	x	x	100.00%

(b)

Table 4 compares the results of edge coverage between (a) the instrumentation testing and (b) the iterative instrumentation testing. The formats of the results are the same as the node coverage shown in Table 3, and the contents are similar except the number of edges is larger and the coverage percentages are slightly different. As the comparison shows, the effectiveness of both techniques is identical. Similar to node coverage, the iterative instrumentation testing has a longer total execution (because of the execution of multiple mutants) but less individual execution time (due to earlier termination) than the instrumentation.

3.3.3 Logic Coverage

Logic coverage [24] targets coverage of clauses in boolean expressions used in conditional and iteration statements, and is more complicated than node and edge coverage, since it may consist of multiple clauses resulting in several instrumented points to satisfy its test requirements. Instead of inserting a single statement at an instrumented point it places an instrumented probe (a function call) for each logic coverage and the related instrumented points are located inside the called function.

As there are many types of logic coverage [52]: Clause Coverage (CC), General Active Clause Coverage (GACC), Correlated Active Clause Coverage (CACC), and Restricted Active Clause Coverage (RACC), we select CACC, which is similar to MCDC required by the US FAA for safety-critical applications, to present here. CACC requires a major clause to evaluate to both true and false, so a test suite should cover both boolean values of each major clause. For example, in Figure 7 the CACC coverage L6 (at node 10) represents a predicate $A \wedge B$ where clause A represents (output >= -epsilon) and clause B represents (output <= epsilon). In this case, the minor clause must be true, so that the major clause can determine the predicate $A \wedge B$. This predicate will be passed to the CACC_AaB function for measuring the CACC coverage of $A \wedge B$. To satisfy this

CACC coverage, at least three test cases: (A=true, B=false), (A=false, B=true), and (A=true, B=true) are required.

```
int CACC_A(int i, int A)
{
  if (A)
    lc[i]++;
  else
    lc[i+1]++;
  lci = i+2; return lci;
}
```

```
int CACC_A(int i, int A)
{
  if (i == 0 && A)
    exit(MUT_KILLED);
  else if (i == 1 && !A)
    exit(MUT_KILLED);
  return 0;
}
```

(a)

```
int CACC_AaB(int i, int A, int B)
{
  if (A && B) lc[i]++;
  else if (!A && B)
    lc[i+1]++;
  else if (A && !B)
    lc[i+2]++;
  lci = i+3;
  return lci;
}
```

```
int CACC_AaB(int i, int A, int B)
{
  if (i == 0 && A && B) exit(MUT_KILLED);
  else if (i == 1 && !A && B)
    exit(MUT_KILLED);
  else if (i == 2 && A && !B)
    exit(MUT_KILLED);
  return 0;
}
```

(b)

```
int CACC_AaBaCaD(int i,int A,int B,int C,int D)
{
  if (A && B && C && D)
    lc[i]++;
  else if (!A && B && C && D)
    lc[i+1]++;
  else if (A && !B && C && D)
    lc[i+2]++;
  else if (A && B && !C && D)
    lc[i+3]++;
  else if (A && B && C && !D)
    lc[i+4]++;
  lci = i+5;
  return lci;
}
```

```
int CACC_AaBaCaD(int i,int A,int B,int C,int D)
{
  if (i == 0 && A && B && C && D)
    exit(MUT_KILLED);
  else if (i == 1 && !A && B && C && D)
    exit(MUT_KILLED);
  else if (i == 2 && A && !B && C && D)
    exit(MUT_KILLED);
  else if (i == 3 && A && B && !C && D)
    exit(MUT_KILLED);
  else if (i == 4 && A && B && C && !D)
    exit(MUT_KILLED);
  return 0;
}
```

(c)

Figure 10. Compare CACC predicates: (a) A, (b) $A \wedge B$, and (c) $A \wedge B \wedge C \wedge D$, for (left) the instrumentation and (right) iterative instrumentation techniques

```
CACC_AaB(16,output >= -pid->eps, \
output <= pid->eps);

if (output >= -pid->eps && output < =pid-
>eps)  {
  speed = 0;
} else {
  CACC_AaB(19, output > -min_speed,…)


  if (output > -min_speed && output < 0)
{
    speed = -min_speed;
  } else {
    CACC_AaB(22, output > 0,…)


    if (output > 0 && output<min_speed) {
      speed = min_speed;
    } else {
      CACC_AaB(25, output < -max_speed);

      if (output < -max_speed) {
        speed = -max_speed;
      } else {
        CACC_AaB(27, output > max_speed);

        if (output > max_speed) {
          speed = max_speed;
        } else {
          speed = (int)lroundf(output);
        }
      }
    }
  }
}
```

```
//logic16:CACC_AaB(0,output >= -pid->eps,
//logic17:CACC_AaB(1,output >= -pid->eps,
//logic18:CACC_AaB(2,output >= -pid->eps,
if (output >= -pid->eps && output <= pid-
>eps)  {
  speed = 0;
} else {
  //logic19:CACC_AaB(0,output> -min_speed,
  //logic20:CACC_AaB(1,output> -min_speed,
  //logic21:CACC_AaB(2,output> -min_speed,
  if (output > -min_speed && output<0) {
    speed = -min_speed;
  } else {
    //logic22:CACC_AaB(0, output > 0,
    //logic23:CACC_AaB(1, output > 0,
    //logic24:CACC_AaB(2, output > 0,
    if (output > 0 && output < min_speed)
{
      speed = min_speed;
    } else {
  //logic25:CACC_AaB(0,output<-max_speed);
  //logic26:CACC_AaB(1,output<-max_speed);
      if (output < -max_speed) {
        speed = -max_speed;
      } else {
  //logic27:CACC_AaB(0,output>max_speed);
  //logic28:CACC_AaB(1,output<-max_speed);
        if (output > max_speed) {
          speed = max_speed;
        } else {
          speed = (int)lroundf(output);
        }
      }
    }
  }
}
```

Figure 11. Compare (left) the instrumented code and (right) the iterative-instrumented code for logic coverage

Figure 10 compares CACC predicates: (a) A, (b) A ∧ B, and (c) A ∧ B ∧ C ∧ D between the instrumentation (left) and iterative instrumentation techniques (right). The instrumented version updates a related instrumented point if its condition is true, while the iterative one terminates the program if a mutant identifier is matched and a corresponding condition is true (i.e., variable i is used as a test case identifier instead of an index). The instrumentation returns an updated index, whereas the iterative version simply returns zero (unsatisfied). In addition to these predicates, we can implement as many CACC predicates as needed depending on particular programs.

Figure 11 compares the instrumented code (left) and the iterative-instrumented code (right) for logic coverage. Since each CACC function contains many instrumented points, the number of instrumented points is about 3 times the number of CACC function calls in the program. Thus the cost of logic coverage is much more than the costs of node and edge coverage.

Table 5. Compare logic coverage results between (a) the instrumentation testing and (b) the iterative instrumentation testing

tc	**sp**	**pos**	**exp**	**act**	**time**	**0**	**1**	**2**	**3-25**	**27**	**28**	**coverage**
0	78	112	17	17	46	1	0	0	...	0	1	37.93%
1	80	347	-47	-47	42	2	0	0	...	0	2	51.72%
2	460	42	99	99	2	2	0	0	...	0	2	58.62%
...	...	...	...	...	...	...	...	...	...	...	...	...
14	304	298	-5	-5	32	12	0	1	...	2	6	93.10%
15	314	314	0	0	22	13	0	1	...	2	6	93.10%
16	49	49	-99	-99	1	13	1	1	...	2	6	100.00%

(a)

tc	**sp**	**pos**	**exp**	**act**	**time**	**0**	**1**	**2**	**3-25**	**27**	**28**	**coverage**
0	78	112	17	100	42(940)	1	0	0	...	0	1	37.93%
1	80	347	-47	100	38(573)	x	0	0	...	0	x	51.72%
2	460	42	99	100	1(354)	x	0	0	...	0	x	58.62%
...	...	...	...	...	...	...	...	...	...	...	...	...
14	304	298	-5	-5	29(86)	x	0	x	...	x	x	93.10%
15	314	314	0	0	18(42)	x	0	x	...	x	x	93.10%
16	49	49	-99	100	1(1)	x	1	X	...	x	x	100.00%

(b)

Table 5 compares the logic coverage results between (a) the instrumentation testing and (b) iterative instrumentation testing. Similar to the node and edge coverage, the iterative instrumentation technique is just as effective as the instrumentation technique for measuring logic coverage. The instrumentation is less expensive than the iterative one for the overall testing time, but it is inefficient for each individual test case. More precisely, the execution time of the iterative technique is less than or equal to the instrumentation, but some test cases are not different (e.g. tc#16 only executes for 1 µs) as they terminate the program at the beginning.

The results in this case study indicate that iterative instrumentation is as effective as instrumentation for the types of coverage criteria we considered. Although the total cost of the mutation is more than the instrumentation, we are rather interested in the individual cost of each execution that the instrumentation takes more time than the mutation in most test cases. This implies that the iterative instrumentation is an alternate code coverage testing technique suited for time-sensitive systems.

3.4 Threats to Validity

In this section, we explain both internal and external threats to validity, and we also discuss how these threats could be alleviated.

3.4.1 Internal Threats to Validity

All variables (i.e., function input) in the case studies are controlled by the test case generator that generates the test cases from the valid values in six partitions. We also increase the confidence of the cause-effect relationships by using different types of code coverage criteria (i.e., node, edge, and logic coverage) to observe the runtime overhead (i.e., additional execution time) caused by these instrumented code coverage. Since our primary objective is to indicate the effect of the instrumentation runtime overhead for time-sensitive systems, it is unnecessary to use more complicated coverage criteria to demonstrate that issue. In our future work, however, we may include more complex coverage criteria such as data flow and path coverage that are essential for testing safety-critical systems.

3.4.2 External Threats to Validity

Although iterative instrumentation adapts the concept of weak mutation, it only uses a single exit operator and it does not check the states changed during the test. Therefore, it cannot simulate various faults like the traditional mutation analysis using several different types of operators. To create a fully-functional mutation testing tool (that is also able to measure code coverage), we would have to use a set of relevant mutation operators rather than using only the exit operator. However, this leads to more cost and several mutant reduction techniques would need to be applied.

In addition to the operators, the number of test cases in the Case Study 2 might be difficult to affirm that the obtained results and conclusions are conclusive. However, to mitigate these threats, we can use real-world applications with a larger number of test cases in the evaluation. In the next chapter, therefore, we use another PID controller from the widely used embedded project called "cleanflight" along with a set of large test cases in the remaining case studies.

3.5 Discussion

We analyze the trade-off of iterative instrumentation compared to traditional instrumentation. We also discuss possible approaches to improve the quality of our technique.

3.5.1 Trade-off Analysis

The iterative instrumentation technique (i.e., iterative) has many advantages compared to the traditional instrumentation technique (i.e., instrumentation). First, the iterative one does not interfere the timing of time-sensitive systems. It only inserts a single statement at the instrumented point aborting the program immediately, whereas the instrumentation inserts a number of instructions. Second, the iterative technique can precisely measure the execution time at any point of the code. It can measure the same timing as the original code (i.e., from the beginning to the measured point), while the instrumentation adds additional time to the SUT resulting in different timing from the original code. Lastly, in memory-constrained systems (i.e., non-OS embedded platforms), the iterative technique has no impact on memory usage, since it only adds one exit statement into the original code. In contrast, the available memory of such systems may not be sufficient for a large number of additional instructions inserted by the instrumentation technique, along with the in-memory data structures used to track coverage. In our case studies, the executable file sizes of the testing code are increased by 20-30% for node and edge coverage and 75% for logic coverage.

3.5.2 Possible Improvements

As discussed above, our code coverage technique could have the scalability and performance issues from executing multiple variants especially when the number of variants is larger. Based on our studies, there are two major approaches for solving these problems: reducing the number of mutants and parallel execution.

In large programs, the number of variants would increase much more than our example resulting in a scalability issue. To improve this we need to reduce the number of generated variants. As we only use a single exit operator, the mutant reduction techniques like selective mutation [49-50] and mutant sampling [54-55] are irrelevant. Selective mutation only selects most effective and unique operators, while mutant sampling randomly selects a subset of the operators. However, the idea of higher order mutation or second order mutation [56-57] may be possible to reduce the number of variants in our case as one instrumented code (i.e., variant) can have two or more exit statements. If a variant consists of two exit statements, the number of variants can be reduced by half.

In addition to reducing the number of variants, the multi-variant executions are also the other issue to address. In particular for large programs, the total execution time can be expensive. Mateo and Usaola study the size and complexity of the SUT affecting both generation and execution costs. In their experiments, the generating cost grows linearly, while the execution cost increases exponentially [40]. Krauser et al. [58] use SIMD machines to execute multiple mutants concurrently, while Offutt et al. [59] distribute mutants to execute on MIMD machines. To improve the performance, we can apply parallel computing techniques for multi-variants executions, since the process is inherently parallelizable.

According to our discussions, we propose cluster iterative instrumentation, a graph clustering technique that can reduce the number of nodes in a control flow graph, which is more efficient than iterative instrumentation in the next chapter. We also apply parallel computing to our unit testing framework using a cluster of embedded platforms described in Chapter 5.

CHAPTER 4: CLUSTER ITERATIVE INSTRUMENTATION

As demonstrated in the previous chapter, while iterative instrumentation can measure code coverage without runtime overhead, its total execution time is expensive due to multiple executions of variants (each test case is executed by all variants of the system under test). To improve the efficiency of iterative instrumentation, we propose cluster iterative instrumentation that can reduce the number of nodes in a control flow graph (CFG) resulting in a fewer number of variants to execute. In this chapter, we describe cluster rules and algorithms. Then, we evaluate the cluster iterative instrumentation approach with Case Study 3 by comparing the execution times of node coverage between the cluster iterative instrumentation and iterative instrumentation techniques. Lastly, we discuss trade-off analysis and possible improvements of cluster iterative instrumentation.

4.1 Cluster Rules and Algorithm

In this section, we explain the concept of cluster iterative instrumentation with the graph clustering rules. Then we provide the algorithm for transforming a CFG into a cluster graph.

4.1.1 The Graph Clustering Rules

Graph clustering is a grouping method that groups elements in the same group that are related to each other or have a similar property [61]. Our graph clustering rules are used to transform the original CFG into a cluster graph for evaluation with the cluster iterative instrumentation technique.

The idea of this technique is to reduce the number of nodes in the CFG by grouping nodes having the same property into one node. As the number of nodes is reduced, the number of executions (one for each variant) is also decreased. With this concept, we define our graph clustering rules as follows:

- Given a direct graph G = (C, B, s) where C is a set of common nodes, B is a set of branched nodes, and s is a start node in the graph.
- C is a set of nodes that are grouped in the same "common node" if and only if all test paths starting from node s to the last node always pass through these nodes.
- B is a set of nodes that are grouped in the same "branched node" if and only if these nodes are not common nodes and they are children from the same parent node (i.e., they are siblings).

The concept of the common node in our clustering rules is adapted from a *dominator tree* originally proposed by Prosser [71]. A "dominator" of node n is a node that is always reachable by any test path from the root to node n, and we can define a notation of this dominator as DOM(n). A set of dominators from the root to node n is called a "dominance frontier," and one of the dominators that is also a parent of node n is called an "immediate dominator" or IDOM(n). The first dominance algorithm is presented by Lowry and Medlock [72] for optimizing structures in FORTRAN compiler. However, there is the important condition that all branched nodes require to comply with our rules. All nodes grouped into a branched node cannot be executed in the same test path, as the test case will reach the exit statement inserted in the previous nodes of the path and exit the program before reaching the remained nodes in the path.

There are both similarity and differences between a dominator tree and our common nodes. The similarity between these concepts is that all dominators are reached by a test path from the root to the last node of the path. This allows the last node to refer to all dominators along the test path, so that these nodes are grouped into the same common node in our clustering algorithm. Despite having the similarity, there are many differences between the dominator tree and our concepts. First, in the common nodes, all test paths from the root to the last node must pass all dominator nodes. However, the dominators of the dominator tree are unnecessary to be reachable by all test paths. In our rules, therefore, nodes that are not always reachable by all test paths will be separated into the other group called branched nodes. Second, the dominator algorithm transforms a CFG into a dominator tree, but our clustering algorithm transforms a CFG into a list of clustered nodes (i.e., common nodes and branched nodes).

In addition to the clustering rules defined above, our implementation addresses two important points in transforming a CFG into a list of clustered nodes. First, although all nodes in a common node are always reached by the same test paths, we add the common node as the last node in the cluster graph, because this implies that the test case reaching the last node already executed with all previous common nodes along the test path. Next, to deal with loops in a CFG, our algorithm tracks a common node by the number of node occurrences counted; within a loop, this must be greater than the number of tests. For example, a test path [1, <u>2, 3, 4, 2</u>, 6] means a path from node 2 to node 4 is a loop, so the number of node 2 will be greater than the number of tests. If node 1 is a start node and node 6 is a stop node, the execution count of node 1 and the execution count of node 6 will be equal to the number of tests. With our logic described above, all nodes that have the number of occurrences greater or equal to the number of tests will be the common nodes, so the nodes 1, 2 and 6 are the common nodes in this example.

As an example, we use a simple CFG to show how our clustering algorithm transforms it into a cluster graph with our rules as follows.

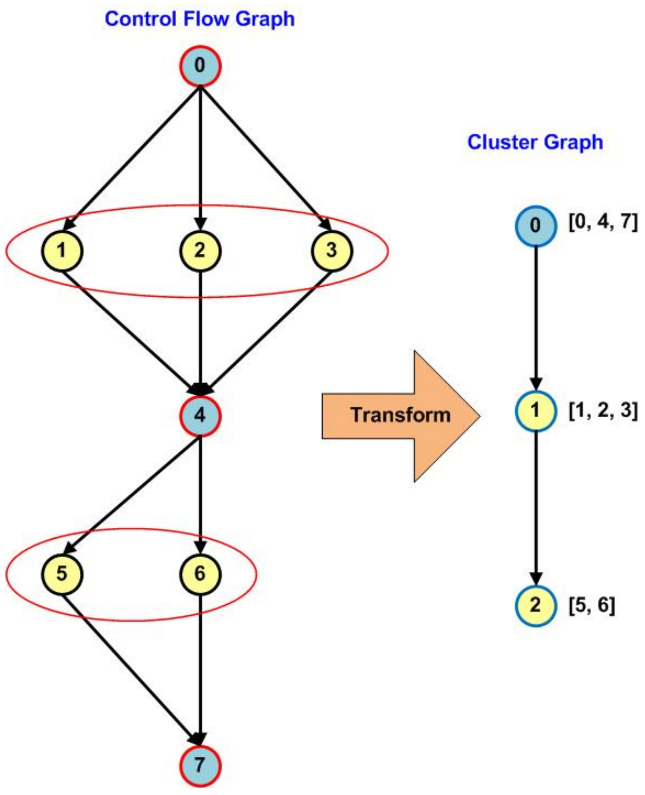


Figure 12. The cluster graph (right) transformed from a CFG (left)

Figure 12 shows the cluster graph (right) transformed from a CFG (left) by grouping nodes into three clustered nodes using the above rules. In the right graph, the clustered node 0 (cn0) is a common node consisting of node 0, 4, and 7 from the CFG, as these nodes are always executed by any test case. The clustered node 1 and 2 are branched nodes because they are branched from the same parents. Node 1, 2, and 3 are branched from the parent node 0, and node 5 and 6 are branched from the parent node 4 respectively. Next, we describe the cluster iterative instrumentation algorithm that transforms the CFG into the cluster graph and evaluate code coverage using the cluster iterative instrumentation technique.

