Identifying Defects Related to the Order in which Messages are Received in Message-Passing Systems

by

Milad Irannejad

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Improving the quality of software artifacts and products is an essential activity for everyone working on the development of software. Testing is one approach to reveal defects and faults in software. In recent years, message-passing systems have grown to a significant degree due to the rise of distributed systems, embedded systems, and so forth. In message-passing systems, components communicate with each other through sending and receiving messages. This message-passing mechanism introduces new opportunities for testing programs due to the fact that the time a message is delivered is not guaranteed, so the order in which messages are delivered is also not guaranteed. This non-determinism introduces interleaving and parallelization and subsequently a new source of software defects like race conditions. In this thesis, we have explained a new approach to testing a given component for identifying software faults related to the order in which messages are received by that component. We reorder messages coming to a certain component and deliver them in a different distinct ordering each time. We have three different methods for achieving message reordering: *Blocking*, *Buffering*, and *Adaptive Buffering*. We evaluate the effectiveness of our new testing methods using four metrics: *Ordering Coverage*, *Coverage Rate*, *Slowdown Overhead*, and *Memory Overhead*. We have implemented our *Reordering Framework* on QNX Neutrino 6.5.0 and compared our reordering methods with each other and with the naive random case using our experiments. We have also showed that our testing approach applies to real programs and can reveal real bugs in software.

Acknowledgements

I would like to thank my supervisor, Professor Sebastian Fischmeister, for his guidance. I would also like to thank Zack Newsham for his prior contribution to the implementation. Additionally, I should thank QNX Software Systems for giving us access to the source code of QNX Neutrino micro-kernel.

Dedication

This is dedicated to my family, especially my beloved mother who has been always the source of love and support for me.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

DAG Directed Acyclic Graph

API Application Programming Interface

RTOS Real-Time Operating System

POSIX Portable Operating System Interface

IEEE Institute of Electrical and Electronics Engineers

Chapter 1

Introduction

1.1 Motivation

Consider a message-based software system such as a distributed system or a micro-kernel. In such a system there are some components that communicate and collaborate with each other by sending and receiving messages. Some of the software errors and defects become apparent only when the order in which messages are received changes. As a result of design errors or programming mistakes, components may assume a firm order for receiving messages. This order may hold most of the times, but it is not always guaranteed. In these cases, enforcing an order different from the assumption and seeing how the target component reacts to it is an important part of testing, and it is not easily achievable due to the specific context of an application.

1.1.1 An Example

Figure 1.1.1 shows a case of a message-passing system in which messages can happen in any order. Assume that we have a component R, which receives messages from other components. We have other components, A, B, and C, which send their messages, a, b, and c to R frequently. There is no generic mechanism in place for enforcing a full order or partial order between different messages.

We are interested to see if R can respond to all possible message orderings consistently. For instance, does it make any difference if R receives messages first from A, second from B, and third from C or first from B, second from C, and third from A? Our aim is to present

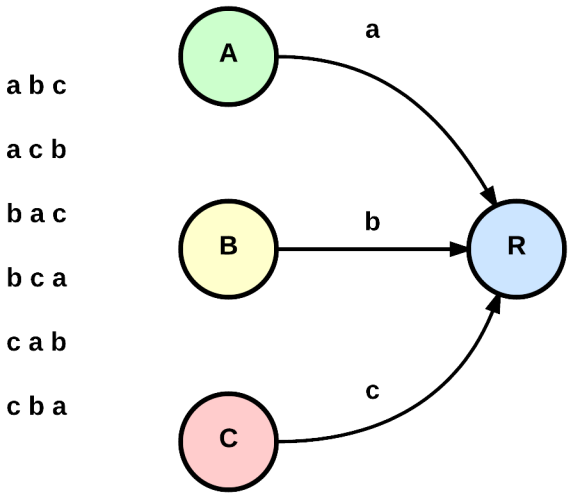

Figure 1.1: An example of possible message orderings in a message-passing system.

a method or some methods that enable us to test as many different message orderings as possible or ideally all message orderings possible to find any software defect (bug) caused by these various message orderings.

Instead of three sender components, imagine we have ten sender components. Assume that the receiver component R sees four distinct message orderings every second on average (we will later on show in our experiments that this assumption can be real). Now, let's calculate how long does it take for R to see all possible message orderings.

Total no. of message orderings:

$$10! = 3628800$$

Total time for seeing all message orderings:

$$\frac{3628800}{4} = 907200\ seconds = 252\ hours = 10.5\ days$$

Now, let's only add 5 more sender components to our example, and calculate how long does it take for R to see all message orderings in this case.

Total no. of message orderings:

$$15! = 1.3076744e + 12$$

Total time for seeing all message orderings:

$$\frac{15!}{4} = 326918592000\ seconds = 3783780\ days = 10366.5\ years$$

What if we have hundreds of components interacting which each other in our message-passing system?

1.2 Definitions

Before formally defining and explaining our problem, we will first go through a set of definitions that we are going to use in the subsequent sections and chapters.

Software Component
: A software component is a part of a software application, which can run independently, and represents a single uniform unit of computation [43, 23].

Software Defect
: A software defect is an error or flaw either in designing or development of a software component, which can lead the integral component to an incorrect result, unexpected behaviour, or a failure [43, 23].

Message-Passing System
: A message passing system consists of a number of components, which only communicate and interact with each other through sending and receiving messages [35].

Message Order
: Each message-passing event is associated with a timestamp. Using these timestamps, we can define a total order relation among messages [18].

1.3 Problem Statement

Having defined our terminology in the previous section, we are now able to characterize our problem as follows:

> "Given a software component in a message-passing system, efficiently identify defects related to the order in which messages are received."

Based on the above definition, we are going to consider all messages received by a particular component in our message-passing system, and then we should change the order in which messages are received and try as many different orderings as possible. Afterwards, we are interested in finding software defects caused by these various orderings.

1.4 Overview of our Approach

The general approach is to enforce as many different orderings as possible for messages that are sent to a particular component in a message-passing system during the execution time. For this purpose, we should identify all components of interest which are going to interact and communicate with each other. Then, we need to specify those interactions in which we are interested in terms of source and destination components. We then provide a mechanism or a set of mechanisms that let us compel a different ordering between messages rather than the one which is the default. For this purpose, we need to be able to stop a message and continue it later on. This would be a critical accomplishment in our methodology.

1.5 Contributions

- First, we propose two methods, the *blocking* method and the *buffering* method, for enforcing different orderings of interest for messages between distinct components in message-passing systems.
- We utilize a statistical technique in the name of reliability analysis (survival analysis) to adjust the time-window for our buffering method in an adaptive manner.
- We implement a framework inside the QNX Neutrino 6.5.0 microkernel which establishes a foundation for re-ordering messages between any set of processes.
- We provide a user-space library for QNX application developers which lets them make use of our framework for trying different message orderings as quickly and easy as possible.
- More interestingly, we also equip the QNX user shell with a utility tool that lets testers and system administrators enforce different message orderings between processes without even having access to their source codes.

1.6 Thesis Organization

This thesis consists of seven different chapters each focusing on an independent aspect of my work. In this section I outline my thesis and sequence of proceedings through the rest of this dissertation. One can skip various parts of this document as needed.

In order to understand the work presented in this thesis an adequate amount of background knowledge in different areas is indispensable for any reader. Much of the pertinent topics are discussed in Chapter 2. We start the chapter with an overview of software testing and then we explicate the concept of software coverage criteria and test oracles. We will provide a concise discussion about software testing alongside formal verification and compare these two different approaches for finding software defects. We proceed to explain message-passing systems and their architectures. Next, we present a statistical method in the name of reliability analysis that we will use later on in our work. Finally, at the end of Chapter 2, we will review the related work has been done so far.

In Chapter 3 we shed light on our methodology to approach our problem, which is stated in Section 1.3. More specifically, we will present three different methods for our problem: the *blocking* method, the *buffering* method, and the *adaptive buffering* method.

In Chapter 4 we first define the metrics we are going to use for evaluating our proposed methods. These metrics are *Coverage*, *Coverage Rate*, *Slowdown*, and *Memory Usage*. With respect to these criteria, we can compare our methods in a quantitative way. Furthermore, we bring a qualitative comparison between our different methods and their pros and cons at the end of this chapter.

In Chapter 5 we will go through the implementation aspects of our work. We first introduce the QNX Neutrino microkernel as an example of message-passing systems, explain its architecture. Next, we will determine our development toolchain and development environment. Finally, we will elucidate how we have implemented our framework both in kernel space and user space.

In Chapter 6 we will illustrate the design of our experiments and how they are setup. After that, we will bring forward our experiment results, and examine them.

Finally, in Chapter 7, we revisit and discuss our problem and the methods proposed in order to summarize our work and provide a succinct conclusion. We will wrap up our dissertation by suggesting extension points for future work.

Chapter 2

Background

In this chapter we will revisit the theories and concepts that we are going to use in our work. More precisely, we briefly review software testing, we explain coverage criteria, we compare software testing with formal verification, and finally we review approaches that are aligned with our own solution.

2.1 Software Testing

With many programming languages, agile frameworks, and code-generation tools emerging every day, development of software applications has grown rapidly and exponentially. Today software includes millions of lines of codes. These expansions alongside the rise of various hardware architectures, platforms, frameworks, and environments make software testing more significant. More importantly, software testing has become a complex process that needs its own expertise [40]. We will define software testing as follows taken from [40]:

"**Testing is the process of running a program with the intention of finding faults.**"

Software testing involves a number of different activities such as deciding on test inputs, designing test cases, running test plans, and reporting results. Although some people are focusing specifically on testing, software testing is an essential task for all. Every developer, engineer, tester, and manager should practice software testing on a regular basis. All software artifacts should come with associated tests. Nowadays, every large organization or group of people has a dedicated test team. Usually, a *test manager* sets *testing policies* and directs a number of *test engineers* who design, run, and analyse test cases [1].

2.1.1 Testing Levels

Software testing can be done at different levels. These levels are usually isolated. Within each level, we design our tests with respect to development level. These levels are as follows [1].

Acceptance Testing evaluates software with respect to elicited requirements.

System Testing evaluates software with respect to architectural design.

Integration Testing evaluates software with respect to sub-system design.

Module Testing evaluates software with respect to detailed design.

Unit Testing evaluates software with respect to implementation and code.

These models are also referred as *V-Model* [60, 34]. Figure 2.1.1 illustrates the software testing activities for each software development activity.