4.1.2 The Cluster Iterative Instrumentation Algorithm

To transform a CFG into a cluster graph, we use a set of possible test paths as input test cases. From the CFG in Figure 12, there are six possible test paths as shown in Table 6.

Table 6. All possible test paths from the CFG in Figure 12

No	**Test paths**
0	{0, 1, 4, 5, 7}
1	{0, 2, 4, 5, 7}
2	{0, 3, 4, 5, 7}
3	{0, 1, 4, 6, 7}
4	{0, 2, 4, 6, 7}
5	{0, 3, 4, 6, 7}

From Table 6, we can analyze the execution count of each node across all test paths and determine whether a node is a common or branched node. A common node has the execution count greater or equal to the number of test paths, while a branched node has an execution count more than one but less than the number of test paths. In addition to the execution count across all test paths, we also need to know a parent of each node especially the branched node. Thus, our algorithm must count the number of nodes in all test paths and link them to their parents as well.

```
// init code coverage, parents, test paths, and test cases
initialize a list of coverage points to 0 (cov[] = {0})
initialize a list of parents to -1 (parents[] = {-1})

// count node frequencies and link to their parents
for p in range (0, len(tp))
  parent = -1
  for n in range (0, len(tp[p]))
    node = tp[p][n]
    cov[node] += 1
    parents[node] = parent
    parent = node

// add nodes into common and branch lists
for node_no in range (0, NUM_NODES)
  if (cov[node_no] >= len(tp))
    append node_no to a common list
  else
    parent = parents[node_no]
    append node_no to a branch list

// merge common and branch lists into a cluster_node_list
prev_parent = -1
for i in range(0,len(branch_list))
    parent = parents[branch_list[i]]
    if (parent = prev_parent)
        append current branch list to a temp list
    else:
        append a temp list to a cluster node list
        clear a temp list
        append current branch list to a temp list
    prev_parent = parent
if (a temp list is not empty)
    append a temp list to a cluster node list
```

Figure 13. The pseudo-code of cluster iterative instrumentation algorithm

In addition to the mechanism to group nodes, the exit value from a variant for cluster iterative instrumentation is also different from the one in iterative instrumentation. In the iterative instrumentation technique, the returned value from the executed variant is always the same value (a special value) indicating the variant is killed [35]. However, this cannot identify which node is reached in our cluster node, as each variant in the cluster iterative instrumentation technique can be two or more. Thus, we return the node identifier reached instead of a special value from the executed variant. In this version, if there is no node reached, the variant returns a negative value instead. The node identifiers range from 0 to n where n+1 is the number of nodes in the CFG, and the negative value (i.e., -1 in our prototype) indicates that the variant is unkilled.

Using this algorithm with the given example test paths in Table 6, the cluster list generated by the algorithm contains three clustered nodes (cn0 - cn3) as follows:

cn_list[0] = {0, 4, 7}

cn_list[1] = {1, 2, 3}

cn_list[2] = {5, 6}

From this cluster list, each clustered node contains a set of nodes in a CFG: clustered node 0 links to three nodes (0, 4, and 7); clustered node 1 links to three nodes (1, 2, and 4), and clustered node 2 links to two nodes (5 and 6). This implies that each variant (i.e. clustered node) in the cluster iterative instrumentation technique can have two or more exit statements. However, with our graph clustering rules, only one of them can be reached by each test execution.

Although cluster iterative instrumentation can reduce the number of nodes from a CFG, the evaluation process of this technique is more complicated than iterative instrumentation due to multiple exit statements in each variant. If an exit statement is reached, the variant returns a node identifier. However, if none of the exit statements reached, the variant returns a negative value instead. For the above example, if any exit statement in the branched node is reached, the variant returns a different value (i.e., node identifier) depending on which exit statement is detected. Since each branched node can have more than two exit statements, the variant will be killed if all nodes (exit statements) are reached. In this case, this branched node needs at least three tests to kill the variant.

Unlike the branched nodes, all exit statements in the common node are always reached by any test. In this case, we only insert a single exit statement in the last node of this set (i.e., the last node in the test path), so the returned value from the common node is the identifier of the last node (i.e., 7 in this example). Placing the exit statement at the last node assures that there is no additional runtime overhead before this location. Thus, a common node can be killed with a single test. After executing the current test with all unkilled variants, the coverage percentage is updated by dividing the number of killed nodes (normal nodes in the CFG) with the number of all nodes and multiplying by 100. The framework keeps testing until the threshold is reached or all tests are executed. The implementation of the graph clustering algorithm written in Python, example input (i.e., a set of test paths for a CFG with a loop), and the output of the example (i.e., a cluster graph) are described in Appendix A.

To evaluate the efficiency of cluster iterative instrumentation, we compare the code coverage results between this technique and the iterative instrumentation technique in the next section.

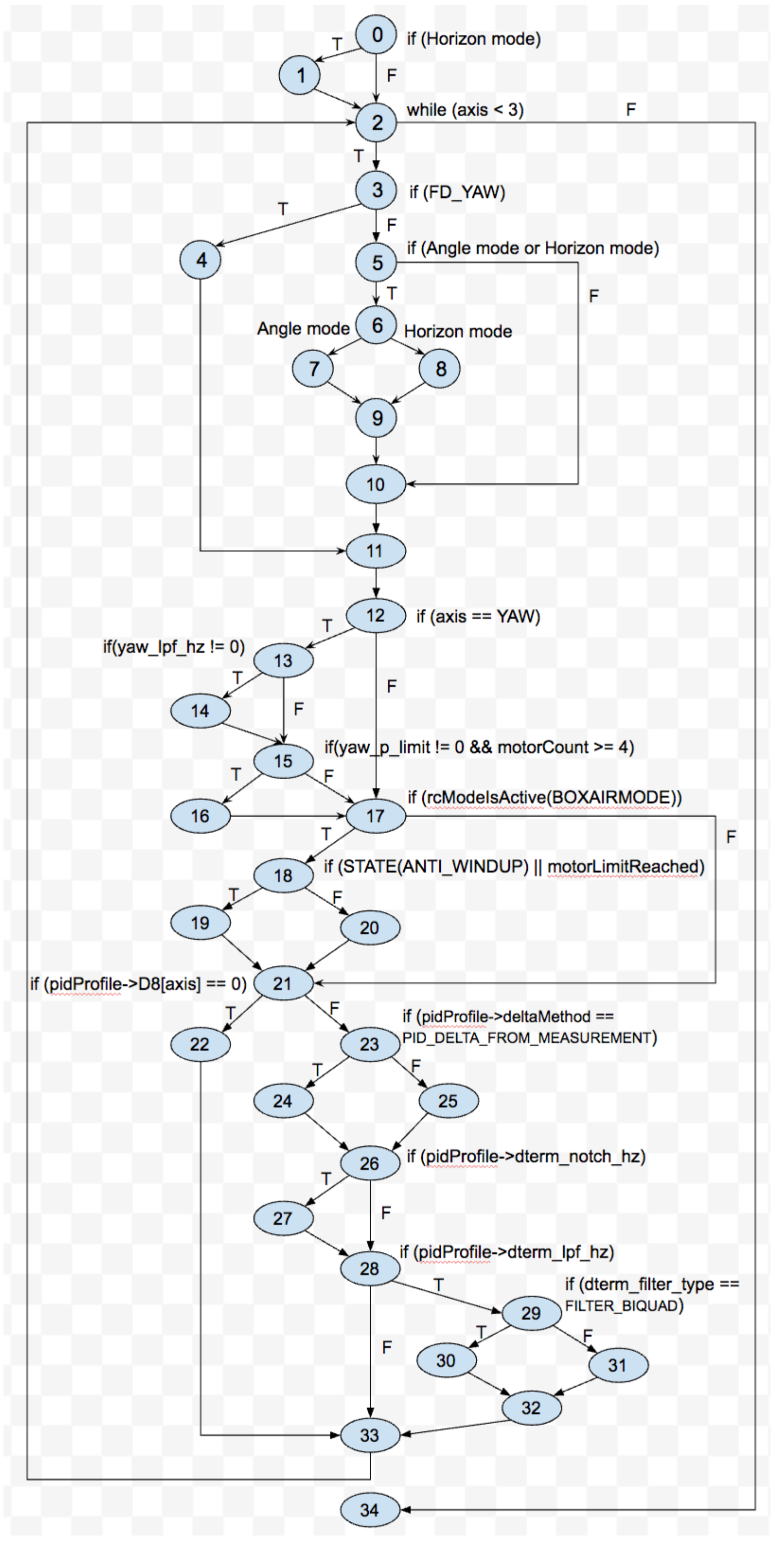

if (Horizon mode)
while (axis < 3)
if (FD_YAW)
if (Angle mode or Horizon mode)
Angle mode
Horizon mode
if (axis == YAW)
if(yaw_lpf_hz != 0)
if(yaw_p_limit != 0 && motorCount >= 4)
if (rcModeIsActive(BOXAIRMODE))
if (STATE(ANTI_WINDUP) || motorLimitReached)
if (pidProfile->D8[axis] == 0)
if (pidProfile->deltaMethod == PID_DELTA_FROM_MEASUREMENT)
if (pidProfile->dterm_notch_hz)
if (pidProfile->dterm_lpf_hz)
if (dterm_filter_type == FILTER_BIQUAD)

Figure 14. A CFG of the pidLuxFloat() and pidLuxFloatCore() functions

4.2 Case Study 3: Cluster Iterative Instrumentation Vs. Iterative Instrumentation

This section compares the efficiency of cluster iterative instrumentation and iterative instrumentation by evaluating code coverage on a real-world flight controller named "Cleanflight."

4.2.1 A PID Controller of Cleanflight Project

Cleanflight is a third-party open source project created by Clifton [60]. This project is widely used on multi-rotor and fixed-wing aircraft, and it consists of multiple software modules: drivers, I/O, flight, scheduler, and sensors. Among these modules, a PID controller is an important part that continuously adjusts three parameters or "gains" while flying. These parameters (i.e., proportional, integral, and derivative) are used for determining the output command based on the difference between a measured value and a desired value [40]. It works as a feedback control loop by receiving a current error value and minimizing that error value for the controller by sending a new command to the controlled device. The version used in our case study is the latest release (v1.14.2), and the pidLuxFloat() and pidLuxFloatCore() are the main functions for a PID controller used in our evaluation.

In Figure 14, there are 35 nodes in the CFG, which is a combination of the pidLuxFloat() and pidLuxFoatCore() functions. The pidLuxFloat() function is called directly by our test cases (nodes 0 –11, and 34), and the pidLuxFloatCore() function is called by the pidLuxFloat() function (nodes 12 – 33). The pidLuxFloat() function is concerned with three axes (i.e., roll, pitch, and yaw), and it iteratively computes the PID terms for each axis by calling the pidLuxFloatCore() function. Following our cluster iterative instrumentation algorithm, we can generate a list of cluster nodes as shown in Table 7.

Table 7. The clustered nodes transformed from the CFG of the pidLuxFloat() and pidLuxFloatCore() functions

Clustered nodes	**Normal nodes**
0	{0, 2, 34}
1	{3, 11, 12, 17, 21, 33}
2	{5, 10}
3	{6, 9}
4	{13, 15}
5	{23, 26, 28}
6	{29, 32}
7	{1}
...	...
15	{24, 25}
16	{27}
17	{30, 31}

Table 7 shows a list of cluster nodes transformed from the CFG of the pidLuxFloat() and pidLuxFloatCore() functions. The first 7 nodes (nodes 0 - 6) are common nodes, while the other nodes (nodes 7 - 17) are branched nodes. We can see that most branched nodes consist of two nodes, but some only have a single node in a cluster as their siblings are already grouped in the common nodes. For example, node 2 is grouped in the same common node as node 0 and node 34, so node 1 is the only one assigned in the cluster node 7. According to the cluster list, the number of nodes is reduced by 48.57%.

After generating the cluster list, we evaluate node coverage for the pidLuxFloat() and pidLuxFloatCore() functions using three code coverage techniques: traditional instrumentation, iterative instrumentation and cluster iterative instrumentation in the next section.

4.2.2 Evaluation Setup

We compare the execution times for evaluating node coverage with three code coverage techniques: traditional instrumentation (i.e., instrumentation), cluster iterative instrumentation, and iterative instrumentation. Although we address the efficiency between the iterative instrumentation and cluster iterative instrumentation techniques, we also evaluate the traditional instrumentation with the same test cases to show the increased times for both approaches compared to the base technique.

In this evaluation, we generate three types of executable code: instrumentation code, iterative instrumentation code, and cluster iterative instrumentation code. The instrumentation code only has a single executable file as this file contains all instrumented nodes. In contrast, the iterative instrumentation and cluster iterative instrumentation consist of a series of executable files. Each executable file represents a unique node in the CFG (35 nodes for the iterative instrumentation) and the cluster graph (18 nodes for the cluster iterative instrumentation) respectively. These instrumented files are executed on the same embedded platform (i.e., Beaglebone Black) with different code coverage techniques selected through the user interface of our unit testing framework.

To generate the test cases for the pidLuxFloat() function, we set values for three global parameters (i.e., flight modes, flight states, and box types), and pass PID profiles (e.g. P, I, and D terms, yaw, tilt, and notch) as parameters to the testing function. In this evaluation, we create a range of test cases by selecting the number of possible values for the configurations and PID profiles as follows:

1. Test suite 1 consists of 128 tests with the following values:
    - Two flight modes, two flight states, two box types, two D terms, two yaw values, two delta methods, and two filter types
    - The PID profiles only use the default values
2. Test suite 2 consists of 288 tests with the following values:
    - Three flight modes and three flight states
    - The remaining values are the same as the test suite 1
3. Test suite 3 consists of 720 tests with the following values:
    - Four flight modes, four flight states, and three box types
    - The rests are same as the test suite 2
4. Test suite 4 consists of 1600 tests with the following values:
    - Five flight modes, five flight states, and four box types
    - The others are the same as the test suite 3
5. Test suite 5 consists of 2880 tests with the following values:
    - Five flight modes, six flight states, and six box types
    - The rests are same as the test suite 4

4.2.3 Test Results

In the evaluation, we randomly select the order of tests to execute with the testing function until the threshold has been met (100% coverage in this case). Since our test cases are randomly selected, each test case has been generated and executed three times resulting in the average times listed. Thus, the test results in this table are the average values of the three random sets of test cases as shown in Table 8.

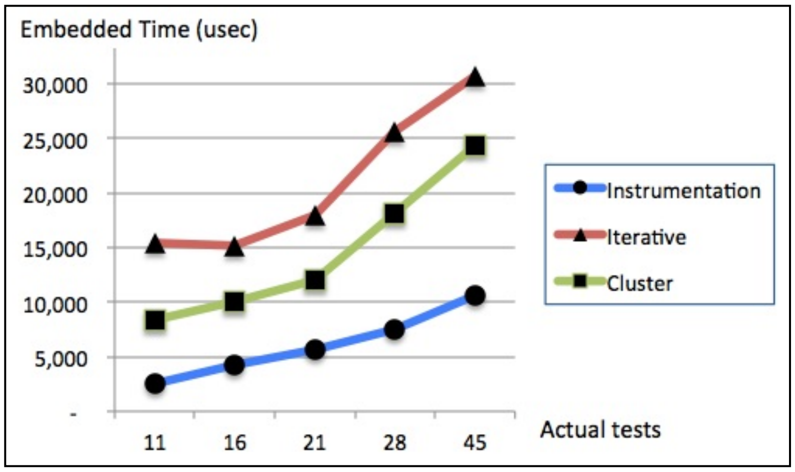
Table 8. Node coverage results of the pidLuxFloat() function evaluated on three code coverage techniques: traditional instrumentation, iterative instrumentation, and cluster iterative instrumentation

Actual Tests	Instrumentation		Iterative instrumentation		Compared to Instrumentation Increased by		Cluster iterative instrumentation		Compared to Instrumentation Increased by		Cluster iterative Vs. Iterative Reduced by	
(all tests)	Emb. (usec)	User (msec)	Emb. (usec)	User (msec)	Emb.	User	Emb. (usec)	User (msec)	Emb.	User	Emb.	User
11 (128)	1,459	7.78	11,070	66.74	86.82%	88.34%	6,865	39.67	78.75%	80.39%	37.99%	40.56%
16 (288)	1,878	9.40	11,432	73.34	83.57%	87.18%	7,131	40.65	73.66%	76.88%	37.62%	44.57%
21 (768)	4,394	21.89	16,880	102.81	73.97%	78.71%	11,353	65.99	61.30%	66.83%	32.74%	35.81%
28 (1600)	9,658	45.44	24,604	128.63	60.75%	64.67%	16,919	99.59	42.92%	54.37%	31.23%	22.58%
45 (2880)	10,094	49.00	32,025	177.65	68.48%	72.42%	24,011	131.43	57.96%	62.72%	25.02%	26.02%

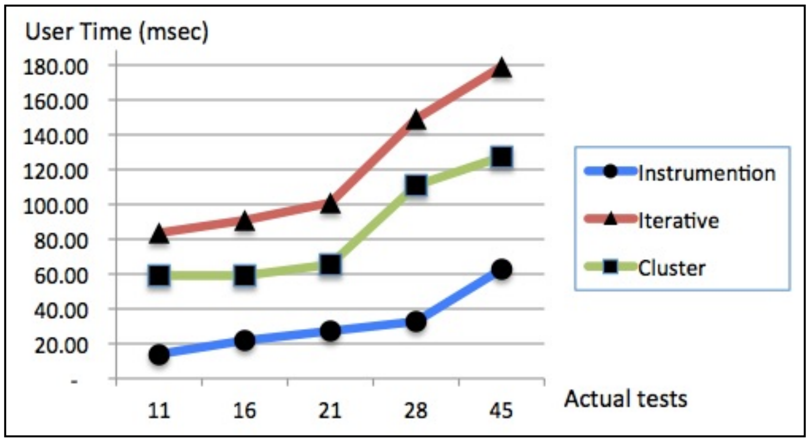
Table 8 compares the node coverage results on the pidLuxFloat() function that are evaluated by three code coverage techniques: instrumentation, iterative instrumentation, and cluster iterative instrumentation. In the first column, the number of actual tests for each test case is less than all possible tests, as the evaluation is terminated when the coverage percentage has met the threshold (100% in our case study). Columns 2 and 3 are the execution times of instrumentation; columns 4 and 5 are the execution times of iterative instrumentation; columns 6 and 7 are the increased execution times of iterative instrumentation compared to instrumentation; columns 8 and 9 are the execution times of cluster iterative instrumentation; columns 10 and 11 are the increased execution times of cluster iterative instrumentation compared to instrumentation; the last two columns are the reduced execution times of cluster iterative instrumentation compared to iterative instrumentation.