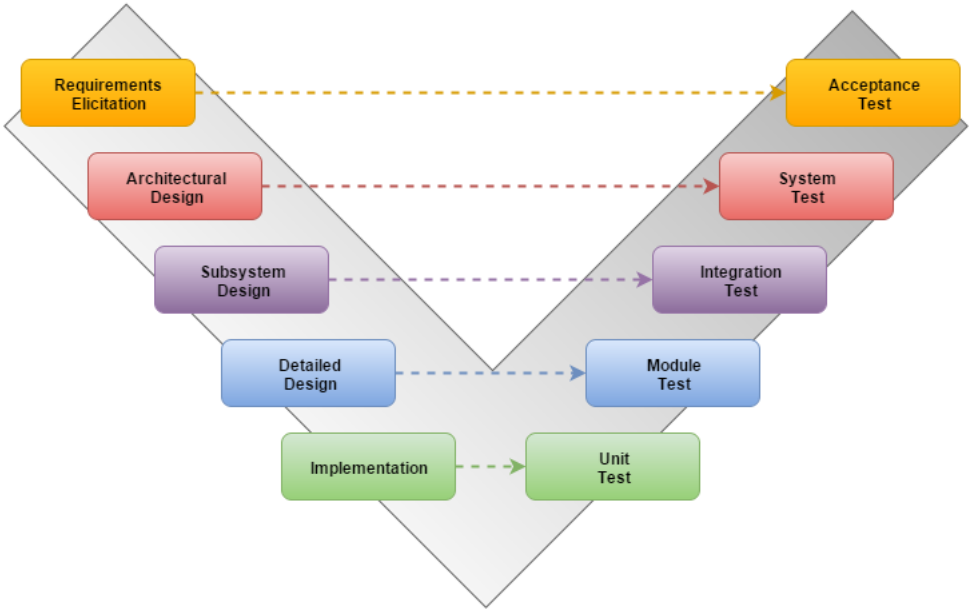

Figure 2.1: The V-Model which shows software development and testing activities hand-in-hand.

In another categorization, software testing levels have been classified based on maturity of testing [5]. These levels are defined as follows:

Level 0 there is no difference between testing and debugging.

Level 1 the goal is to demonstrate that the software works.

Level 2 the intent is to show that the software does not work.

Level 3 the purpose is to reduce the risk of using the software.

Level 4 testing becomes a practice to improve the quality of software.

Our new testing approach, which we will elucidate it in Chapter 3, targets the level 2 in this model.

2.1.2 Testing Automation

Back to the discussion we brought in the beginning of this section, software testing has grown and become more complex as platforms, frameworks, and tools for software development have flourished. As a result, deciding which test inputs we should use and how to develop test cases have become more demanding. Consequently, we need criteria that can help automate the generation of test inputs and test cases. Coverage criteria are the tools for this purpose.

2.1.3 Coverage Criteria

Most of the times it is not possible to test an actual software with all possible inputs due to either huge or infinite input space. Coverage criteria provide a ground for determining test inputs and test cases. We will first define some terms.

Definition 2.1. *A* test requirement *is a particular aspect of a software artifact which a test case must cover or satisfy [1].*

We usually have a set of test requirements denoted by TR. As an example, assume that we are given a *control flow graph (CFG)* [42] of a program, so we know the likely execution paths. Testing a certain execution path can represent a test requirement.

Definition 2.2. *A* coverage criterion *is a guideline or number of guidelines that generates test requirements for a test set [1].*

A coverage criterion imposes test requirements on a test set. In other words, it implicitly determines the set of test requirements we need. In our example, the *execution path criterion* says that all execution paths within a program should be tested. A coverage criterion leads us to a set of test requirements which contains a test requirement for each execution path in the program. We can now define the *coverage* concept as follows:

Definition 2.3. *Given a set of test requirements TR for a coverage criterion C, a test set T satisfies C if and only if for every test requirement tr in TR, at least one test t in T exists such that t satisfies tr [1].*

In other words, if for every test requirement we have at least one test case in our test tests, our test set covers our coverage criterion. Again, in our example, if we have test sets that can test all execution paths in the program, we have covered the *execution path criterion*. The *coverage level* or percentage can be simply defined as follows:

Definition 2.4. *Given a set of test requirements TR and a test set T, the coverage level is the ratio of the number of test requirements satisfied by T to the size of TR [1].*

We may have some test requirements which cannot be covered by any of our test cases. These test requirements are called *infeasible*. Due to large space of test inputs in real applications, achieving a coverage level of 100% is not practical.

2.1.4 Test Oracles

One question that may arise following the determination of our test inputs and test cases is: how can we understand whether or not the software responds correctly to test cases? *Test oracles* can provide the correct output for test inputs to a software. Testers can verify the output of their tests using test oracles. A complete test oracle knows the expected behavior for every possible input. Providing complete test oracles is not a straightforward task, and it should be automated. *Oracle problem* is the challenge of finding expected outputs and behaviors [52].

A *Test oracle generator (TOG)* automates and solves the test oracle problem. Given a test execution $< input, output >$ pair, TOG tells you if the *output* is the expected correct output for the *input* or not [48].

In some application domains we programs for which we do not have a reliable test oracle or automatic generation of test oracle is not doable. These programs are called

non-testable programs [66]. In this case *pseudo-oracles* are used. A pseudo-oracle makes use of multiple implementations of an algorithm. For a given input the outputs from these alternative implantations are compared for a conclusion [39].

2.1.5 Testing vs. Formal Verification

Another approach to finding errors and faults in software is using formal mathematical methods to verify a program. Formal verification methods sometimes have been referred to as static testing. These methods assume an *execution model* and analyze the mathematical model of a program in order to show that it works with respect to its specification. *Model checking* is one technique for formal verification [3].

Software testing techniques investigate programs at runtime while they are working whereas formal verification techniques investigate programs statistically prior to execution. Testing methods try to show that there exist a bug/defect in the program while formal verification methods attempt to show that there is no bug/defect in the program. As a result, if our testing process does not reveal any error in the program, it does not imply that our program is defect-free [20].

Formal verification methods are not scalable. They soon face the state-explosion problem. On the other hand, software testing is applicable to big-scale real applications in use. Moreover, assuming an execution model which corresponds to the actual execution model brings more complexity into calculations. Another challenge with formal verification is that the specification of a given program should be represented formally and mathematically [20].

2.2 Reliability Analysis

Reliability analysis or *survival analysis* is statistical method predicting time-to-event variables. It is similar to regression analysis with time to occurrence of an event as a response variable. In a population we have measured how long it take for an individual to experience an event of interest alongside a number of other independent variables. We aspire to devise a model that lets us predict the time-to-event for every individual based the values of its independent variables.

The advantage of reliability/survival analysis over regression analysis lays down in considering censored data. Censored data are those individuals that we do not know the

exact time-to-event for; however, we do know that the event of interest has not occurred up until a certain point in time, or we know that the event of interest has occurred to them before a certain point in time. Regression analysis techniques will dispose of these data points; however, they are still carrying valuable information. Survival analysis methods take these data points into account as well.

In Section 3.4, we will define and explain concepts and equations of reliability/survival analysis in the context of our work (reordering framework).

2.3 Related Work

In this section we review approaches to reordering messages in message-passing systems that are aligned with our own solution. Prior to reviewing these approaches we would like to revisit our work briefly. We have provided a new approach for testing software applications on message-passing systems by reordering the messages received by a given component.

In [64, 65, 15] authors presented a new method for software failure recovery using message reordering, message logging, and message replaying. They proposed a multi-stage recovery method. In stage one, for example, they bypass the faulty process by regenerating the messages it sent from its message logging. If that does not work, they reorder the messages from message logging and replay them again in stage two. In each stage, they increase the scope of recovery in the system. By reordering messages, they managed to mask those bugs related to the ordering of messages.

In [62] authors recommended a perturbation technique that introduces timing-related defects in message-passing systems. Their technique increases the message ordering coverage in a non-deterministic way in order to disclose those bugs which are not easy to test under normal circumstances. Their work is analogous to ours. Their work is mainly focused on sender-based perturbation of messages which does not guarantee message reordering. On the other hand, we have provided a receiver-based perturbation that ensures message reordering.

Authors in [54] suggested a structural testing criteria and coverage for message-passing applications. Their tool tracks the control flow and data flow of programs by distinguishing the sequential and parallel parts of programs.

Detecting race conditions is another trend for testing message-passing systems. In [41], authors proposed an algorithm for finding data race in message-passing systems using

logical clocks. MARMOT [29] is another tool for detecting deadlocks and race detections in standard message-passing applications. It improves reproducibility of non-deterministic faults.

There also a number of testing techniques for message-passing systems by profiling and trace analyzing. In [61] authors have implemented a new tool for dynamically testing message-passing systems. Their tool intervenes the message-passing APIs, profile message-passing calls, and then runs an analysis in order to to detect errors such as deadlocks.

Model checking techniques and other formal verifications methods are other approaches to find bugs in message-passing systems. Authors in [8] described a new tool for model checking using the type information provided by the programmer from source. Authors in [53] have reviewed works done on formal analysis of message-passing systems.

Chapter 3

Methodology

In this chapter we will explain our approach to the problem stated in Chapter 1, and then we will illustrate our methods alongside some examples. We are going to present three different methods for reordering the messages in a message-passing system as follows:

- Blocking method
- Buffering method
- Adaptive Buffering method

3.1 Our Approach

From our problem statement elaborated in Chapter 1, we have a number of components that send messages to another component from time to time. So, the receiver components see messages from sender components in a fixed or random order. Having secured these components, we are interested in changing the order in which are sent to the receiver component and systematically apply all message orderings possible.

For reordering messages we have to be able to intercept every message of interest and pause it. Later on, we should also be able to release the paused message and resume it upon fulfilment of some criteria. These criteria are dependencies between messages of interest. As a simple example, assume that we have two components A and B which send messages to component R. Without any mechanism for reordering in place, A usually sends

Sender	Receiver	Event	Dependency
A	R	a	b
B	R	b	c
C	R	c	-

Table 3.1: Definitions for an example in a hypothetical message-passing system.

a message to R first, and then B sends another message to R. Having a reordering method in place, when A sends its message to R, we stop it and release only when B has sent its message to R. So, in this case, we changed the order [AR, BR] to [BR, AR]. As a result, we need to designate a message as a dependency for another message and by chaining these dependencies we can define the desired ordering between messages. By having a pausing-resuming mechanism implemented we can enforce our desired message order. So finally, by changing the orders of interest on-the-fly we can systematically explore all message ordering possible.

Before further exploring our methods we will consider a hypothetical message-passing system in which we have a couple of sender components and a receiver component. We define and denote what we are going to use in the rest of this chapter as follows.

A, B, and C are sender components, and R is receiver components in our message-passing system. We call the message from A component to R component a, the message from B component to R component b, and the message from C component to R component c. The message from A to R (a) is dependent on the message from B to R (b), and the message from B to R (b) is dependent on the message from C to R (c). So, this dependency chain implies that the message c should happen first, and then the message b can happen, and finally the message a is allowed to happen. So, we expect the component R to receive messages in the order [c, b, a] instead of [a, b, c].

The other thing that we would like to make clear is the way that we illustrate our examples. We will use a diagram similar to sequence diagram in UML [44]. Each color relates to a software component, each horizontal rectangle at the top of the diagram represents a software component, each vertical slender horizontal rectangle denotes the lifetime of a component, and each horizontal arrow connotes a message-passing from a source component to a destination component. Moreover, each message passed is labeled with on of the events represented in Table 3.1. Figure 3.1 shows an example of sending messages in our hypothetical message-passing system.