In this table, there are two types of execution times: embedded time (the time from the first node to the last node in a test path) and user time (the time from a test case sent to the SUT to the time the test result sent back to the unit testing framework). The embedded times are measured in microseconds (usec) by real-time clocks on non-OS platforms, and by the embedded system clocks with the POSIX gettime() function on OS platforms. The user times are measured in milliseconds (msec), because they include the transfer times (both test case data and test results). Unlike the embedded times, the user times are measured by the time() and clock() functions written in Python on the host machine (time() is on Linux/Mac and clock() is on Windows systems). Thus, the values of the user times are not as accurate as the embedded times.

As expected, the execution times of both iterative instrumentation and cluster iterative instrumentation are higher than instrumentation due to multiple executions of variants for each test case. In this case, iterative instrumentation increases the execution times up to 86% (embedded time) and 88% (user time), while cluster iterative instrumentation increases the execution times up to 78% (embedded time) and 80% (user time). However, cluster iterative instrumentation is more efficient than iterative instrumentation, because it has fewer nodes to execute. Based on the results, cluster iterative instrumentation can reduce the execution times up to 37% (embedded time) and 40% (user time) compared to iterative instrumentation.



(a)



(b)

Figure 15. Compare the relations between the number of actual tests and execution times: (a) embedded times and (b) user times, evaluated by three code coverage techniques: instrumentation, iterative instrumentation, and cluster iterative instrumentation

Figure 15 compares the relations between the number of actual tests and execution times evaluated by three code coverage techniques. Figure 15 (a) shows the trend of the embedded times and Figure 15 (b) shows the trend of the user times. Although the user times are not as accurate as the embedded times, both graphs depict the similar trends indicating that the more the actual tests, the higher the execution times.

Additionally, when the number of actual tests is higher, the gaps between the iterative approaches (i.e., iterative instrumentation and cluster iterative instrumentation) and traditional instrumentation are larger. Since the trends of both iterative instrumentation and cluster iterative instrumentation grow exponentially, they indicate that these approaches could have scalability and performance issues when the number of actual tests is larger.

4.3 Threats to Validity

We explain both internal and external threats to validity, and we also discuss how these threats could be alleviated as follows.

4.3.1 Internal Threats to Validity

There are internal threats to validity in our case study. First, our evaluation is based on the PID function of the Cleanflight project. However, here are also other functions from different modules (e.g. drivers, IO, and sensors modules) in this project that might produce broader results of the experiment. Additionally, the number of tests in this case study may not be large enough for predicting trends of the execution times for both iterative instrumentation and cluster iterative instrumentation. Having a larger number of tests could increase the quality of the evaluation resulting in more precise prediction. Lastly, the embedded hardware used in the case study is limited to a single platform (i.e., Beaglebone Black) that has specific hardware components: processors, develop tools, hardware interfaces, and loading methods. Using a wide variety of embedded platforms in the test would produce the test results differently (e.g., the transfer times between Ethernet and WIFI are much different).

4.3.2 External Threats to Validity

We try to generalize our evaluation by using an open source embedded project instead of our own application used in previous case study (in Chapter 3). However, this embedded software is only from one domain from real world applications, so it might not be representative of all embedded applications. Nevertheless, in Chapter 6, we evaluate our case study with more types of software modules in this project (i.e., flight, I/O, and sensors modules) that could mitigate these threats.

4.4 Discussion

In this section, we discuss the trade-off analysis and possible improvements of cluster iterative instrumentation.

4.4.1 Trade-off Analysis

Cluster iterative instrumentation has both advantages and disadvantages. First, this approach is more efficient than iterative instrumentation, because the number of variants is reduced resulting in less execution time compared to iterative instrumentation. From our test results, the nodes in our cases study was reduced by 48% resulting in less execution time up to 37% (embedded time) and 40% (user time).

However, despite being more efficient than iterative instrumentation, cluster iterative instrumentation is still expensive compared to traditional instrumentation. This problem exists in general with mutation analysis, as the concept of iterative instrumentation is adapted from weak mutation that each test case needs to execute with multiple variants. Therefore, many researchers try to reduce this cost using parallel execution [40, 58-59, 68], and we apply the parallel mutation testing from [40] to our framework in Chapter 5.

4.4.2 Possible Improvements

Although our clustering algorithm regarding common nodes is simple to understand, the time complexity is $O(n^2)$. To improve the efficiency of this algorithm, we may redesign our algorithm using more efficient technique like Lengauer and Tarjan algorithm proposed in [72]. Their algorithm uses depth first search to check the dominators of each node, so the complexity of the algorithm is only $O(n \log n)$. However, this algorithm is much more complicated than ours, and it is difficult to prove that this can replace our method.

Despite being more efficient than iterative instrumentation, the trends of the graphs in Figure 15 (a) and Figure 15 (b) indicate that cluster iterative instrumentation could still have scalability and performance issues when the number of actual tests is larger. Additionally, traditional instrumentation clearly outperforms cluster iterative instrumentation. In order to use cluster iterative instrumentation in practice, we still need to improve the performance of this technique to be as equal as traditional instrumentation. Because the cluster iterative instrumentation technique has already reduced the number of nodes from a CFG used by the iterative instrumentation technique, we consider parallel computing for multiple variants as a possible solution. As all variants are independent, we can execute them on multiple embedded platforms simultaneously. In Chapter 5, we present parallel embedded testing by executing several variants on a cluster of embedded platforms along with a case study comparing the execution times from different numbers of computing nodes.

CHAPTER 5: PARALLEL EMBEDDED TESTING

Although cluster iterative instrumentation can improve the efficiency of iterative instrumentation, this approach still has a drawback of the expensive cost resulting from the execution of multiple variants. The results in Case Study 3 indicate that the total execution time can increase more than linearly as the number of variants increases. Therefore, the cluster iterative instrumentation technique proposed in the previous chapter could have scalability issues, if the SUT is larger (i.e., a large number of variants to execute for each test).

In this chapter, we tackle the scalability issues of cluster iterative instrumentation by applying parallel execution to our testing framework. Since all variants are independent from each other, we can execute multiple variants at the same time. Next, we evaluate our parallel unit testing framework through Case Study 4 comparing the execution times from different numbers of embedded platforms (i.e., parallel computing nodes). Lastly, we discuss trade-off analysis and possible improvements of our parallel unit testing framework.

5.1 Parallel Unit Testing for Embedded Systems

Inheriting the property of mutation analysis, cluster iterative instrumentation is expensive due to the execution of multiple variants. There are several researchers [40, 58-59, 68] that try to reduce this cost by using parallel mutation testing on different types of systems. Mateo and Usaola [40] execute multiple mutants (i.e., variants) on a cluster of

PCs; Krauser and Mathur [58] execute a set of mutants on a SIMD machine; Offutt et al. [59] and Choi and Mathur [68] use a MIMD machine to execute mutants simultaneously. In addition to the parallel processing units, these researchers also use various methods for managing the mutants to be executed (e.g., FIFO, random, and uniform distribution scheduling). The results in their experiments indicate that using dynamic distribution algorithm with parallel execution can provide an optimal solution.

Although the current technologies allow us to execute multiple variants using multithreading or multiprocessing on a single multi-core machine, we use a cluster of embedded platforms to execute the parallelizable variants (one variant on each embedded platform) due to two main reasons. First, real-time embedded applications usually require real interactions with hardware modules, so the testing unit must have its own resources including hardware peripherals connected to the testing platform during the test. Second, the cost of existing embedded platforms is low (e.g., the current price of the latest Raspberry Pi platform is only $35), so we can build a cluster of these low-cost embedded platforms with a small budget.

Figure 16 is a diagram of the parallel iterative instrumentation technique that is redesigned from the iterative instrumentation diagram presented in Section 3.2.2. Although most components in this diagram are similar to the previous version, there are two differences here. First, there are a group of embedded platforms instead of a single platform interacting with the variant executor, as we need to execute multiple variants on a cluster of platforms simultaneously. Second, there is one additional component in the variant executor called a "variant tracker" that manages all unkilled variants to execute with the current test case on the embedded platforms.

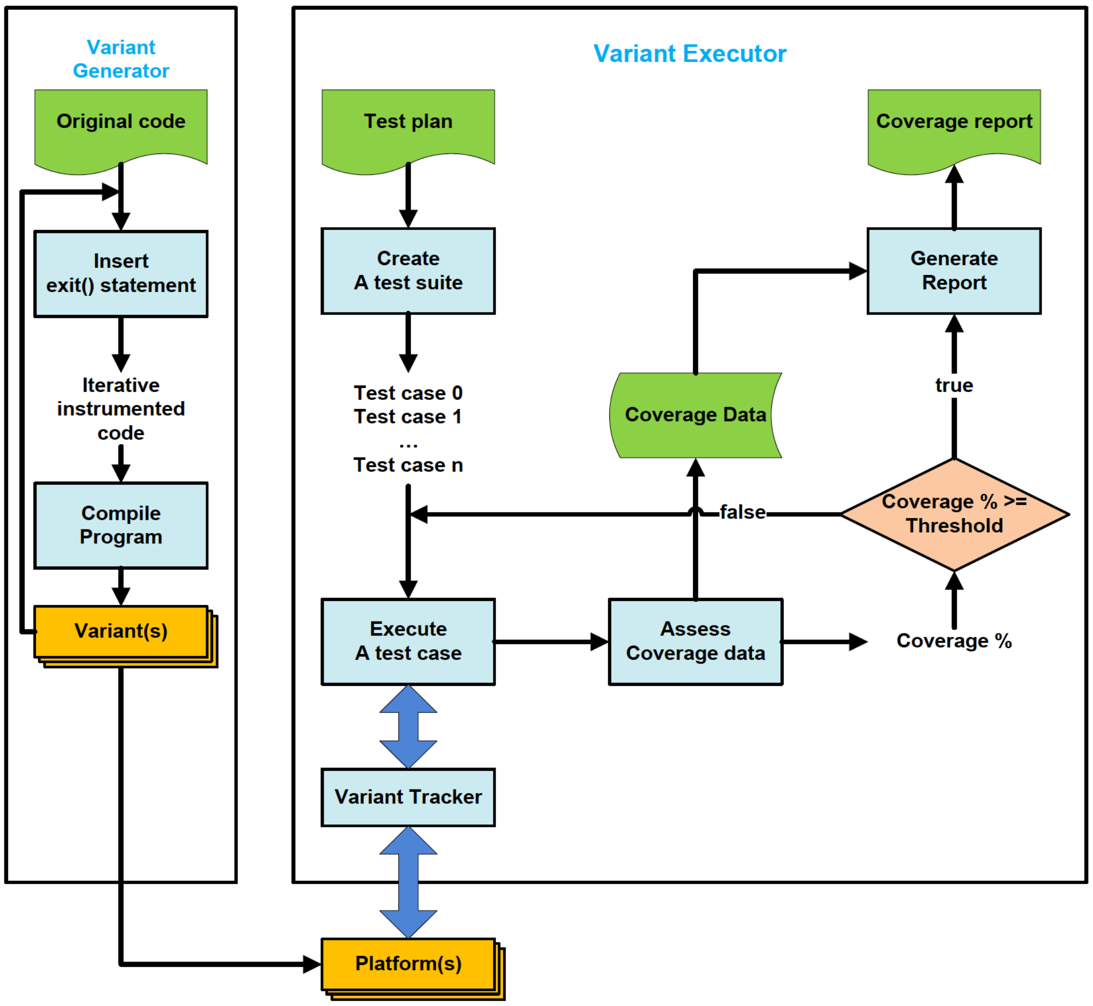


Figure 16. A diagram of the parallel iterative instrumentation technique

Since most components in the diagram are identical to the iterative instrumentation diagram, we only describe the variant tracker in this section. The variant tracker manages the parallel execution of variants on the cluster using two independent queues: a job queue and a worker queue. All unkilled variants are stored in the job queue, while the cluster platform identifiers (IDs) of the available platforms are stored in the worker queue. In this version, we only use FIFO job scheduling for managing variants (i.e., jobs) to be executed on the embedded platforms, because we need a variant to be executed on each embedded platform without any preemption. Additionally, the execution time (i.e., embedded time) on the platform is dominated by the transfer time that is unpredictable as the embedded platforms use shared network communication, so there is no need to determine the order of the variants by their execution times.

There are many steps at the variant tracker. Initially, all unkilled variants are appended into the job queue and all platform IDs are added into the worker queue. Whenever there is an available platform in the worker queue, the variant tracker assigns the first variant in the job queue to be executed by that embedded platform. At the end of the execution, the variant returns a test result back to the variant tracker and the ID of the executing platform is put back to the worker queue. When the job queue is empty (i.e., all unkilled variants are executed), the variant tracker sends all test results to the variants executor. After the test results are evaluated, the unkilled variants are updated, and the remaining unkilled ones are loaded into the job queue. The variant tracker keeps repeating these steps until the coverage threshold has been met or all test cases are executed with all variants.

5.2 Case Study 4: Parallel Iterative Instrumentation

This section evaluates the performance of the parallel iterative instrumentation technique through a case study of the PID controller of the cleanflight project described in the previous chapter. To increase the quality of the evaluation, we generate larger test cases than the ones used in the Case Study 3. We also discuss the test results at the end of this section.

5.2.1 Evaluation Setup

In this case study, we create a cluster of sixteen embedded platforms (i.e., Raspberry Pi 3 platforms), which are connected to the host PC through a local area network. In Figure 17, two stacks of embedded platforms are connected to the network switch through their on-board Ethernet interfaces. With the network switch properties, these platforms can have individual network links that enables equal network capacity to all of them. Additionally, all embedded platforms are configured with the same working environments (i.e., OS, dependencies, and system configurations) including the generated variants on them. Since each platform has the same set of variants on it, the variant tracker can remotely execute a variant on any platform during the test.

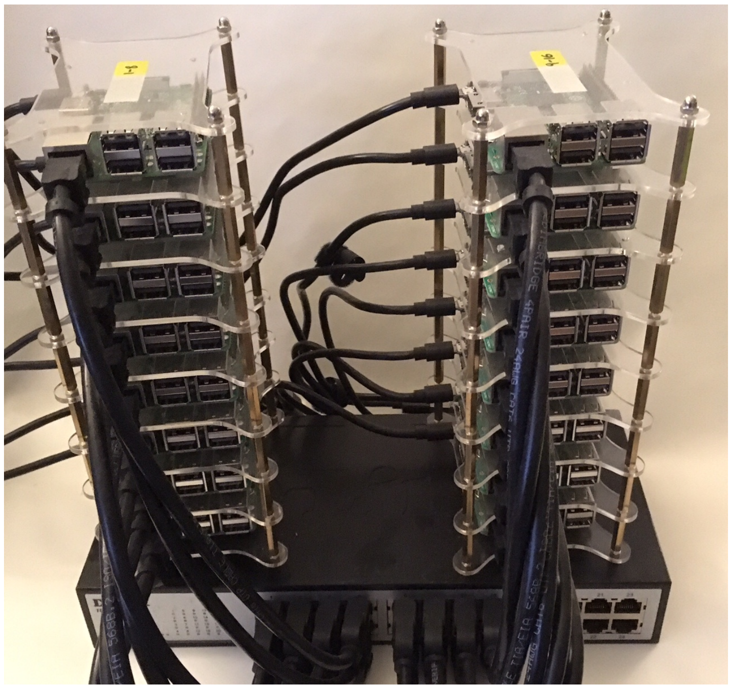


Figure 17. Parallel iterative instrumentation setup with a cluster of Raspberry Pi 3

In addition to the hardware setup, we also generate a range of test suites of the cleanflight PID controllers: 128, 288, 768, 1600, 2880, 8640, 9600, 38400, 96768, and 153600 tests respectively. These test cases are generated by the combination of four partitions of input parameters: twelve flight modes, six flight states, thirty box (component) types, and thirteen PID parameters. The variant executor will randomly select at runtime until the threshold is met or all tests are executed with the variants. Moreover, we run each test case with different numbers of parallel nodes enabled: 1, 2, 4, 8, and 16 nodes specified by the last command line argument of the test command.

5.2.2 Test Results

In this case study, we evaluate both non-parallel (1 node) and parallel versions (2, 4, 8, and 16 nodes) for the iterative instrumentation and cluster iterative instrumentation techniques. Because the embedded times (the time from the first node to the last node in the test path executed) are dominated by the testing times (the time from a test data sent to the platform to the time the test result sent back to the variant tracker), we only compare the testing times of these experiments.

Table 9 compares the testing times for measuring node coverage by the iterative instrumentation technique on different numbers of parallel-embedded platforms: 1, 2, 4, 8, and 16 nodes. The first column is the number of actual tests (i.e., the number of tests to achieve the coverage percentage of 100%) and the number of all possible tests. The remaining columns are the testing times (in seconds) for measuring the node coverage for each test suite by different numbers of nodes (1, 2, 4, 8, and 16 nodes respectively).

Table 9. The Iterative instrumentation node coverage results on different numbers of parallel-embedded platforms: 1, 2, 4, 8, and 16 nodes respectively

Actual Tests (all tests)	**Parallel Iterative Instrumentation Testing Times (seconds)**								
	n = 1	**n = 2**	**% Reduction from n=1**	**n = 4**	**% Reduction from n=1**	**n = 8**	**% Reduction from n=1**	**n = 16**	**% Reduction from n=1**
11 (128)	15.13	8.29	45.22%	4.82	68.13%	3.42	77.38%	2.86	81.10%
16 (288)	16.12	9.25	42.60%	6.04	62.55%	4.57	71.65%	4.02	75.04%
21 (768)	17.65	10.89	38.28%	7.40	58.07%	5.87	66.76%	5.20	70.54%
28 (1600)	25.19	14.39	42.89%	9.77	61.23%	7.52	70.16%	6.87	72.72%
45 (2880)	29.72	19.01	36.05%	13.95	53.08%	11.41	61.60%	10.50	64.67%
77 (8640)	38.78	26.9	30.63%	20.84	46.27%	18.35	52.69%	17.2	55.64%
96 (9600)	43.8	31.75	27.52%	24.53	43.99%	21.76	50.31%	21.14	51.73%
113 (38400)	45.49	33.82	25.66%	26.28	42.23%	24.32	46.55%	23.76	47.78%
146 (96768)	52.37	40.75	22.19%	33.82	35.42%	32.09	38.73%	31.48	39.90%
170 (153600)	53.96	44.04	18.38%	38.8	28.10%	36.79	31.82%	36.14	33.02%

From the test results in Table 9, the times are significantly reduced when the number of parallel nodes is increased from 1 to 2, 4, and 8 nodes respectively. The results indicate that the performance of the iterative instrumentation technique can be improved with more parallel computing nodes. However, although the times of all test suites are reduced when the number of parallel nodes is increased, the reduction percentage of each test is lower than our expectation. Theoretically, when the number of parallel nodes is increased to 2, 4, 8, and 16 nodes, the expected reduced values (%) should be 50.0%, 75.0%, 87.5%, and 93.75% respectively. The reason that our parallel unit testing is not as efficient as expected, is due to limitations in our current implementation of the scheduling algorithm that we only execute variants for each test case. When the number of unkilled variants is lower than the number of available parallel platforms, the algorithm does not utilize the remaining platforms for executing other variants of the next test case. This clearly needs to be addressed as our future work.