In this example, there is no reordering mechanism in place. First, A sends a message to

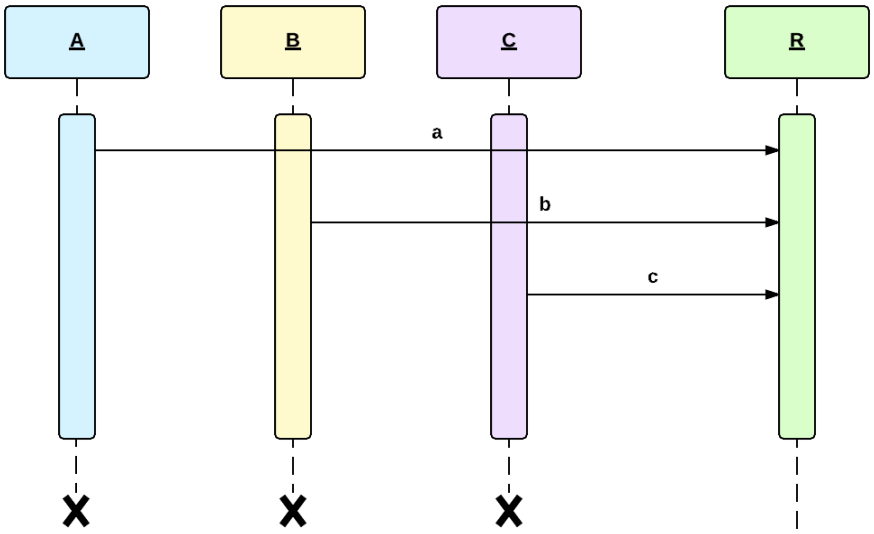

Figure 3.1: An example of message-passing without any reordering method.

R, and it is delivered immediately. Second, B sends a message to R, and it is also delivered immediately. Finally, C sends a message to R, and it is delivered instantly. As a result, the ordering will be [AR,BR,CR] or [a,b,c]. We want to change this ordering to [CR,BR,AR] or [c,b,a], so the receiver component can experience a different ordering for the sake of testing. In the subsequent sections, we describe our various methods for this purpose.

3.1.1 Messages Dependency Graph

In our reordering framework we represent the dependencies between messages through a dependency graph. Dependency graphs have been used in a variety of contexts and applications. They have been used for static analysis and testing of programs [28, 63], for program execution analysis and dynamic slicing [9, 4, 32], and for scheduling purposes [30, 47].

Figure 3.1.1 shows an example of a message dependency graph that can be defined in the reordering framework. In such a graph each node or vertex represents an event that is a message from a source component to a destination component. These source and destination components can be seen inside each node. The first capital letter implies the source, and the second capital letter implies the destination. Each edge portrays

a dependency between two events (messages). For example, the edge from AR to BR says that the occurrence of AR is dependent on the occurrence of BR, which means that component B first has to send a message to component R, and then component A can send its message to component R.

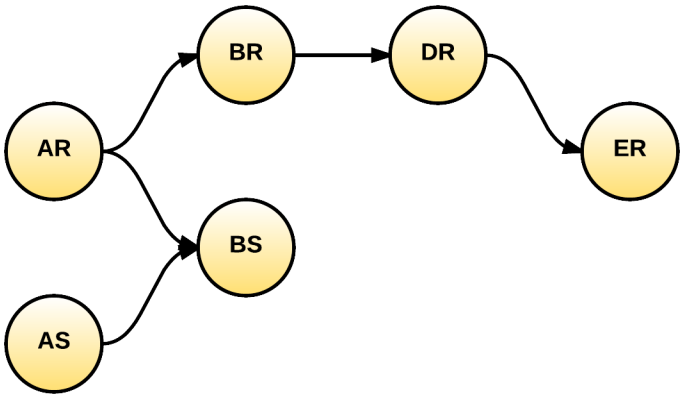

Figure 3.2: An example graph which shows dependencies between messages.

The dependency graph represents what we would like to see not what the application does. A dependency graph only allows certain orderings for defined events (messages). The following message orderings are valid with respect to the example graph shown in Figure 3.1.1:

- BS AS
- ER, DR, BR, BS, AR
- ER, BS, DR, BR, AR
- ER, DR, BR, BS, AS, AR
- BS, ER, DR, BR, AR, AS

As you may have inferred from this example, the dependency graph is a general way of representing dependencies between messages, and we can potentially impose a dependency between any pair of messages. However, we prevent cyclic dependencies from being created

in our dependency graph by disabling those dependencies, which causes the addition of a cycle. A cycle in a dependency graph causes a deadlock in the system, and it should thus be eliminated. In other words, the dependency graph should be a Directed Acyclic Graph (DAG).

You may ask yourself where this dependency graph originates and how dependencies should be decided upon in the graph. As we mentioned earlier, each dependency graph filters all possible message orderings and only allows certain orderings to occur. So, if we are interested in a specific message ordering to happen, we should define a number of dependencies in such a way that the resulting dependency graph forces our ordering of interest. Consider the linear dependency graph shown in Figure 3.1.1. If you interpret this graph, you will conclude that this dependency graph can only allow the ordering [CR, BR, AR], the same ordering we have defined in Table 3.1.

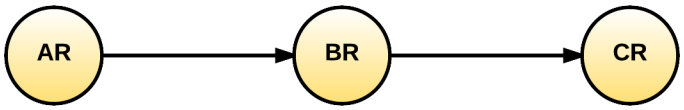

Figure 3.3: An example of linear dependency graph for enforcing a specific order.

To put in a nutshell, we will review our approach once more as follows:

1. We start by assigning available tokens to components of our interest using their PIDs or names.
2. We specify those messages between tokenized components in which we are interested in by registering as events.
3. We then consider those orderings between events (messages) we want to apply or test.
4. Now, we enforce each ordering by defining the corresponding dependencies for the given ordering.

In order to enforce all orderings of interest, we need the dependency graph to be configurable on-the-fly during runtime.

3.2 Reordering Methods

In the subsequent sections we explain our three methods of reordering for reordering messages between components in a message-passing system. These methods are called *blocking*, *buffering*, and *adaptive buffering*.

3.2.1 Blocking Method

One method we can use for changing the order in which messages are received by R is making each message-pass dependent on another message-pass. So, in this method we define each message as an event, and an occurrence of a certain event can be dependent on occurrences of other events. Figure 3.2.1 shows an example in which we are interested in forcing the order [CR,BR,AR] or [c,b,a]. To this end, we first define our events and dependencies between them. We would like to emphasize the fact that it is not necessary to know these dependencies upfront. These dependencies only impose a specific message ordering that is [c,b,a] in this example. So, we simply represent every ordering of interest using a set of dependencies which forms a linear dependency graph.

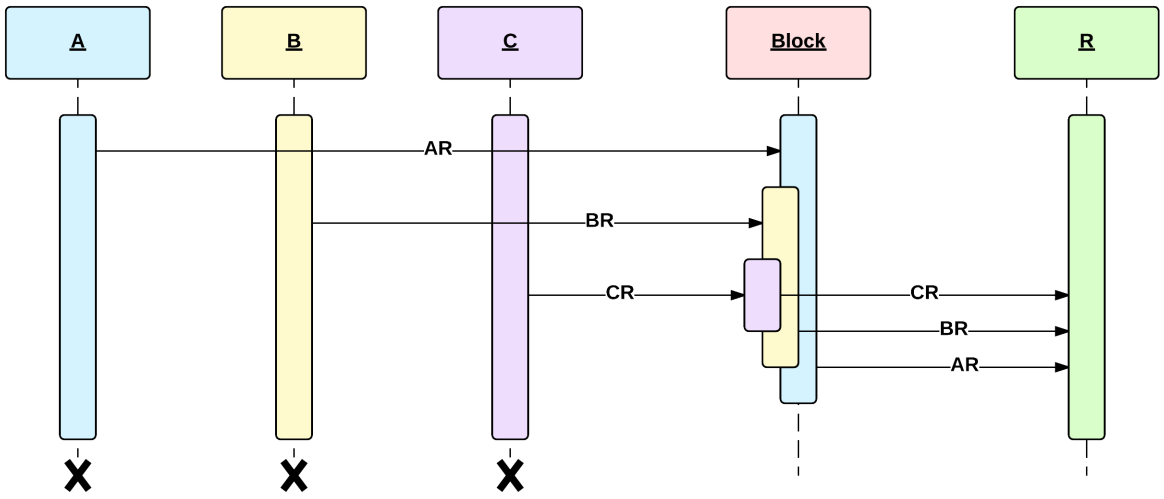

Figure 3.4: An example of blocking method for reordering messages.

Using the blocking method, if AR (a) arrives first, we block it since its dependency, BR (b), has not yet happened. Similarly, when BR (b) arrives next, we block it again since its

dependency, CR (c), has not yet occurred. Ultimately, once c arrives, we let it continue, and release all other events (messages) dependent on it which are blocked. It means we first release BR (b), and then AR (a).

3.2.2 Buffering Method

Instead of blocking individual messages, we can buffer all messages by default and periodically empty the buffer and release buffered messages based on order of interest. Figure 3.2.2 shows an example of this method. Once AR (a), BR (b), and CR (c) arrive, we buffer all of them and then every *50ms* we check all the buffered messages, and release them in the order of interest which is CR (c) first, BR (b) second, and AR (a) last. In this case, the order that the component R sees is [CR,BR,AR] or [c,b,a].

One challenge with the buffering method is the frequency of emptying the buffer. How quickly must we empty the buffer and release messages? On the one hand, we prefer to see and buffer all messages before a timeout. On the other hand, we should be careful about not choosing a long timeout since it is not safe to delay messages more than a few ten milliseconds roughly speaking. Delaying messages more than a certain threshold could have serious consequences in some application domains like embedded software or safety-critical software. But what if our timeout frequency is so short that we cannot see a new message ordering? Consider the example shown in Figure 3.2.2.

In the example shown in Figure 3.2.2 we have the same set of components and the same dependency. The components A, B, and C send their message AR (a), BR (b), and CR (c) respectively. The natural order of messages is [AR,BR,CR] or [a,b,c], but we want to force another ordering which is [CR,BR,AR] or [c,b,a]. In this example, only the message AR (a) and BR (b) have happened before the first timeout. Since we have only seen two (AR, BR) out of three messages (AR, BR, CR) of interest when the first timeout comes, we partially reorder them and deliver messages in ordering [BR, AR]. The message CR (c) has occurred after the timeout. When the second timeout comes, the message CR (c) gets released. Thus, the message ordering that the component R has seen in this case is [BR,AR,CR] or [b,a,c] which is a different message ordering, but not the one we were interested in initially.

As the example above demonstrates, the problems with the buffering method are as follows:

- Choosing a well-balanced timeout for every single application is a challenge.