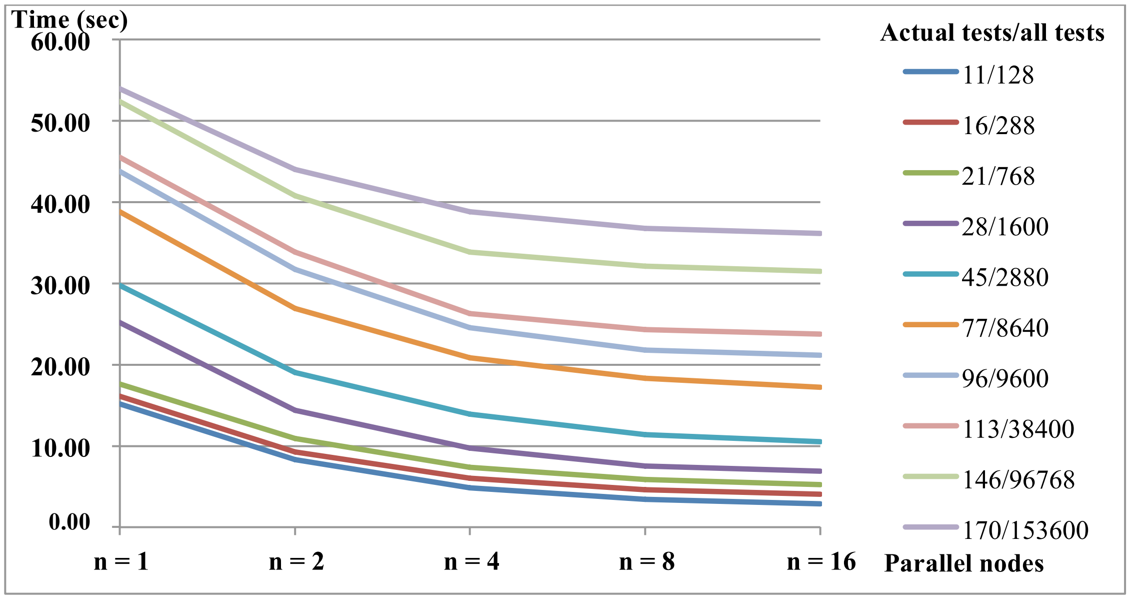


Figure 18. Trends of the parallel iterative instrumentation testing times when the number of embedded platforms increased

Figure 18 shows the trends of the parallel iterative instrumentation testing times when the number of embedded platforms is increased: 1, 2, 4, 8, and 16 nodes respectively. Since we have ten test cases from different numbers of actual tests: 11, 16, 21, 28, 45, 77, 96, 113, 146, and 170 tests, there are ten lines in the figure. Although the execution times of these line graphs are varied due to the different numbers of actual tests, all of them have similar directions that the times are lower noticeably when the number of nodes is increased to 2 and 4 nodes. As mentioned earlier, the trends of all test cases are almost the same when the parallel nodes are increased from 4 to 8 nodes and 8 to 16 nodes. This confirms our analysis that the optimal number of nodes in this experiment is around 4 nodes. Although the 8-node cluster is more efficient than the 4-node cluster, the performance is not as high as when the numbers of nodes increased from 1 to 2 nodes or from 2 to 4 nodes respectively.

Table 10. The cluster iterative instrumentation node coverage results on different numbers of parallel-embedded platforms: 1, 2, 4, 8, and 16 nodes respectively

Actual Tests (all tests)	**Parallel Cluster Iterative Instrumentation Testing Times (seconds)**								
	n = 1	**n = 2**	**% Reduction from n=1**	**n = 4**	**% Reduction from n=1**	**n = 8**	**% Reduction from n=1**	**n = 16**	**% Reduction from n=1**
11 (128)	10.37	5.88	43.32%	3.66	64.72%	2.79	73.07%	2.53	75.61%
16 (288)	11.00	6.51	40.87%	4.84	56.01%	3.91	64.44%	3.71	66.31%
21 (768)	12.29	8.24	32.94%	6.15	49.92%	5.01	59.22%	4.80	60.93%
28 (1600)	18.07	10.87	39.83%	7.97	55.91%	6.83	62.22%	6.33	64.95%
45 (2880)	22.00	15.04	31.64%	11.95	45.69%	10.63	51.69%	9.97	54.67%
77 (8640)	28.70	21.87	23.80%	18.38	35.94%	16.88	41.17%	16.44	42.70%
96 (9600)	33.63	24.78	26.31%	22.36	33.53%	20.91	37.82%	20.43	39.27%
113 (38400)	34.98	27.84	20.42%	25.35	27.52%	24.27	30.62%	24.05	31.25%
146 (96768)	42.55	38.30	18.38%	32.22	24.86%	31.10	27.38%	30.73	27.75%
170 (153600)	49.34	41.43	16.02%	37.44	24.11%	36.40	26.23%	35.97	27.10%

Table 10 shows the cluster iterative instrumentation node coverage results on different numbers of parallel-embedded platforms: 1, 2, 4, 8, and 16 nodes, that test against different numbers of actual tests: 11, 16, 21, 28, 45, 77, 96, 113, 146, and 170 tests. The formats of this table are the same as the ones in Table 9, but the testing times are a lot lower than those as we know that the cluster iterative instrumentation technique is more efficient. Similar to the parallel iterative instrumentation testing, the reduction percentage of parallel cluster iterative instrumentation is not equal to the expectation due to the current implementation of our scheduling algorithm. Additionally, the efficiency between the cluster iterative instrumentation and iterative instrumentation techniques are not much different when we increase the parallel nodes from 4 to 8 nodes and from 8 to 16 nodes.

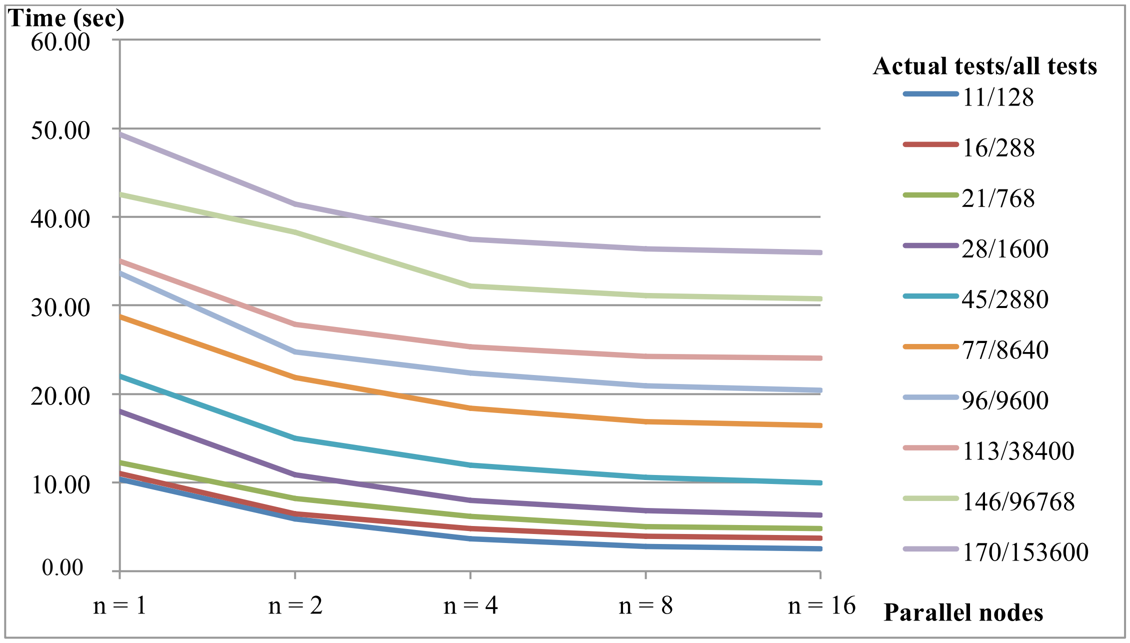


Figure 19. Trends of the parallel cluster iterative instrumentation testing times when the number of embedded platforms increased

Figure 19 demonstrates the trends of the parallel cluster iterative instrumentation testing times when the number of embedded platforms increases: 1, 2, 4, 8, and 16 nodes respectively. Similar to the trends in Figure 18, the lines in this figure show the same direction that the performance can be improved significantly when increasing the parallel nodes up to 4 nodes. However, it is not much improved when the number of nodes is increased more than 4 nodes.

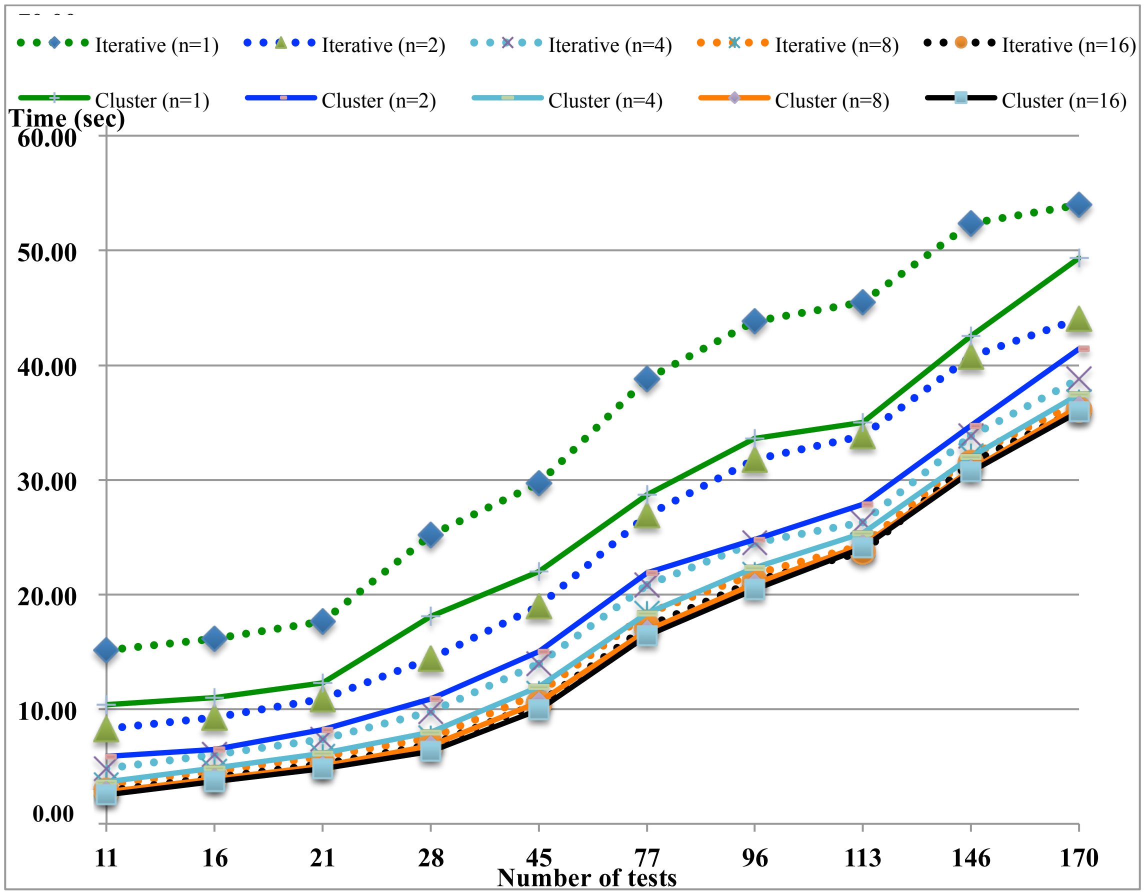


Figure 20. Compare the testing times between parallel iterative instrumentation and parallel cluster iterative instrumentation

To confirm this analysis, Figure 20 compares the execution times between the iterative instrumentation and cluster iterative instrumentation techniques when the number of parallel nodes increased. From the graphs, the cluster iterative instrumentation technique outperforms the iterative instrumentation technique when the numbers of parallel nodes are less than 4. Moreover, the efficiency of the cluster iterative instrumentation technique is equal to the performance of the iterative instrumentation version while using the number of parallel nodes only half of the nodes used by the iterative one. For example, the performance of the cluster iterative instrumentation with a single node (solid green line) is a little less than the performance of the iterative one with two parallel nodes (dash blue line) and the performance of the cluster version with two parallel nodes (solid blue line) is as equal as the performance of the iterative technique with four parallel nodes (dash light blue line). However, the efficiency between these techniques is almost the same when using 8 and 16 parallel nodes.

5.3 Threats to Validity

We explain both internal and external threats to validity, and we also discuss how these threats could be mitigated as follows.

5.3.1 Internal Threats to Validity

Same as the previous case studies, the generated test case values are controlled by our test case generator that randomly selects the data from a set of valid values for each partition. However, this case study only evaluates the performance of the parallel unit testing on node coverage that may not sufficient for predicting the performance trends. To improve this, we can add more complicated code coverage criteria (e.g., edge, logic, and path coverage) to the case study in our future work.

In addition to the node coverage criteria, data communication used in this case study is also limited to local area network (LAN). In the case study setup, we simply set static IP addresses for all nodes and each node connects to the LAN through a network switch. Since the link between each node to the host is direct (not a shared data link), we do not perform load analysis across nodes during the evaluation. However, if the data communication uses shared networks (e.g., WIFI or LAN through a hub), such analysis should be considered.

5.3.2 External Threats to Validity

Although we use larger test suites in this case study to improve the quality of our experiment, this case study still evaluates only one software module (i.e., a PID controller of the Cleanflight project). To generalize the evaluation, we could test our unit testing framework with different types of software modules. In the next chapter, we evaluate our unit testing framework with four different Cleanflight modules (i.e., flight controller, drivers, IO, and sensors).

In addition to the software modules, the hardware platform used in this case study may be too specific, as the cluster is built from the Raspberry PI platform only. To mitigate this threat to validity, we evaluate our unit testing framework with seven different embedded platforms in the next case study (in Chapter 6). Additionally, the hardware interface used in this experiment is limited to Ethernet connection only, so this might affect the testing times measured by the transferred times of the test input and test results. To alleviate this threat, we use different hardware interfaces between the host PC and the testing platforms in the next case study (e.g., UART, WIFI, on-board Ethernet, and Virtual Ethernet).

5.4 Discussion

In this section, we analyze the trade-off analysis of the parallel unit testing concept, and we also discuss possible improvements of this approach.

5.4.1 Trade-off Analysis

Using parallel execution for multiple variants can solve the performance and scalability issues of the iterative instrumentation technique. First, parallel computing improves the efficiency of the iterative instrumentation and cluster iterative instrumentation techniques, as it allows multiple variants to test on a cluster of embedded platforms at the same time resulting in lower total testing time. From Figure 20 in our case study, when the number of parallel nodes increases, the execution times of both iterative instrumentation and cluster iterative instrumentation techniques are rapidly decreased. Particularly, when the numbers of nodes increase to 2 and 4 nodes, the execution times are almost reduced by half. This can save developer time by allowing them to work on other tasks earlier. In particular, if the test cases are extremely large (i.e., millions of test cases), they can cause the testing times up to hours.

Second, this approach can improve the scalability issue especially in large programs (i.e., the size of variants is much higher). In the case study, the trends of execution times in both Figure 18 and Figure 19 are similar by quickly decreased when using parallel execution. More precisely, when the numbers of nodes increase from 1 to 2 nodes and from 2 to 4 nodes, the execution times are decreased more than linearly. This provides the scalability to the unit testing framework, as its performance can be maintained while a large number of nodes added to the testing code.

Despite having the advantages in efficiency and scalability, parallel unit testing can have some drawbacks. First, the design and implementation are more complicated, because we need to properly manage multiple jobs to be executed at the same time. Thus, job scheduling must be used fairly. Additionally, if the jobs share some resources (e.g., shared data) during the tests, resource management like locking is required. If the results are to be constructed in the same order as the input test cases, data synchronization is also needed.

Second, the data communications could be the bottleneck of the system. This depends of what types of communications will be used. In our case study, we use Fast Ethernet connection that is efficient enough for our test. However, if we use low-speed communications (e.g., UART or other low-speed serial communications) due to the hardware specifications of particular embedded platforms, the performance of the whole system could be worse.

Lastly, the cost of the parallel unit testing is more expensive than a simple version. Since this concept uses multiple embedded platforms to execute code simultaneously, there are extra costs for the additional devices, communication cables, hubs/switches, and power adapters. In addition to the hardware costs, energy consumption is also higher for the parallel-embedded platforms. At least, the energy from multiple nodes is increased (up to the number of nodes), and it can be higher from additional devices (e.g., hubs/switches for network connections).

5.4.2 Possible Improvements

Although the concept of parallel unit testing can provide both scalable and efficient properties to our unit testing framework, this framework can be improved in some aspects. First, the existing embedded platforms are varied, as different manufacturers build them. The different aspects among these embedded platforms will be a huge challenge for programmers to develop applications and test them across different platforms. To solve this issue, we redesign our unit testing framework that enables a tester using a single framework to run the same code on any embedded platforms by using runtime adapters and a runtime protocol. The details of this cross-platform unit testing framework are explained in the next chapter along with a case study evaluating the effectiveness of our framework.

Second, our current version of the parallel unit testing only runs one job on each embedded platform, because our main purpose is to test jobs independently and provide them sufficient and equal resources. In particular, if the testing code needs to work with any hardware components on the platforms, it may be difficult to share this among other jobs on the same platform at the same time. However, to utilize the resources of the embedded platforms more efficiently, we can redesign the framework to allow multiple jobs to be executed on each platform (e.g., the platforms have multicore processors and the real-time OS supports multiprocessing/multithreading). Although the framework will be more complicated, we may consider this in our future work.

Lastly, although the scheduling algorithm used in the case study is simple and effective, its performance is lower than our expectation. This results from the limitation of the current design and implementation in that it only allows variants of the same test case to run on the parallel platforms concurrently. To improve the efficiency of parallel unit testing, we could redesign and implement the scheduling algorithm by allowing variants of test cases to run on any available platforms in the worker queue. We will add this in our future work list.

CHAPTER 6: CROSS-PLATFORM EMBEDDED TESTING

From the literature review presented in Chapter 2, most unit testing frameworks [2, 6-9] do not address the memory-constrained property of embedded systems. Thus, testers can only test embedded applications on a PC instead of real embedded platforms with traditional unit testing frameworks. Although the frameworks from [5, 10, 35] try to deal with this issue by minimize the code to fit small memory footprints of embedded systems especially non-OS embedded platforms, these do not directly support cross-platform testing. This challenge inspires us to introduce a cross-platform unit testing framework for enabling testers to test the same code across different embedded platforms. While most embedded systems target a specific platform, utilizing a standard cross-platform testing framework allows developers to quickly evaluate alternatives and reduce business risks associated with becoming locked-in to a specific platform.