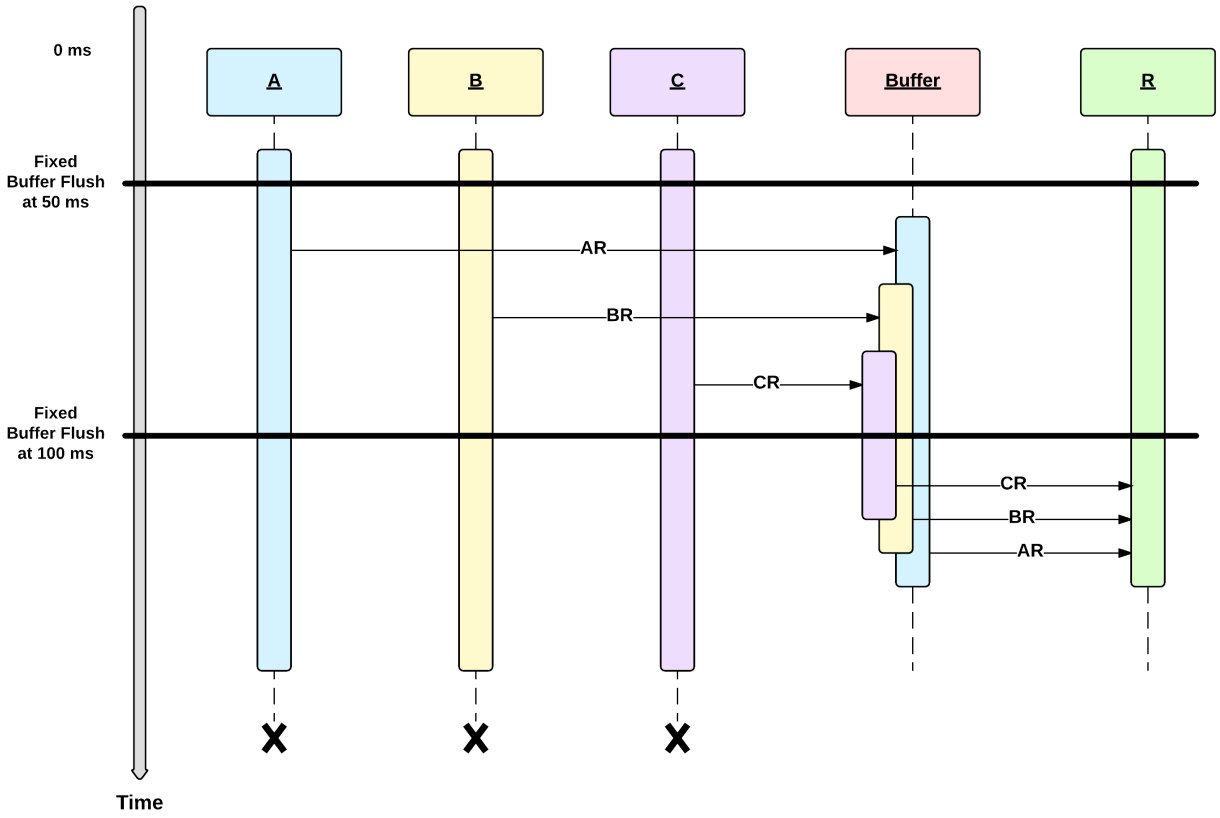

Figure 3.5: An example of buffering method for reordering messages.

- The timeout value is fixed each time, and it does not consider the dynamic behavior of an application.

In order to overcome these challenges and improve the buffering method, we introduce another method that tries to adjust the timeout frequency dynamically on-the-fly with respect to the behavior of the messages of interest.

3.2.3 Adaptive Buffering Method

We improve the limitations of the buffering method by dynamically adjusting the timeout values. We start by the same example shown in Figure 3.2.2 and show how the adap-

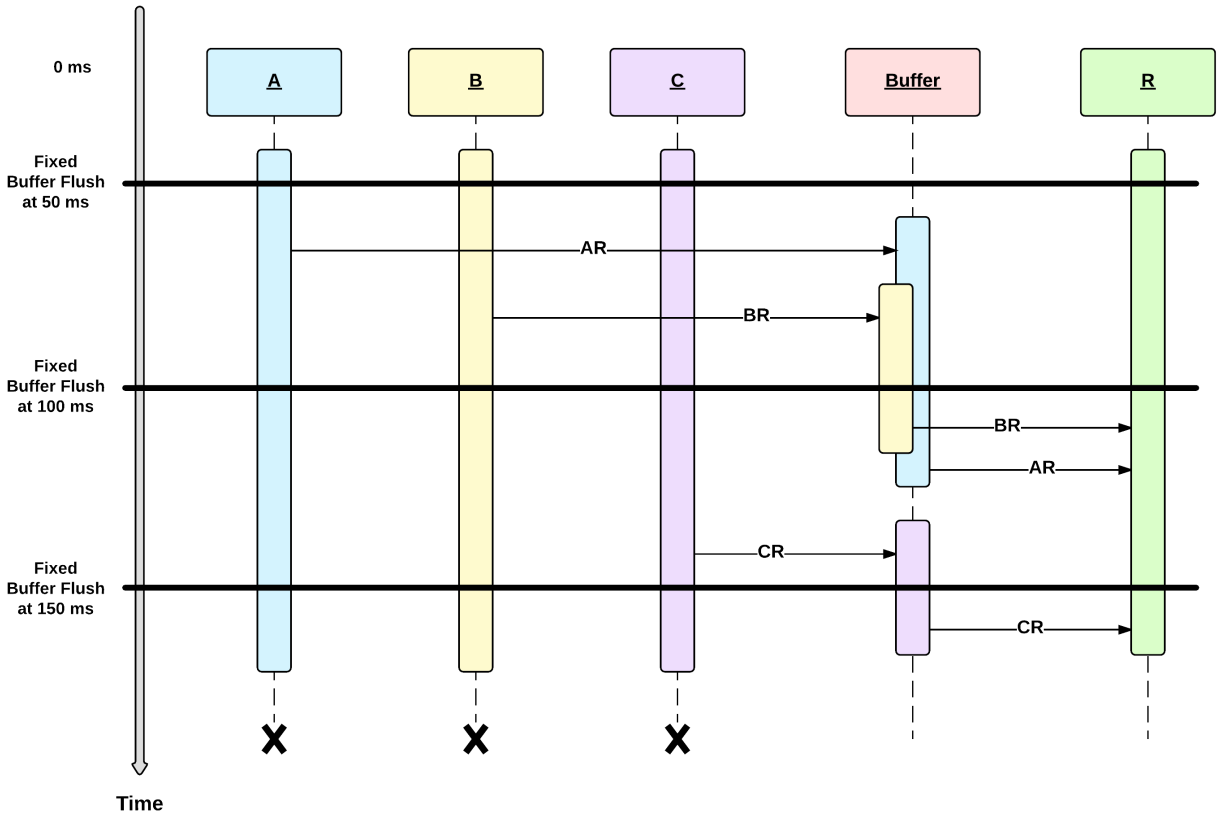

Figure 3.6: An example of buffering method which does partial reordering.

tive buffering method can improve the problems mentioned. Figure 3.4.4 illustrate the mechanism of the adaptive buffering method.