6.1 A Cross-platform Unit Testing Framework for Real-time Embedded Systems

We present a cross-platform unit testing framework called XEUnit, which is capable of evaluating both black-box and white-box test adequacy criteria in C code running on a wide variety of embedded platforms. This framework adapts the concept of Boydens et. al in [35] along with our own techniques in particular runtime adapters and runtime protocol.

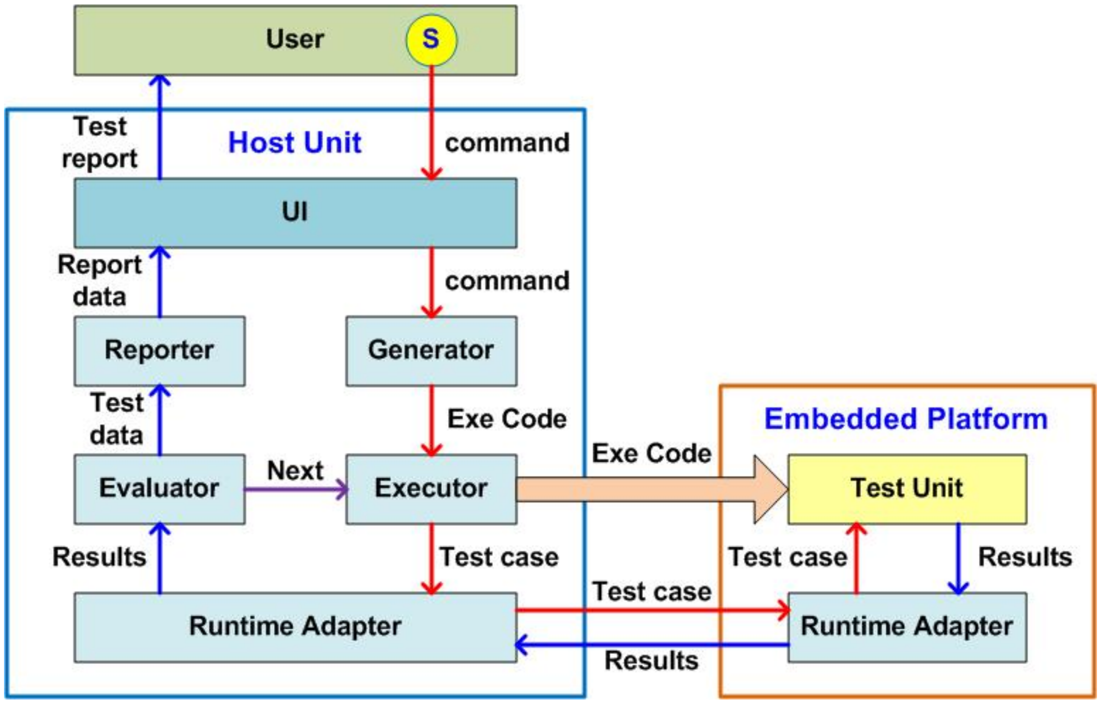


Figure 21. XEUnit framework system diagram

As shown in Figure 21, the XEUnit framework is separated into two parts: the host sub-system (left) running on a PC and the test sub-system (right) running on an embedded platform. The host module consists of six components: user interface, code generator, code executor, runtime adapter (RA) server, evaluator, and reporter. This module can run on any recent operating systems (e.g., Windows, Linux, and Mac OS X), as it is written in Python, a scripting language supported by most operating systems. In contrast to the host sub-system, the test sub-system only includes the RA client to the testing code, as the code can be fit to small memory footprints of memory-constrained embedded platforms.

6.1.1 User Interface (UI)

In this project, we use a command-line user interface (UI) for XEUnit (the component receiving a command from a user in Figure 21). This component simply receives a user command and validates it. As the current version of XEUnit can evaluate three types of testing: black-box testing, traditional code coverage, and iterative instrumentation code coverage (time-sensitive applications), the command usage is as follows:

python xeunit.py <platform name> <test type> [number of parallel nodes]

The Python script name is "xeunit.py" follows by a platform name, a test type, and the number of parallel nodes (optional) respectively. The latest version of XEUnit supports eight embedded platforms: ARM mbed, Atmel AVR, Intel Edison, Qualcomm DragonBoard, Microchip PIC32, Raspberry Pi, TI Beaglebone, and XMOS, and the three test types are "-b" (black-box testing), "-c" (traditional code coverage), "-i" (iterative code coverage). Please note that we use the cluster iterative instrumentation technique (in Chapter 4) instead of the original version (in Chapter 3) for the iterative code coverage test type.

For example, the command "python xeunit.py avr –i" is to evaluate iterative code coverage for Atmel AVR platforms without parallel nodes (the default value of the number of parallel nodes is 1 or no parallel nodes), and the command "python exunit.py rpi -i 4" is to measure iterative code coverage for Raspberry Pi platforms with 4 parallel nodes. XEUnit allows developers to easily add additional platforms by adding runtime adapters (i.e., RA client and RA server) for a new platform into the XEUnit framework. In addition to validating user commands, the UI also loads the test case data from a test case file and print the test report to the screen after receiving the report data.

6.1.2 Code Generator (CG)

The component that receives a valid command from the UI is the code generator (CG). The CG generates an executable code into one of the three versions based on the specified test type (i.e., black-box testing, traditional code coverage, or iterative code coverage). To generate executable code, XEUnit does not provide its own cross-compilers, but the CG selects the cross-compiler tools provided by the vendors of particular embedded platform at runtime. Therefore, all compilers and their configurations are specified within the XEUnit framework and defined in the configuration of particular platforms before the first use. The details of three test types are follows:

6.1.2.1 Black-box testing

The CG generates an executable code for the black-box testing by including the RA client with the original code to enable the testing code to interact with the RA server running on the host unit. This executable code is called a "test unit."

6.1.2.2 Traditional code coverage

Traditional code coverage is more complicated than the black-box testing. First the CG generates a control flow graph (CFG) of the original code. Next, it includes the RA client with the code and inserts additional instrumented instructions (e.g. insert a nc[i++]; statement at the beginning of each block for checking node/block coverage reached by a test case) to all nodes according to the generated CFG. Lastly, the instrumented code is compiled into a test unit. Please note that these steps are still managed by developers, but we could automate them in the next version.

6.1.2.3 Iterative code coverage

The iterative code coverage is the most complicated analysis, as the CG inserts only some exit statements in a clustered node (i.e., these exit statements represents the node locations from the original CFG before transforming into a cluster graph). Then, the CG compiles each instrumented code into executable code called a variant. In the original iterative instrumentation version proposed in Chapter 3, it requires the number of variants equal to the number of nodes in a CFG, which each variant represents a unique location in the CFG [36]. However, with the cluster version presented in Chapter 4, the number of variants is reduced by grouping nodes never executed by the same test case. This can improve the efficiency of the iterative instrumentation technique as demonstrated in the Case Study 3 of Chapter 4. Therefore, the number of variants generated by the CG for the iterative code coverage is the same number of the clustered nodes transformed by our cluster iterative instrumentation algorithm (explained in Section 4.1.2).

6.1.3 Code Executor (CE)

Upon receiving executable code from the CG, the code executor (CE) selects a loading method for a particular platform to properly load that code to execute on the embedded platform. For example, it uses an uploader application for a non-OS platform to upload the code into the platform and reset/restart the code after the RA server is ready to interact with the RA client running on the test unit. For the OS platforms, the CE remotely connects to the test platform and loads the code to it through a network protocol. Because each OS platform usually has different network configurations (e.g., host name, user name, and password), the CE will select the network configurations (predefined in the platform's configuration in the XEUnit framework) for the testing embedded platform at runtime.

6.1.4 Runtime Adapter (RA)

The runtime adapter (RA) is the most important component, as it enables XEUnit to test code across different platforms. In Figure 22, the RA component consists of two parts: the RA server (left) and the RA client (right), because we need to keep the memory footprint of the test code as small as possible. The RA server is running on the host unit (PC), while the RA client is running on the test unit (embedded platform). Both RA client and server also consist of two main modules: runtime connector and runtime protocol as described next.

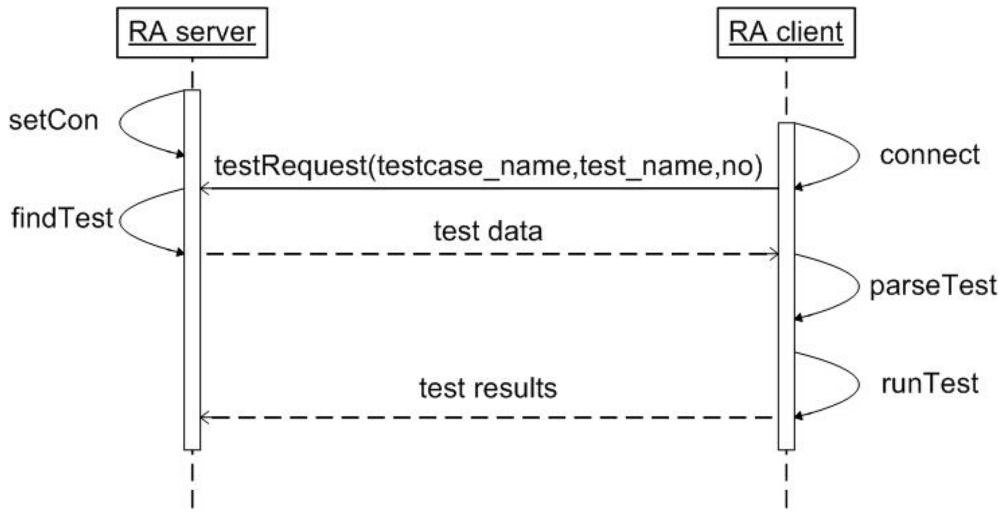


Figure 22. XEUnit runtime protocol sequence diagram

6.1.4.1 Runtime connector

The runtime connector dynamically selects the hardware interface and loading type of a particular platform defined in its configuration. For example, Atmel AVR uses the **atprogram** application to upload a binary code to the AVR platform through its

JTAG programmer. The connector also selects UART communication with the default baud rate of 9600 as a communication link between the RA client and RA server. In contrast, the OS platforms (e.g., Beaglebone Black, DragonBoard 410c, Intel Edison, and Raspberry Pi 3) can use SCP (Secure Socket Protocol Copy) to upload code to an embedded platform and use SSH (Secure Shell) to remotely execute code on the platform with different network configurations: host names, user names, and passwords.

6.1.4.2 Runtime protocol

The runtime protocol defines the interactions between the RA server and the RA client. The interactions for both OS platforms and non-OS platforms are identical, and the runtime protocol sequence diagram is shown in Figure 22.

In Figure 22, the RA server establishes a connection and waits for an incoming request from the RA client. When the test unit starts (on the embedded platform), the RA client sets up its connection and sends the RA server (host platform) a test request message consisting of three parameters: a test case name, a test name, and a test number, respectively. The RA server concatenates these parameters as a unique key called a test key for searching the test data in its hash table loaded from a test case file when XEUnit starts up. If the test key is found, the RA server replies with the test data for the RA client; otherwise it returns an error message. The test data consists of an expected output (oracle) followed by a list of input parameters. The RA client then parses the test data into the actual data, and passes it to execute on the testing unit. Since the RA client has its own timer (the embedded timer), it can precisely measure the execution time of the test (from the first node to the killed node or last node). Finally, the RA client sends the test results consisting of the output and the execution time back to the RA server to evaluate on the evaluator component next.

In contrast to many other unit testing frameworks, XEUnit uses JavaScript Object Notation (JSON) format, rather than Extensible Markup Language (XML) for storing data (i.e., platform configurations and test cases) and constructing the runtime protocol messages. Because JSON is simpler than XML, it uses a smaller structure to store data and its metadata is enough for describing all common data types (i.e., integer, floating point, string, list, and object) used in XEUnit [63-64]. This also keeps the memory footprint low on the test unit and simplifies the parsing process on both the client and server.

6.1.5 Evaluator

The evaluator receives the test results from the RA server (returned back from the RA client) and evaluates the results in two types: black-box testing and code coverage testing.

6.1.5.1 Black-box Testing

In the black-box testing, the test results are the actual output and the embedded time, so the evaluator can compare the actual output to the expected output defined in the black-box test cases. The test is passed if these values are matched, and it is failed if they are different. If all tests in a test case pass, that test case is passed; otherwise it is failed. Since black-box test cases in our examples are not large, we simply include these test cases in the test unit code like the other unit testing tools do. Nevertheless, XEUnit allows testers to define black-box test cases externally (in an external file read by the host sub-system) if needed.

Allowing users to store test cases externally has both advantages and disadvantages. The benefits of storing the black-box test cases externally are to enable the testers to change the test cases without recompiling the test unit and to keep the memory usage on the test unit low. However, the drawbacks of this method are the additional times: the loading time of the test cases on the host and the transmission time of the test cases from the RA server to the RA client. Since the transmission cost is much higher than the execution cost, storing the black-box test cases internally is recommended if they are rarely changed during the tests.

6.1.5.2 Code coverage Testing

Unlike the black-box testing, the code coverage testing does not require XEUnit to compare the actual output and expected output, since it only evaluates how many coverage points are reached by the test cases (code coverage techniques allow the tester to evaluate the quality of a test suite, once the test cases are established, they can be run in black-box mode to evaluate the correctness of the SUT). To check this, the RA client only returns the node identifier (a positive value for the killed variant or a negative value for the unkilled variant) along with the embedded time (from the start node to the killed node or last node) to the RA server. In the code coverage evaluation, the evaluator computes a coverage percentage.

The evaluator controls the whole testing process, because it keeps executing the specified test with different test data until the pre-defined threshold is met or all test data has been used. For the iterative code coverage, the evaluator must iteratively evaluate each variant with each test data and calculate the coverage percentage after the last variant has been tested. In a parallel testing mode, the variant tracker collects the test results from all executed variants through the RA server before sending these results to the evaluator.

6.1.6 Reporter

The last component is the reporter as it receives the test data from the evaluator to generate report data for the current test and sends the report data to the UI. The reporter uses the platform name and test type from the UI to properly generate a test report for a particular test. In this version, the report data is generated as a text file. This component is useful for the embedded platforms that cannot display the test results explicitly (e.g., most non-OS platforms usually send messages through serial ports or debugger devices). As future work, this component will support other report types widely used in software testing, such as, XML, JSON, and other standard formats, if needed.

6.2 Case Study 5: Black-box Testing on Seven Different Embedded Platforms

We demonstrate the effectiveness of XEUnit by evaluating the same set of black-box test cases on four Cleanflight units (i.e., flight altitude hold, flight PID, I/O serial, and sonar units) across seven different embedded platforms: ARM mbed LPC1768, Intel Edison, Microchip PIC32MX79, Raspberry Pi 3, Qualcomm DragonBoard 410c, TI Beaglebone Black, and XMOS XK-1A. Please note that we exclude the Atmel AVR platform from this case study, because the memory size of this platform is too small for the cleanflight modules to load.

Table 11. Four different black-box test cases from the cleanflight project

No	Test cases	Tests
1	Flight altitude hold	(1) Is thrust facing downward (2) Apply multirotor altitude hold
2	Flight PID	(1) PidLuxFloat (2) PidLuxFloat Integration for linear function (3) PidLuxFloat Integration for quadratic function (4) PidLuxFloat for I-term constrains (5) PidLuxFloat for D-term constrains
3	I/O serial	(1) Soft serial ports enabled (2) Soft serial ports disabled (3) Find port configuration
4	Sonar sensor	(1) Sonar constants (2) Sonar initialization (3) Sonar distance (4) Sonar altitude

6.2.1 Evaluation Setup

This case study evaluates the same black-box test cases on different embedded platforms. First, we generate the black-box test cases from four different cleanflight modules: flight altitude hold, flight PID, I/O serial, and sonar sensor. Each module consists of multiple tests as shown in Table 11. In the table, these test cases have different number of tests designed by the original cleanflight project [60]. As discussed in section 6.1.5.1 (Black-box Testing), these test cases are directly embedded in the test files before the test files have been compiled by the cross-compilers. Thus, this method does not have runtime overhead for transferring the test data like the code coverage test cases. However, it still needs to transfer the test results (i.e., actual output and embedded times of all tests) back to the XEUnit host after each test case finished.

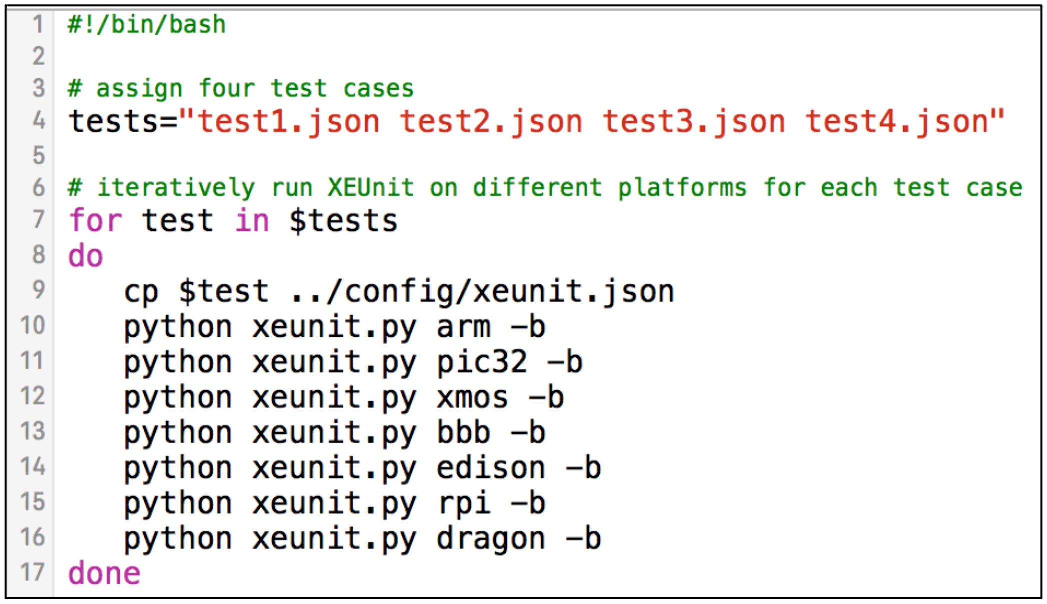
In addition to the test cases, we set up seven different embedded platforms to the XEUnit as shown in Table 12.

Table 12. Different embedded platform configurations used in the evaluation

No	Platforms	Interfaces	Load methods
1	ARM mbed LPC1768	mini USB	Copy and reset
2	Microchip PIC32MX79	mini USB	bootloader
3	XMOS XK-1A	USB to JTAG	xrun
4	Beaglebone Black	Virtual Ethernet (mini USB)	SCP and SSH
5	Intel Edison	WIFI	SCP and SSH
6	Raspberry Pi 3	Ethernet	SCP and SSH
7	DragonBoard 410c	USB to Ethernet	SCP and SSH

Table 12 shows the configurations from seven different embedded platforms: ARM mbed, Microchip PIC32MX79, XMOS XK-1A, Beaglebone Black, Intel Edison, Raspberry Pi 3, and DragonBoard 410c respectively. In the third column, these embedded platforms are connected to the XEUnit with different types of hardware interfaces (e.g., serial communication, WIFI, and Ethernet). In the last column, each non-OS platform uses different methods for loading the code to run, while all OS platforms simply use network protocols (i.e., SCP and SSH) to remotely load and run code.

In the evaluation, we use a script to automatically test the same test cases of four cleanflight modules on different embedded platforms as shown in Figure 23.



```
 1  #!/bin/bash
 2
 3  # assign four test cases
 4  tests="test1.json test2.json test3.json test4.json"
 5
 6  # iteratively run XEUnit on different platforms for each test case
 7  for test in $tests
 8  do
 9     cp $test ../config/xeunit.json
10     python xeunit.py arm -b
11     python xeunit.py pic32 -b
12     python xeunit.py xmos -b
13     python xeunit.py bbb -b
14     python xeunit.py edison -b
15     python xeunit.py rpi -b
16     python xeunit.py dragon -b
17  done
```

Figure 23. Unix shell scripts for running XEUnit on seven different platforms

Figure 23 shows the commands in Unix shell scripts. On line 9 the XEUnit configuration file (exunit.json) is replaced by each test configuration (i.e., test1.json, test2.json, test3.json, and test4.json respectively) defined on line 4. Then, from lines 10 to 16, a sequence of XEUnit commands is called to execute the black-box test cases on seven different embedded platforms. At the end of the tests, the test reports are generated in the reports directory under the XEUnit root directory. We discuss the test results in the next section.

6.2.2 Test Results

Since the current Cleanflight project uses Google Test for testing black-box units, we compare the executable file sizes of the original code to the Google Test and XEUnit code in Table 13.

Table 13. Comparison of the original code to Google Test code and XEUnit code

Test Units	**Original Code Sizes**	**Google Test Code**		**XEUnit Code**	
		Sizes	**Increased by**	**Sizes**	**Increased by**
Flight Altitude	31 KB	2.3 MB	98.68 %	33 KB	6.06 %
Flight PID	35 KB	2.6 MB	98.69 %	46 KB	23.91 %
I/O Serial	25 KB	2.3 MB	98.94 %	27 KB	7.41 %
Sonar	27 KB	2.4 MB	98.90 %	32 KB	15.63 %
Average	**29.5 KB**	**2.4 MB**	**98.80 %**	**34.5 KB**	**13.25 %**

Table 13 compares the original code to the Google Test and XEUnit code for four Cleanflight units (i.e. flight altitude, flight PID, IO serial, and sonar). The second column is the original code having the average size of 29.50KB. The third and fourth columns are the Google Test's code size and increased percentage compared to the original version in the first column. Because Google Test uses C++ standard libraries for storing data structures and printing results on screen, it has the average code size of 2.4MB that is greater than the original code by 98.80%. The last two columns are the XEUnit's code size and the percentage increase compared to the original code. Since XEUnit only includes a RA client having a small connector (e.g., network connector or UART connector) and an embedded timer, it increases the code size by just 13.25%. This enables XEUnit to combine with any testing code especially for memory-constrained embedded platforms.

Table 14. Black-box testing on four units (i.e., flight altitude, flight PID, IO serial, and sonar) across seven different embedded platforms (ARM mbed, Microchip PIC32, XMOS, Beaglebone Black, DragonBoard, Intel Edison, and Raspberry PI 3)

Test Units	**ARM**		**PIC32**		**XMOS**		**BBB**		**Dragon**		**Intel Edison**		**RPI**	
	UART 115200		**UART 115200**		**UART 115200**		**Virtual Ethernet**		**USB to Ethernet**		**WIFI**		**On-board Ethernet**	
	Emb (msec)	**User (usec)**	**Emb (msec)**	**User (usec)**	**Emb (msec)**	**User (usec)**	**Emb (msec)**	**User (usec)**	**Emb (msec)**	**User (usec)**	**Emb (msec)**	**User (usec)**	**Emb (msec)**	**User (usec)**
Flight Altitude	31	**0.952**	33	**1.189**	87	**1.250**	63	**0.123**	25	**0.062**	48	**0.234**	26	**0.031**
Flight PID	256	**5.136**	287	**6.503**	752	**6.915**	546	**0.667**	218	**0.334**	420	**1.267**	228	**0.167**
I/O Serial	11	**0.878**	13	**1.203**	33	**1.257**	24	**0.116**	10	**0.058**	18	**0.220**	11	**0.029**
Sonar	16	**1.149**	18	**1.417**	47	**1.596**	34	**0.148**	14	**0.074**	26	**0.281**	14	**0.037**

Table 14 demonstrates the effectiveness of the XEUnit framework by evaluating black-box testing on four Cleanflight test units across seven different embedded platforms. The first three platforms are non-OS platforms (i.e., ARM, AVR, PIC32, and XMOS), while the last four platforms are OS platforms (i.e., Beaglebone, DragonBoard, Intel Edison, and Raspberry PI 3). As non-OS platforms use UART communications, their user times are a lot longer than the times in all OS platforms. However, among the OS platforms, Intel Edison using WIFI connection has the worst user times than the others in this group, and the Beaglebone Black using a virtual Ethernet connection has user times about double to the other platforms using USB to Ethernet adapter and on-board adapters.

Additionally, DragonBoard 410c and Raspberry PI 3 having the most powerful processors have the lowest embedded times in this test, while XMOS having the lowest processing speed has the most embedded time in this test. Therefore, the types of communications between the RA server and the RA client affect the user times, whereas the processor specifications affect the embedded times.

6.3 Threats to Validity

In this section, we discuss the trade-off analysis and possible improvements of XEUnit, a cross-platform unit testing for real-time embedded systems.

6.3.1 Internal Threats to Validity

For the internal threats to validity, the case study addresses two aspects of the heterogeneity in embedded platforms: hardware interfaces and loading methods. Additionally, the hardware interfaces used by the non-OS platforms are not different, as they use the UART interface in this experiment. There are also other types of hardware interfaces for non-OS platforms (e.g., I2C, SPI, and other serial communications) that can be used in the evaluation. In addition to the types of interfaces, the transmission rates of the communications should also be varied. For example, we can use different baud rates for the UART communication for the non-OS platforms and different protocols of network communications (e.g., TCP and UDP protocols) for the OS-platforms. In extending this work, we may include more types of interfaces, protocols, and transmission rates, to observe boarder transmission times in the evaluation.

6.3.2 External Threats to Validity

There are some external threats to validity in this case study. First, the primary goal of this case study is to demonstrate the effectiveness of XEUnit that enables testers to use a single unit testing framework for testing code across different embedded platforms, so our case study evaluates the black-box test cases across seven different embedded platforms. However, the number of embedded platforms used in this evaluation may not be varied enough for convincing some developers that this framework can test their code on any embedded platforms. To mitigate this threat, we can add more different embedded platforms to the XEUnit as our future work especially for non-OS embedded platforms that have various hardware interfaces and use particular loading methods.

In addition to the embedded platforms, we only evaluate the test cases on a single software project. Although we use four different software modules from real-world embedded project (i.e., Cleanflight), this cases study is still limited to the code of one project. Thus, the generalizability of the results is only for these embedded modules in the same project. To provide this, we could test XEUnit with code from other application areas, such as automotive, agricultural, medical, and aerospace applications.

6.4 Discussion

We analyze the trade-offs of XEUnit compared to recent unit testing frameworks. We also discuss possible approaches to improve the quality of XEUnit.

6.4.1 Trade-off Analysis

XEUnit has several advantages compared to recent unit testing frameworks, most of which are based on xUnit Frameworks.

6.4.1.1 More Efficient Iterative Instrumentation

Since cluster iterative instrumentation can reduce the number of nodes from the original graph, this concept is more efficient than the iterative instrumentation technique proposed in [36]. Our first case study indicates that the cluster version can reduce the total execution time up to 36% compared to the original version.

6.4.1.2 Cross-platform Embedded Testing

XEUnit enables testers to use a single test framework for testing code on different embedded platforms. With the runtime adapter, XEUnit can load executable code to run on a wide variety of embedded platforms, and it can flexibly add new runtime adapters for new embedded platforms as each adapter is written in a separate Python script. Not only the embedded platforms, the XEUnit itself is also cross-platform as it is written with Python, a scripting language supported by popular operating systems.

6.4.1.3 Minimal Execution Code

XEUnit only includes a small RA client on the testing unit. From our case study, the RA client increases the size of the original code about 13.25%, while the Google Test increases the size of the original code about 98.80%. This allows the RA client to function for any types of embedded platforms especially the small memory footprint ones.

6.4.1.4 Flexible Test Cases

Unlike most unit testing tools, the test data are stored externally on the XEUnit framework rather than the test unit. Not only the size of the test unit is small, but also testers can flexibly change the test data without recompiling the test unit.

6.4.1.5 Support for Time-sensitive Testing

XEUnit can evaluate code coverage for time-sensitive applications without additional runtime overhead by using cluster iterative instrumentation. However, for testing normal applications, testers can select traditional code coverage, which is less expensive than the iterative version.

6.4.1.6 Provide Precise Execution Time

XEUnit can precisely measure the execution time of the test unit with a local timer built in the RA client. In addition to the local timer, the iterative instrumentation technique can also measure precise timing between the first node and any node in the structure replaced by a special exit statement (i.e., a variant) at that location.

Despite having many benefits, the XEUnit still has drawbacks largely due to the scalability of iterative instrumentation. Since iterative instrumentation executes each test with multiple variants, the total execution cost is more expensive than traditional instrumentation. However, these issues are with the iterative instrumentation technique, which is an optional part of XEUnit. XEUnit allows testers to use either traditional code coverage technique (instrumentation) for normal software or iterative instrumentation for time-sensitive applications.

6.4.2 Possible Improvements

Although we introduce approaches for solving the performance and scalability issues of the iterative instrumentation technique by using cluster iterative instrumentation and parallel computing nodes, the cluster iterative instrumentation technique used in the current version still has the time complexity of $O(n^2)$. To improve the efficiency of the our clustering algorithm, we may redesign the algorithm using sophisticated approaches like Lengauer and Tarjan fast algorithm [75] having the complexity of $O(n \log n)$ in our next version.

Additionally, our parallel unit testing framework only executes one variant on each embedded platform, although the embedded platform can support multiprocessing or multithreading with multicore processors. To maximize the resource utilization, we could provide an option for users to execute one or more variants on each platform if needed. However, this will make the variant tracker more complicated, as it has to ensure each variant has enough resources to execute.

CHAPTER 7: CONCLUSION

This chapter summarizes all contributions made across the five major aspects of this research: the evaluation of the impact of runtime overhead from traditional instrumentation techniques, iterative instrumentation, cluster iterative instrumentation, parallel iterative instrumentation, and XEUnit. Additionally, we also provide our thoughts in extending our contributions as possible future work.

7.1 Conclusion

In software testing, code coverage testing frameworks use traditional instrumentation techniques that insert additional instructions into the original code resulting in runtime overhead. In most software systems, the runtime overhead can be ignored, but in real-time systems it can lead to incorrect results or even cause system failures during testing. We, therefore, characterize the impact of runtime overhead by comparing the test results of different types of code coverage criteria (i.e., node, edge, and logic coverage) in time-sensitive applications (i.e., heuristic pathfinders) with the time constraints. The results indicate that runtime overhead can alter the testing paths resulting in sub-optimal solutions. In particular, with more complicated criteria (i.e., logic coverage), the results could be worse as the A* algorithm cannot obtain any solution within the given deadlines.

To solve the issue of runtime overhead from traditional instrumentation techniques, we introduce “iterative instrumentation,” a code coverage technique without runtime overhead, by adapting the concept of weak mutation. Since our technique only inserts a single instruction (i.e., an exit statement) at a unique location, the execution time from the start node to the instrumented node is identical to the original code. The results of this evaluation confirm that iterative instrumentation can effectively measure code coverage without additional runtime overhead. Although the iterative instrumentation technique can tackle the runtime overhead problem, its major drawback is the expensive execution time. Additionally, the trend of the execution times indicates that the iterative instrumentation technique has performance and scalability issues when the number of variants increases.

Due to the disadvantages of iterative instrumentation discussed above, we can improve its efficiency with two approaches: reduce the number of variants and reduce the total execution time. The first method can be achieved by using node reduction techniques, while the other one can be accomplished by executing multiple variants simultaneously. Thus, we propose the concept of graph clustering by grouping nodes having similar properties into the same group. Our graph-clustering algorithm classifies nodes into two types: common nodes and branched nodes. The common node is similar to the dominator tree concept, as we group nodes always reached by the same test paths into the same group. In contrast, the branched nodes are nodes derived from the same parent node and never reached by the same test case. We also evaluate the efficiency of cluster iterative instrumentation by comparing the execution times between this technique and the iterative instrumentation technique. The results in this evaluation indicate that cluster iterative instrumentation is more efficient than iterative instrumentation, since the number of nodes in the CFG can be reduced up to 49% and the testing time is reduced up to 40%.

In addition to reducing the number of variants, we also improve the scalability to our unit testing framework by using parallel computing. Since all variant executions are independent and parallelizable, we can execute them at the same time on a cluster of embedded platforms. We evaluate the scalability of our parallel unit testing framework by comparing the execution times of both iterative instrumentation and cluster iterative instrumentation on different numbers of parallel nodes (i.e., 1, 2, 4, 8, and 16 nodes). The results in the evaluation suggest that using parallel execution can improve the efficiency of the iterative instrumentation technique, and it can provide the scalability to the unit testing framework particularly when the number of tests increases.

The last contribution in this paper is XEUnit, a cross-platform unit testing framework for real-time embedded systems, as it enables testers to use a single tool for testing the same code across different embedded platforms. This is the most significant challenge in this paper, as not only the code can be tested across different platforms but also the framework itself can run on most operating systems (e.g., Linux, Mac OS X, and Windows). To achieve our goals, we use several techniques in this framework as follows.

First, we separate the framework into two sub-systems: a host unit and a test unit. This allows XEUnit to test code on memory-constrained platforms, as the RA client on the test unit is small. Second, the host unit is written in Python, a scripting language supported by many popular operating systems, so users can run the host unit on any operating system without any modification. Third, the runtime protocol enables XEUnit to interact with any embedded platforms, as the RA client and RA server of a particular platform can communicate to each other through this protocol. Fourth, with cluster iterative instrumentation, XEUnit can support time-sensitive testing for measuring code coverage on real-time embedded platforms. However, for testing normal applications, users can select traditional instrumentation, because it is less expensive.

Additionally, the embedded timer in the RA client can precisely measure the execution time from the beginning to the variant location inserted in the code. Lastly, the external test cases provide the flexibility to XEUnit, since testers can change the test cases without recompiling the test unit. We demonstrate the effectiveness of the XEUnit by testing four black-box test cases across seven embedded platforms successfully.

7.2 Future Work

There are many aspects can be extended from our contributions in this dissertation. The straightforward point is to implement the framework to support more types of coverage criteria especially modified condition/decision coverage (MC/DC) that is required by critical safety applications. This coverage type could provide high quality to most critical software, e.g., flight and landing control systems of an aircraft.

Next, the graph-clustering algorithm could be improved by using a more efficient algorithm. Although the current algorithm of our graph clustering is easy to understand, the time complexity of this version is $O(n^2)$. One possible efficient solution is to use depth first search for traveling the tree along a test path, which will have the time complexity of $O(n \log n)$. However, this is more complicated, and it needs to comply with our rule that only one branched node is reached by each test.

Moreover, the scheduling algorithm implemented should be redesigned and implemented properly to utilize all available parallel platforms during the test. Our current implementation allows only variants of the same test case to execute at the same time. The results in our study also confirm this analysis, since the improvement from parallel execution is less than our expectation especially when the number of tests is increased. To improve this, the scheduling algorithm must allow variants to execute under any test case; this will cause the design and implementation of the framework more complicated, but should result in a significant performance increase.

Lastly, the variant tracker could utilize more resources (e.g., processing units and memory) on each embedded platform for the parallel embedded testing. If the resources on the platform can be shared and are sufficiently large to handle multiple variants, we can enhance the variant tracker to allow two or more variants to execute on the same embedded platform. Nevertheless, this could be an option by allowing users to choose their preferred method, as some applications are necessary to run alone on the embedded platform (e.g., interacting with hardware components on the board).

BIBLIOGRAPHY

[1] G. J. Myers, C. Sandler, and T. Badgett, “The art of software testing,” *John Wiley & Sons*, 2011.

[2] Q. Yang, J. J. Li, and D. M. Weiss, “A survey of coverage-based testing tools,” The Computer Journal, vol. 52, no. 5, pp.589-597, 2009.

[3] P. Hamill, “Unit Test Frameworks: Tools for High-Quality Software Development,” O’Reilly Media, Inc., 2004.

[4] G. Meszaros, “xUnit test patterns: Refactoring test code,” Pearson Education, 2007.

[5] C. Hujer, “AceUnit,” Retrieved March 1, 2017, from: https://github.com/christianhujer/aceunit.git.

[6] B. Donahue, “Google Test Google C++ test framework,” Retrieved March 1, 2017, from: https://github.com/google/googletest.git.

[7] E. Sommerlade et. al, “CppUnit,” Retrieved March 1, 2017, from: http://cppunit.sourceforge.net/doc/cvs/index.html.

[8] A. Malect et. al, “Check,” Retrieved March 1, 2017, from: http://libcheck.github.io/check/doc/check_html/index.html.

[9] A. Kumar, and J. St.Clair, “CUnit,” Retrieved March 1, 2017, from: http://cunit.sourceforge.net/doc/index.html.

[10] T. Punkka, “embUnit,” Retrieved March 1, 2017, from: http://embunit.sourceforge.net/embunit/index.html.

[11] P. Runeson, “A survey of unit testing practices, Software,” IEEE, vol. 23, no. 4, pp.22-29, 2006.

[12] J. Bishop, and N. Horspool, “Cross-platform development: Software that lasts,” Computer, vol. 39, no. 10, 26-35, 2006.

[13] J. King, and M. Easton, “Cross-platform .NET Development: Using Mono, Portable .NET, and Microsoft .NET,” Apress, 2004.

[14] N. Umesh, and A. Saraswat, “Automation Testing: An Introduction to Selenium,” International Journal of Computer Applications, vol. 119, no. 3, 2015.

[15] H. Kopetz, “Real-time systems: design principles for distributed embedded applications,” Springer Science & Business Media, 2011.

[16] T. D. Morton, “Embedded Microcontrollers,” Prentice Hall PTR, 2000.

[17] M. Schlett, “Trends in embedded-microprocessor design,” Computer, vol. 31, no. 8, pp.44-49, 1998.

[18] R. Toulson, and T. Wilmshurst, “Fast and effective embedded systems design: applying the ARM mbed,” Elsevier, 2012.

[19] EVK1100, “A.T.M.E.L., AVR32 Studio software documentation: www.atmel.com/tools,” EVK1100.aspx.

[20] D. Watt, “Programming XC on XMOS devices,” XMOS Limited, 2009.

[21] MPLAB IDE, “Simulator, Editor User’s Guide,” Manual DS51025D, Microchip, 2001.

[22] C. Churchill, “Cloud 9,” Psychology Press, 1984.

[23] L. D. Jasio, “Programming 32-bit Microcontrollers in C: Exploring the PIC32,” Newnes, 2011.

[24] E. Upton, and G. Halfacree, “Raspberry Pi user guide,” John Wiley & Sons, 2014.

[25] G. Coley, “Beaglebone black system reference manual,” Texas Instruments, Dallas, 2013.

[26] Boards, Intel Edison, “Intel Edison Board Support Package-User Guide,” 2014.

[27] Qualcomm Technologies, Inc., “DragonBoard 410c Linux User Guide,” Retrieved March 1, 2017, from: http://www.github.com/96boards/documentation/blob/master/ConsumerEdition/DragonBoard-410c/Guides/LinuxUserGuide_DragonBoard.pdf.

[28] M. M. Tikir, and J. K. Hollingsworth, “Efficient instrumentation for code coverage testing,” In ACM SIGSOFT Software Engineering Notes, vol. 27, no. 4, pp. 86-96, 2002.

[29] R. Fryer, “FPGA based CPU instrumentation for hard real-time embedded system testing,” ACM SIGBED Review, vol. 2, no. 2, pp.39-42, 2005.

[30] S. Fischmeister, and P. Lam, “Time-aware instrumentation of embedded software, Industrial Informatics,” IEEE Transactions on, vol. 6, no. 4, pp.652-663, 2010.

[31] J. Hawkins, R. B. Howard, and H. V. Nguyen, “Automated real-time testing (ARTT) for embedded control systems (ECS),” In Proceedings of IEEE Reliability and Maintainability Symposium, pp.647-652, 2002.

[32] M. S. AbouTrab, M. Brockway, S. Counsell, and R. M. Hierons, “Testing real-time embedded systems using timed automata based approaches,” Journal of Systems and Software, vol. 86, no. 5, pp.1209-1223, 2013.

[33] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, “Testing real-time embedded software using UPPAAL-TRON: an industrial case study,” In Proceedings of the 5th ACM international conference on Embedded software, pp.299-306, 2005.

[34] F. Mattiello-Francisco, E. Martins, A. R. Cavalli, and E. T. Yano, “InRob: An approach for testing interoperability and robustness of real-time embedded software,” Journal of Systems and Software, vol. 85, no. 1, pp.3-15, 2012.

[35] J. Boydens, P. Cordemans, and E. Steegmans, “Test-driven development of embedded software,” In Proceedings of the Fourth European Conference on the Use of Modern Information and Communication Technologies, pp.117-128, 2010.

[36] T. Pankumhang, and M. Rutherford, “Iterative Instrumentation for Code Coverage in Time-Sensitive Systems,” In Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on, pp.1-10, 2015.

[37] W. E. Howden, “Weak mutation testing and completeness of test sets,” Software Engineering, IEEE Transactions on 4, pp. 371-379, 1982.

[38] M. R. Girgis, and M. R. Woodward. “An integrated system for program testing using weak mutation and data flow analysis,” In Proceedings of the 8th international conference on Software engineering, pp. 313-319, 1985.

[39] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” Computer, vol. 11, no. 4, pp.34-41, 1978.

[40] P. R. Mateo, and M. P. Usaola, “Parallel mutation testing,” Software Testing, Verification and Reliability, vol. 23, no. 4, pp.315-350, 2013.

[41] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” Systems Science and Cybernetics, IEEE Transactions on 4, no. 2, pp. 100-107, 1968.

[42] I. Pohl, “Heuristic search viewed as path finding in a graph,” Artificial intelligence, vol. 1, no. 3, pp. 193-204, 1970.

[43] E. A. Hansen, and R. Zhou, “Anytime Heuristic Search,” J. Artif. Intell. Res.(JAIR), vol. 28, pp. 267-297, 2007.

[44] N. Kumar, B. R. Childers, and M. L. Soffa, “Low overhead program monitoring and profiling,” In ACM SIGSOFT Software Engineering Notes, vol. 31, no. 1, pp. 28-34, 2005.

[45] R. Santelices, and M. J. Harrold, “Efficiently monitoring dataflow test coverage,” In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 343-352, 2007.

[46] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa, “Demand-driven structural testing with dynamic instrumentation,” In Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, pp. 156-165, 2005.

[47] K. R. Chilakamarri, and S. Elbaum, “Reducing coverage collection overhead with disposable instrumentation,” In Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on, pp. 233-244, 2004.

[48] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, “PEBIL: Efficient static binary instrumentation for linux,” In Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium, pp. 175-183, 2010.

[49] J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, “An experimental determination of sufficient mutant operators,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 5, no. 2, pp. 99-118, 1996.

[50] W. E. Wong, and A. P. Mathur, “Reducing the cost of mutation testing: An empirical study,” Journal of Systems and Software 31, no. 3, pp. 185-196, 1995.

[51] H. Zhu, P. A. Hall, and J. H. May, “Software unit test coverage and adequacy,” ACM Computing Surveys (CSUR), vol. 29, no. 4, pp. 366-427, 1997.

[52] A. Paul, and J. Offutt, “Logic Coverage,” in Introduction to software testing, New York, NY: Cambridge Univ. Press, ch. 3, pp. 104-149, 2008.

[53] S. C. Ntafos, “A comparison of some structural testing strategies,” IEEE Transactions on software engineering, vol. 14, no. 6, pp. 868-874, 1988.

[54] A. T. Acree, “On Mutation,” Ph.D. thesis, Georgia Inst. of Technology, 1980.

[55] W. E. Wong, “On Mutation and Data Flow,” Ph.D. thesis, Purdue Univ., 1993.

[56] Y. Jia, and M. Harman, “Higher order mutation testing,” Information and Software Technology, vol. 51, no. 10, pp. 1379-1393, 2009.

[57] M. Polo, M. Piattini, and I. García-Rodríguez, “Decreasing the cost of mutation testing with second-order mutants,” Software Testing, Verification and Reliability, vol. 19, no. 2, pp. 111-131, 2009.

[58] E. W. Krauser, A. P. Mathur, and V. J. Rego, “High performance software testing on SIMD machines,” Software Engineering, IEEE Transactions on, vol. 17, no. 5, pp. 403-423, 1991.

[59] J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, “Mutation testing of software using a MIMD computer,” In in 1992 International Conference on Parallel Processing, 1992.

[60] D. Clifton et al., “Cleanflight,” Retrieved March 1, 2017, from: https://github.com/cleanflight/cleanflight/tree/master/docs.

[61] S. E. Schaeffer, “Graph clustering,” Computer science review, vol. 1, no. 1, pp.27-64, 2007.

[62] G. V. Rossum, and F. L. Drake, “The python language reference manual,” Network Theory Ltd., 2011.

[63] D. Crockford, “The application/json media type for javascript object notation (json),” 2006.

[64] T. Bray et al., “Extensible markup language (XML) 1.0,” 2008.

[65] S. Skogestad, and I. Postlethwaite, “Multivariable feedback control: analysis and design,” vol. 2, New York: Wiley, 2007.

[66] D. D. Gajski et al, “Specification and design of embedded systems,” vol. 13, Englewood Cliffs: Prentice Hall, 1994.

[67] V. Kumar, A. Grama, A. Gupta, and G. Karypis, “Introduction to parallel computing: design and analysis of algorithms,” vol. 400, Redwood City, CA: Benjamin/Cummings, 1994.

[68] B. Choi, A. Mathur, and B. Pattison, “PMothra: Scheduling mutants for execution on a hypercube,” In ACM SIGSOFT Software Engineering Notes, vol. 14, No. 8, pp. 58-65, 1989.

[69] A. Paul, and J. Offutt, “Logic Coverage,” in Introduction to software testing, New York, NY: Cambridge Univ. Press, ch. 3, pp. 104-149, 2008.

[70] J. J. Chilensky, and S. P. Miller, “Applicability of modified condition/decision coverage to software testing,” Software Engineering Journal, 1994.

[71] A. Dupuy, and N. Leveson, “An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software,” In Digital Avionics Systems Conference, In Proceedings of the 19th IEEE DASC, vol. 1, pp. 1B6-1, 2000.

[72] R. T. Prosser, “Applications of boolean matrices to the analysis of flow diagrams,” In Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference, pp. 133-138, 1959.

[73] E. S. Lowry, and C. W. Medlock, “Object code optimization,” Communications of the ACM, vol. 12, no. 1, pp. 13-22, 1969.

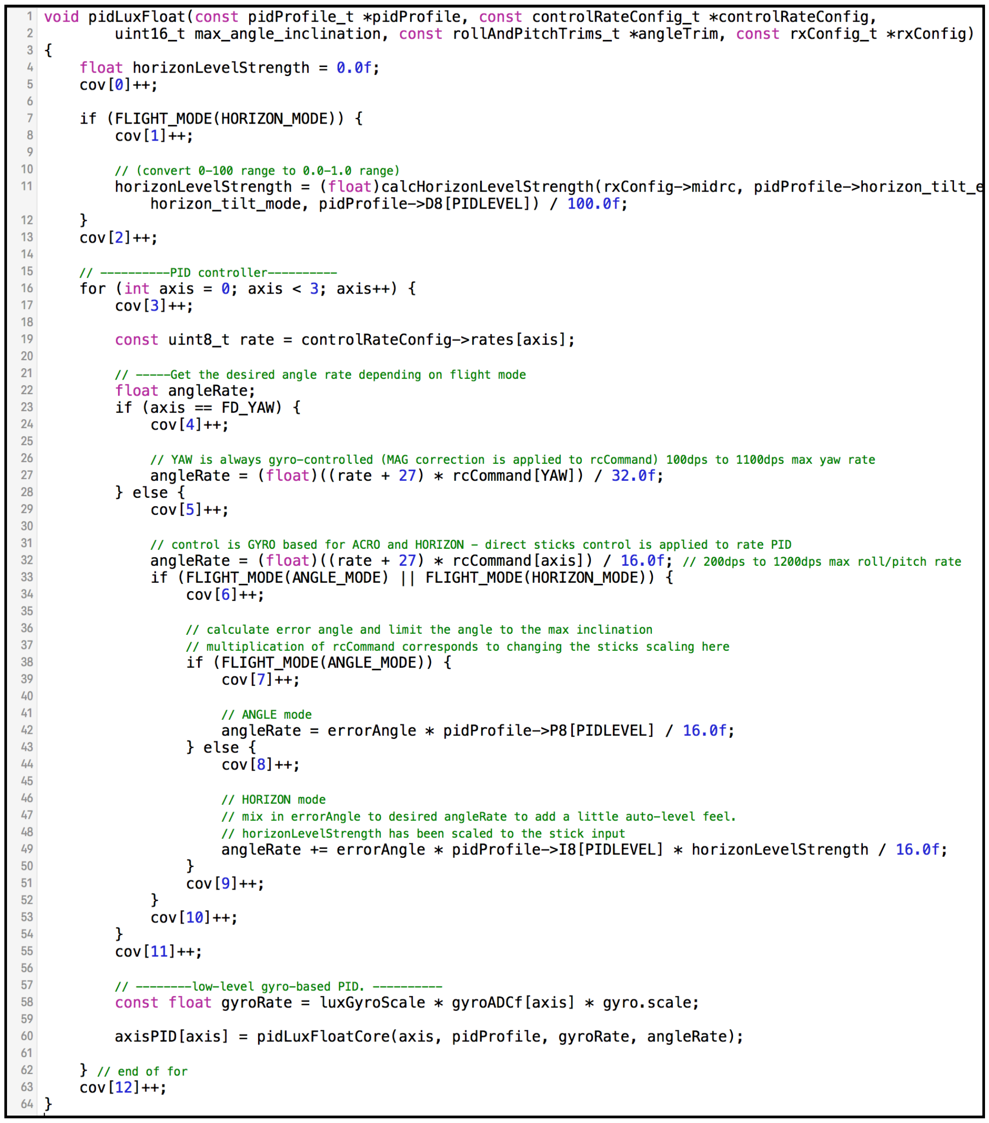
[74] P. W. Purdom Jr, and E. F. Moore, “Immediate predominators in a directed graph [H]. Communications of the ACM,” vol. 15, no. 8, pp. 777-778, 1972.

[75] T. Lengauer, and R. E. Tarjan, “A fast algorithm for finding dominators in a flowgraph,” ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 1, no. 1, pp. 121-141, 1979.

APPENDICES

Appendix A: Graph Clustering Implementation

From our graph clustering rules and algorithm in Chapter 4, we provide an example how can we transform a CFG into a list of cluster nodes (i.e. variants) as a Python script. In this example, we only transform the main function used in Case Study 3 called pidLuxFloat() function.



```c
void pidLuxFloat(const pidProfile_t *pidProfile, const controlRateConfig_t *controlRateConfig,
        uint16_t max_angle_inclination, const rollAndPitchTrims_t *angleTrim, const rxConfig_t *rxConfig)
{
    float horizonLevelStrength = 0.0f;
    cov[0]++;

    if (FLIGHT_MODE(HORIZON_MODE)) {
        cov[1]++;

        // (convert 0-100 range to 0.0-1.0 range)
        horizonLevelStrength = (float)calcHorizonLevelStrength(rxConfig->midrc, pidProfile->horizon_tilt_e
            horizon_tilt_mode, pidProfile->D8[PIDLEVEL]) / 100.0f;
    }
    cov[2]++;

    // ----------PID controller----------
    for (int axis = 0; axis < 3; axis++) {
        cov[3]++;

        const uint8_t rate = controlRateConfig->rates[axis];

        // -----Get the desired angle rate depending on flight mode
        float angleRate;
        if (axis == FD_YAW) {
            cov[4]++;

            // YAW is always gyro-controlled (MAG correction is applied to rcCommand) 100dps to 1100dps max yaw rate
            angleRate = (float)((rate + 27) * rcCommand[YAW]) / 32.0f;
        } else {
            cov[5]++;

            // control is GYRO based for ACRO and HORIZON - direct sticks control is applied to rate PID
            angleRate = (float)((rate + 27) * rcCommand[axis]) / 16.0f; // 200dps to 1200dps max roll/pitch rate
            if (FLIGHT_MODE(ANGLE_MODE) || FLIGHT_MODE(HORIZON_MODE)) {
                cov[6]++;

                // calculate error angle and limit the angle to the max inclination
                // multiplication of rcCommand corresponds to changing the sticks scaling here
                if (FLIGHT_MODE(ANGLE_MODE)) {
                    cov[7]++;

                    // ANGLE mode
                    angleRate = errorAngle * pidProfile->P8[PIDLEVEL] / 16.0f;
                } else {
                    cov[8]++;

                    // HORIZON mode
                    // mix in errorAngle to desired angleRate to add a little auto-level feel.
                    // horizonLevelStrength has been scaled to the stick input
                    angleRate += errorAngle * pidProfile->I8[PIDLEVEL] * horizonLevelStrength / 16.0f;
                }
                cov[9]++;
            }
            cov[10]++;
        }
        cov[11]++;

        // --------low-level gyro-based PID. ----------
        const float gyroRate = luxGyroScale * gyroADCf[axis] * gyro.scale;

        axisPID[axis] = pidLuxFloatCore(axis, pidProfile, gyroRate, angleRate);

    } // end of for
    cov[12]++;
}
```

Figure 24. Node coverage instrumentation of the pidLuxFloat() function

Figure 24 is the instrumented code for node coverage of the pidLuxFloat() function, which is the main function of a PID controller of the cleanflight project. In the figure, there are thirteen node coverage points (i.e., node 0 to node 12) inserted in the original code. The structure of the function can be represented as a CFG as shown in Figure 24.

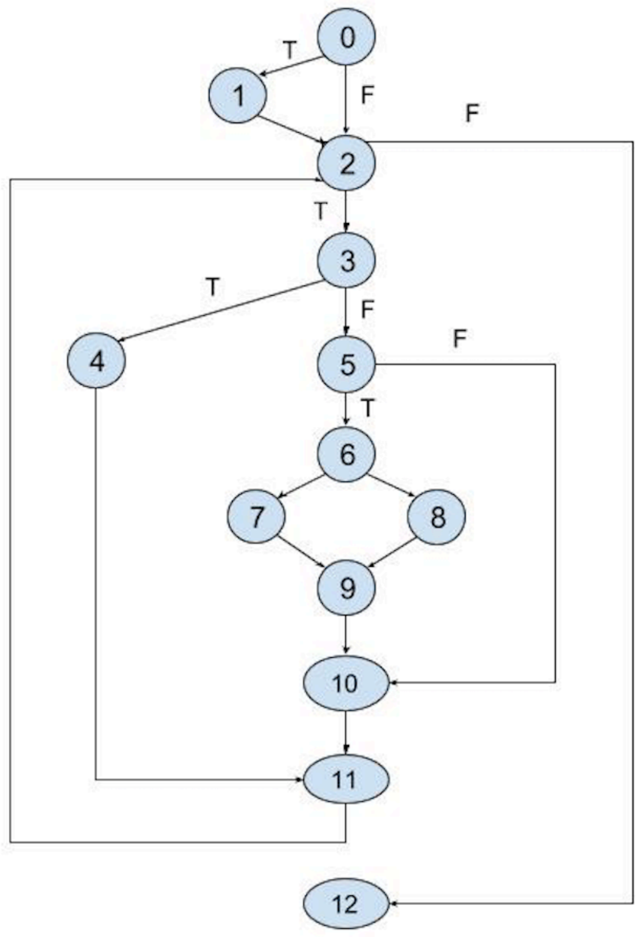


Figure 25. A control flow graph of the pidLuxFloat() function

Figure 25 is a CFG of the pidLuxFloat() function transformed from the C code above. In this CFG, there is a loop between node2 to 11, so the number of occurrences of nodes 2 to 11 can be greater than the other nodes outside the loop in the test path. From this CFG, we generate a set of test paths as follows.

tp0 = {0,1,2,3,4,11,2,12}
tp1 = {0,1,2,3,5,6,7,9,10,11,2,12}
tp2 = {0,1,2,3,5,6,8,9,10,11,2,12}
tp3 = {0,1,2,3,5,10,11,2,12}
tp4 = {0,2,3,4,11,2,12}
tp5 = {0,2,3,5,6,7,9,10,11,2,12}
tp6 = {0,2,3,5,6,8,9,10,11,2,12}
tp7 = {0,2,3,5,10,11,2,12}

From the above test paths, we implement a Python script to transform the given test paths for a CFG into a list of cluster nodes as follows.

```
# Initialize node occurrence and parent lists

nc = []

parents = []

for i in range (0, NUM_NODES):

   nc.append(0)

   parents.append(-1)


# Count the occurrence for each node and link to its parents

for p in range (0,len(tp)):

   parent = -1

   for n in range (0, len(tp[p])):

      node = tp[p][n]

      nc[node] += 1

      parents[node] = parent

      parent = node
```

```
# Create common node and branched node lists

cn_list = []

bn_list = []

for node_no in range (0, NUM_NODES):

   if (nc[node_no] >= len(tp)):

      cn_list.append(node_no)

   else:

      parent = parents[node_no]

      bn_list.append(node_no)


# Initialize variant list (cluster nodes)

var_list = []

var_list.append(cn_list)


# Combine common node and branched nodes into the variant list

tmp_list = []

prev_parent = -1

for i in range (0, len(bn_list)):

   parent = parents[bn_list[i]]

   if (parent == prev_parent):

      tmp_list.append(bn_list[i])

   else:

      if tmp_list:

         var_list.append(tmp_list)

      tmp_list = []

      tmp_list.append(bn_list[i])

   prev_parent = parent
```

```
# add the last branched node list to the variant list
if (len(tmp_list) > 0):
   var_list.append(tmp_list)
```

Appendix B: XEUnit Implementation

```
xeunit
├── configs
│   ├── arm_config.json
│   ├── avr_config.json
│   ├── bbb_config.json
│   ├── dragon_config.json
│   ├── edison_config.json
│   ├── pic32_config.json
│   ├── rpi_config.json
│   ├── xeunit_config.json
│   └── xmos_config.json
├── platforms
│   ├── arm.py
│   ├── avr.py
│   ├── bbb.py
│   ├── dragon.py
│   ├── edison.py
│   ├── pic32.py
│   ├── rpi.py
│   └── xmos.py
├── reports
│   ├── arm_test_pid_nc_report.txt
│   ├── avr_test_pid_nc_report.txt
│   ├── bbb_test_pid_nc_report.txt
│   ├── dragon_test_pid_nc_report.txt
│   ├── edison_test_pid_nc_report.txt
│   ├── pic32_test_pid_nc_report.txt
│   ├── rpi_test_pid_nc_report.txt
│   └── xmos_test_pid_nc_report.txt
├── scripts
│   ├── evaluator.py
│   ├── executor.py
│   ├── gen_mutant.sh
│   ├── gentestcases.py
│   ├── pxeunit.py
│   ├── report.py
│   └── utils.py
├── testcases
│   └── test_pid_testcase.json
└── variants
    ├── arm
    │   ├── test_pid-n0.bin
    │   ├── test_pid-n1.bin
    │   └── test_pid-n2.bin
    ├── avr
    │   ├── test_pid-n0.elf
    │   ├── test_pid-n1.elf
    │   └── test_pid-n2.elf
    ├── bbb
    │   ├── test_pid-n0
    │   ├── test_pid-n1
    │   └── test_pid-n2
    ├── dragon
    │   ├── test_pid-n0
    │   ├── test_pid-n1
    │   └── test_pid-n2
    ├── edison
    │   ├── test_pid-n0
    │   ├── test_pid-n1
    │   └── test_pid-n2
    ├── pic32
    │   ├── test_pid-n0.hex
    │   ├── test_pid-n1.hex
    │   └── test_pid-n2.hex
    ├── rpi
    │   ├── test_pid-n0
    │   ├── test_pid-n1
    │   └── test_pid-n2
    └── xmos
        ├── test_pid-n0.xe
        ├── test_pid-n1.xe
        └── test_pid-n2.xe
```

The previous tree structure is the XEUnit implementation for a PID controller. Under the root directory (i.e., xeunit), there are six directories under the XEUnit root (xeunit): configs, platforms, reports, scripts, testcases, and variants.

In this example, the configs directory stores the configurations of XEunit and all embedded platforms in JSON format. Under the platforms directory, there are eight different embedded platforms: ARM mbed (arm.py), Atmel AVR (avr.py), Beaglebone Black (bbb.py), DragonBoard 410c (dragon.py), Intel Edison (edison.py), Microchip PIC32 (pic32.py), Raspberry Pi (rpi.py), and XMOS XK-1A (xmos.py), written in Python. The reports directory stores the test reports for the current evaluation (i.e., node coverage for a PID controller). The scripts directory contains Python scripts of the XEUnit framework. The test cases file (i.e., test_pid_testcases.json) is stored as a JSON format under the testcases directory. Lastly, the variants directory stores the variants for loading and running on specific platforms. Please note that the number of variants must be the same as the number of cluster nodes for the test. Due to the limitation, however, we only show three variants for each platform (e.g., test_pid-n0, test_pid-n1, and test_pid-n2).

There are two important modules in the XEUnit framework: xeunit.py and executor.py, which are shown here.

xeunit.py

```
from report import Report
from executor import Executor
from utils import *
import timer

# constants
CONFIG_PATH   = "../configs/"
XEUNIT_CONFIG = "xeunit_config.json"
```

```
#=================== main() function =====================
def main():
    # parse command line arguments
    arg = Arg()
    arg.parse()

    # load XEUNIT and platform configurations
    config = Config()
    config.load_config("xeunit",CONFIG_PATH+XEUNIT_CONFIG, arg)
    config.load_config("platform",CONFIG_PATH+arg.platform+"_config.json", arg)
    config.print_config("xeunit", arg)
    config.print_config("platform", arg)
    config.set_cov(arg.test_type)     # set prefix and related data

    # validate variant directory and create an empty kill directory
    generator = Generator()
    generator.check_varpath(arg.platform)

    # load test cases from external file for code coverage testing
    report = Report(arg, config)
    if arg.isCovTest():
        config.load_testcase(arg.platform)
        report.printTestcases()
        if arg.isIterativeTest():
            report.printWorkers()

    # -------------------- start system time
    t0 = timer.timer()

    # execute all tests
    executor = Executor(arg)
    test_results = executor.run_all_tests(arg, config)

    # -------------------- stop system time
    t1 = timer.timer()
    sys_time = t1 - t0

    # generate a test report
    report.printTestResults(arg, test_results)
    print "Test time: {:>8.2f} sec".format(sys_time)

#====================== END OF MAIN =======================

if __name__== "__main__":
    main()
```

executor.py

```python
from Queue import Queue
from threading import Thread
from threading import Lock
import json
import os
import sys
import socket
import string
import time
import timer
import utils
sys.path.insert(0, '../platforms')

UNKILLED = -1
MAX_CONNS = 16
MAX_NODES = MAX_CONNS
SERVER_IP = "192.168.7.1"
CLUSTER_CONFIG = "../configs/pid_cluster.json"

jobQ = Queue()

class Executor(object):
    def __init__(self, test_type=""):
        self.test_type = test_type
        self.tt0 = -1
        self.tt1 = -1
        self.socket = -1
        self.cov = []
        self.var_list = []
        self.num_kill = 0
        self.num_all_cov = 0
        self.num_detect_cov = 0
        self.cov_percent = 0
        self.total_embtime = 0
        self.total_testtime = 0

        # load reduced node list
        pid_cluster = utils.fread(CLUSTER_CONFIG)
        data = json.loads(pid_cluster)
        self.cn_list = list(data['cn_list'])
        self.bn_list = list(data['bn_list'])
        self.rn_list = self.cn_list + self.bn_list

    # ==================== execute a variant
    def exe_variant(self, arg, config, variant, tc_input="", w=-1):
        # non-OS platform
        if arg.platform in utils.NON_OS_PLATFORMS:
            nonos_platform = __import__(arg.platform)
            nonos_platform.exe_platform(config.platform_port,
config.baud_rate, variant)
```

```
        # OS platform
        elif arg.platform in utils.OS_PLATFORMS:
            os_platform = __import__(arg.platform)
            if w == -1:
                os_platform.exe_platform(config.user_host, variant)
            else:
                worker_name = "pi@node"+str(w+1)
                os_platform.exe_platform(worker_name, variant)

        # --------------- get test result
        # get black-box testing result
        if arg.test_type == '-b':
            ret_result = self.blackbox_test()
        # get code coverage result
        else:
            ret_result = self.cov_test(tc_input)

        return ret_result

    # run all jobs (variants) with the current test data
    # all test results will be stored in the var_list[] variable
    def run_jobs(self, arg, config, var_list, q, tc_i=-1, w=-1):

    # cluster iterative version: each variant represents a group of
nodes in the CFG, and the return output will be the node number in the
group (killed) or -1 (unkilled)
        while not q.empty():
            variant = q.get()

            ret_result = self.exe_variant(arg, config, variant['name'],
config.tc[tc_i].input, w)

            # store returned result and test time to the var_list
            var_list[variant['no']]['output'] = ret_result['result']
            var_list[variant['no']]['embtime'] = ret_result['embtime']
            var_list[variant['no']]['worker'] = (w+1)

            q.task_done()

    # ======================= run the current test case with one
(normal code coverage) or all variants (iterative code coverage)
    def run_test(self, arg, config, variant, tc_i, var_list,
test_results):

        # ----------------------------- iterative code coverage
        if arg.test_type == '-i':

            test_result = {'cov':([0]*config.num_cov), 'percent':0,
'emb_time':0, 'test_time':0}
            pre_cov = list(self.cov) # store previous coverage
            # put unkilled variants into the job queue
            tmpQ = Queue()
            for var_no in range(0,len(self.cov)):
                if self.cov[var_no] == 0:
                    jobQ.put(var_list[var_no])
                    tmpQ.put(var_no)

            # clear emb. time in variant list
            for i in range(0,len(var_list)):
                var_list[i]['embtime'] = 0

            # generate a set of threads to execute all jobs in queue
            # ------------- start test timer (tt0)
            tt0 = timer.timer()

            for w in range(0, arg.num_workers):
                worker = Thread(target=self.run_jobs, args=(arg,
config, var_list, jobQ, tc_i, w,))
                worker.start()
            jobQ.join()

            # ------------- stop test timer (tt1)
            tt1 = timer.timer()
            self.total_testtime = (tt1 - tt0)*1000

            # calculate emb. time for parallel execution
            self.total_embtime = self.parallel_embtime(var_list,
arg.num_workers)

            # update code coverage stats
            while not tmpQ.empty():
                # get next variant
                var_no = tmpQ.get()

                # update code coverage stats
                no = int(var_list[var_no]['output'])
                if no != UNKILLED:
                    self.cov[no] = 1
                    test_result['cov'][no] = '1'

            # calculate number of killed variants and generate report
            self.num_kill = 0
            for var_no in range(0,len(self.cov)):
                # caluculate the number of killed variants
                if self.cov[var_no] == 1:
                    self.num_kill += 1
                # generate code coverage report
                if pre_cov[var_no] == 1:
                    if int(test_result['cov'][var_no]) != UNKILLED:
                        test_result['cov'][var_no] = "x"
                    else:
                        test_result['cov'][var_no] = "1"

            # calculate coverage percentage
            self.cov_percent = self.num_kill*100/self.num_all_cov
            # construct a result for each test
            test_result['emb_time'] = self.total_embtime
            test_result['test_time'] = self.total_testtime

        # ----------------------------- Cluster iterative code coverage
        elif arg.test_type == '-c':
            # each test result consists of four elements: coverage 
data, coverage percentage, embedded time, and user time
            test_result = {'cov':([0]*config.num_cov), 'percent':0, 
'emb_time':0, 'test_time':0}
            pre_cov = list(self.cov) # store previous coverage

            # put unkilled variants into the job queue
            tmpQ = Queue()
            for rn_no in range(0,len(self.rn_list)):
                if self.all_node_kill(self.rn_list[rn_no]) == 0:
                    jobQ.put(var_list[rn_no])
                    tmpQ.put(rn_no)

            # clear emb. time in variant list
            for i in range(0,len(var_list)):
                var_list[i]['embtime'] = 0

            # ------------- start test timer (tt0)
            tt0 = timer.timer()

            for w in range(0, arg.num_workers):
                worker = Thread(target=self.run_jobs, args=(arg, 
config, var_list, jobQ, tc_i, w,))
                worker.start()
            jobQ.join()

            # ------------- stop test timer (tt1)
            tt1 = timer.timer()
            self.total_testtime = (tt1 - tt0)*1000

            # calculate emb. time for parallel execution
            self.total_embtime = self.parallel_embtime(var_list, 
arg.num_workers)

            # update code coverage stats
            while not tmpQ.empty():
                # get next variant
                rn_no = tmpQ.get()

                # update code coverage stats
                no = int(var_list[rn_no]['output'])
                if no != UNKILLED:
                    # check common list first
                    if self.cov[no] == 0:
                        inCommonList = 0
                        for i_cn in range(0,len(self.cn_list)):
                            if no in self.cn_list[i_cn]:
                                inCommonList = 1
            # update flags & cov of all nodes in the same common list
                                for i_no in
range(0,len(self.cn_list[i_cn])):
                                    node_no = self.cn_list[i_cn][i_no]
                                    self.cov[node_no] = 1
                                    self.num_kill += 1
            # remove the killed sub-list from the commond list
                                self.cn_list.remove(self.cn_list[i_cn])
                                break
                        #-------- end for i_cn ---------

                        # not in the common list, i.e. in branch list
                        if (not inCommonList):
                            self.cov[no] = 1
                            self.num_kill += 1
            #------------ end while not tmpQ.empty()

            # generate coverage report
            for no in range(0,len(self.cov)):
                if self.cov[no] == 1:
                    if pre_cov[no] == 1:
                        test_result['cov'][no] = 'x'
                    else:
                        test_result['cov'][no] = '1'

            # calculate coverage percentage
            self.cov_percent = self.num_kill*100/self.num_all_cov

            # construct a result for each test
            test_result['emb_time'] = self.total_embtime
            test_result['test_time'] = self.total_testtime

        # ----------------------------- normal code coverage
        elif arg.test_type == '-n':

            # each test result consists of four elements
            test_result = {'cov':[], 'percent':0, 'emb_time':0,
'test_time':0}

            # ------------- start test timer (tt0)
            self.tt0 = timer.timer()

            ret_result = self.exe_variant(arg, config, variant,
config.tc[tc_i].input)

            # ------------- stop test timer (tt1)
            self.tt1 = timer.timer()
            test_time = self.tt1 - self.tt0

            # evaluate normal code coverage
            output = ret_result['result']
            self.num_detect_cov = 0
```

```
                for ci in range(0,config.num_cov):
                    self.cov[ci] += int(output[ci])
                    if self.cov[ci] > 0:
                        self.num_detect_cov += 1
                        test_result['cov'].append(1)
                    else:
                        test_result['cov'].append(0)
                # ---------- end for ci

                # calculate coverage percentage
                self.cov_percent = self.num_detect_cov*100/self.num_all_cov

                # construct a result for each test
                test_result['emb_time'] = int(ret_result['embtime'])
                test_result['test_time'] = float(test_time)

            # update coverage percentage in the test result
            test_result['percent'] = self.cov_percent
            test_results.append(test_result)

        # run all tests for a particular test type
        def run_all_tests(self, arg, config):

            # start test time
            self.tt0 = timer.timer()

            # create a socket and listen on the given port
            if (arg.platform in utils.OS_PLATFORMS):
                self.network_connection(SERVER_IP,
int(config.network_port))

            # init test_results list and get a variant name
            if (arg.test_type != '-b'):
                test_results = []
                variant = config.prog_name

            # ================== Black-box Testing ==============
            if arg.test_type == '-b':

                # ------------- start test timer (tt0)
                self.tt0 = timer.timer()

                test_results = self.exe_variant(arg, config,
config.prog_name)

                # ------------- stop test timer (tt1)
                self.tt1 = timer.timer()
                test_time = self.tt1 - self.tt0

                test_results['test_time'] = test_time*1000 # msec

            # ================== Code Coverage Testing ==========
            elif arg.test_type == '-n' or arg.test_type == '-c':
```

```
            self.cov = [0] * config.num_cov # clear all node coverage
            self.cov_percent = self.num_kill = num_tests = 0
            self.num_all_cov = config.num_cov

            # initialize a list of unkilled variants
            if arg.test_type == '-n':
                num_variants = config.nc_variants
            elif arg.test_type == '-c':
                num_variants = len(self.rn_list)

            if arg.isCovTest():
                var_list = []
                for var_no in range(0,num_variants):
                    var_name = config.prog_name + "-n" + str(var_no) +
config.extension
                    var_obj = {'no':var_no, 'name':var_name,
'worker':0, 'output':UNKILLED, 'embtime':0, 'testtime':0}
                    var_list.append(var_obj)

            # executing each test with all nodes
            while (self.cov_percent < config.threshold and num_tests <
len(config.tc)):

# run the current test with all variants (normal code coverage only has
# one variant/program). the test result will be appended to the
test_results[] list (last argument)
                self.run_test(arg, config, variant, num_tests,
var_list, test_results)

                # next test case
                num_tests += 1

        # close the server socket
        if (arg.platform in utils.OS_PLATFORMS):
            self.socket.close()

        return test_results
    # ========== end of run_all_tests() function ===========

    # evaluate the black-box testing results
    def blackbox_test(self):

        # accept a client connection
        client_socket, address = self.socket.accept()

        # start timer
        self.t0 = timer.timer()

        # get the test case output
        ret_result = client_socket.recv(512)

        # stop timer and calculate execution time (user time)
        self.t1 = timer.timer()
        user_time = self.t1 - self.t0

        # TODO: we will move this section to the Evaluator object later
        # ------------------ evaluate the test results
        end_test = False
        num_pass = num_fail = 0
        js = json.loads(ret_result)

        test_results = {"testcase": js["testcase"], "num_pass":0,
"num_fail":0, "results":[], "user_time":user_time*1000, "test_time":0}
        js_results = js["results"]

        for i in range(0,len(js_results)):
            if (js_results[i]['result'] == 1): num_pass += 1
            else: num_fail += 1
            test_result = {"test_name":str(js_results[i]["test"]),
"result":int(js_results[i]["result"]),
"emb_time":long(js_results[i]["time"])}
            test_results['results'].append(test_result)

        # ------- end for i ---------
        test_results['num_pass'] = num_pass
        test_results['num_fail'] = num_fail

        # evaluate the test case result (passed or failed)
        if (len(js_results) == num_pass):
            test_results['testcase_result'] = 1
        else:
            test_results['testcase_result'] = 0

        return test_results

    # code coverage test
    def cov_test(self, testdata):

        # accept a client connection
        client_socket, address = self.socket.accept()

        # start user timer (t0)
        self.t0 = timer.timer()

        # send test data in JSON format:
        # { "no": <test_no>, "input": <a list of input parameters> }
        client_socket.send(testdata)
        # receive test result
        ret_result = client_socket.recv(512)

        # stop user timer (t1) and calculate transmission time
        self.t1 = timer.timer()
        user_time = self.t1 - self.t0

        # close client sockets
        client_socket.close()
        js = json.loads(ret_result)
        return {'result': js['result'], 'embtime': long(js['time']),
'usertime': user_time, 't0':self.t0, 't1':self.t1}

    # create a socket and wait for incomming requests on the given port
    # return the socket
    def network_connection(self, server_ip, server_port):
        # listen on the TCP port
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.socket.bind((server_ip, server_port))
        self.socket.listen(MAX_CONNS)

    def parallel_embtime(self, var_list, num_workers):
        embtime = 0
        max = num = 0
        for i in range(0,len(var_list)):
            w = var_list[i]['worker']
            t = int(var_list[i]['embtime'])

            if t != 0:
                if max < t:
                    max = t
                num += 1

            if num == num_workers:
                embtime += max
                max = num = 0

        # add the remaining emb. time
        if num != 0:
            embtime += max
        return embtime

    # check if all nodes in a list are killed
    # return 1 if true; otherwise return 0
    def all_node_kill(self, node_list):
        for i in range(0, len(node_list)):
            node_no = node_list[i]
            if (self.cov[node_no] == 0):
                return 0
        return 1
```