In this example, we have the same set of components, messages, and dependencies. Components A, B, and C send their messages AR (a), BR (b), and CR (c) respectively, and want to change the ordering from [AR,BR,CR] to [CR,BR,AR]. Akin to the buffering method, we have the concept of a timeout in this method. We start with an educated estimation for timeout values. In this example, we begin with the same timeout value we have used in Figure 3.2.2, *50ms*. As illustrated in the diagram, message AR (a) comes and then message BR (b) comes. Subsequently, the first timeout happens and we again partially reorder the messages and release them in the order [BR,AR]. After that, the message CR (c) happens. The second timeout comes with the same value of *50ms*, and

the message CR (c) gets released. So, the ordering that the component R sees is the same order [BR,AR,CR] we have achieved in the previous example.

The power of the adaptive buffering method comes into effect from this point onwards. Based on the results of the first and second timeouts we adjust the value of timeouts. We accomplish this by carrying out a survival analysis in hope that we will see all of the messages of interest the next time, and we can fully reorder them rather than doing partial reordering. As a result, we increase the timeout value from *50ms* to *100ms*. When the third timeout comes, all messages of interest AR (a), BR (b), and CR (c) have happened, so we can release them in the order [CR,BR,AR], the same order we want to enforce.

Furthermore, we can bound the timeout value between a lower bound and upper bound. The lower bound guarantees that our timeouts will not come too fast, and we will not incur too much overhead to the system. The upper bound ensures that we will not delay the release of messages too long, so we can guarantee the delivery of each message after at most a certain amount of time.

3.3 Topological Sort of Messages

Earlier in this chapter, when we were explaining our buffering and adaptive buffering, we mentioned that if we have seen all message of interest before a timeout, we partially reorder messages and release them. This partial reordering is done through a topological sort on dependency graph. Topological sort or toposort is an algorithm for ordering the vertices of a directed acyclic graph in such a way that all edges are pointing in one direction (outward), and no edge is pointing backward [51]. On-the-fly topological sort at runtime has been also used in a number of applications [25, 46].

Consider the same dependency graph we have shown in Figure 3.1.1. If we do a topological sort on this graph and order its vertices, we will have the graph shown in Figure 3.3. Notice that these two graphs are isomorphic, which means they are identical. As you can see in the topologically sorted dependency graph, all edges point to one direction. This is only possible if there is no cycle in the directed graph.

Now, based on the topological order of our dependency graph at any given time, we can decide in what order we should release the messages with the intent of satisfying as many dependencies as possible and providing a partial ordering. For example, imagine that message AR, AS, BR, and DR have occurred before a timeout. At timeout, we follow the topological order of vertices (messages) of the dependency graph and release each buffered message with respect to this order. In the context of our example, we will follow the order

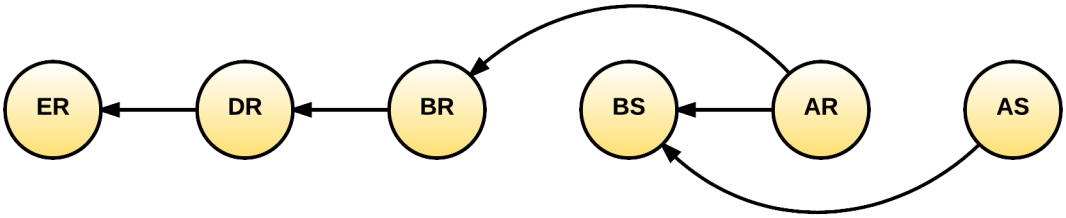

Figure 3.7: An example of a dependency graph in topological order.

ER, DR, BR, BS, AR, and AS and release each message as if it is in the buffer. From there we will release the messages DR, BR, AR, and AS.

The topological sort algorithm is implemented by performing a depth-first search algorithm on an acyclic directed graph [51]. It is worth mentioning that we do not need to run the topological sort algorithm on every single timeout; we only need to compute the topological order of our dependency graph whenever we change the configuration of our dependency graph either by adding a new dependency or by removing an existing dependency.

3.4 Survival Analysis Review

We just mentioned in Section 3.2.3 that the adaptive buffering method tries to dynamically adjust the right value for timeouts on-the-fly. To this end, we need a precise and reliable analysis to calculate a fit timeout using the past historical data. Such a method should take context-specific properties of our data into account.

Survival analysis is a statistical technique used in a variety of contexts and domains for predicting time to occurrence of an event of interest [58, 26, 38, 27]. Survival analysis or reliability analysis [50] is similar to regression analysis in which the response variable or dependent variable is time, and we are interested to know when an event will most likely recur. Survival analysis can consider *censored data* [58, 27] which regression analysis methods ignore as missing data.

In survival analysis, like any other statistical method, we sample from a select population of individuals. For each individual in the sample, we measure some dependent or predictor variables and a dependent or response variable. Based on these observations,

we want to infer a model that helps us to predict the response variable value using the values of independent or predictor variables that ideally reflect every individual in the entire population. This is how a regression analysis method functions without censored observations.

In survival analysis, our response variable is always time-to-event T which is the time takes for an individual until an event of interest happens for that individual. This event of interest is called *failure* or *death*, but it can be any event of interest. It could be time-to-failure, time-to-accident, time-to-restart, time-to-delivery, or any other event in which we are interested. We measure T for each individual in our sample alongside dependent variables, if any, and we report T either as a continuous or discrete measure [58].

There are some cases in which we may lose some individuals before they experience the event of interest. For example, let's say that we want to predict the time-to-accident for beginner drivers. We select a number of novice drivers and follow up with them during a certain period. We might lose track of some of our drivers. For example, an individual may move out to another city before our experiment ends, or a driver may stop driving for some reason. However, we know that the event of our interest (accidents) has not happened for an individual before leaving the population. Furthermore, our experiment running time may end for some reason, and there may still be accident-free drivers. Although we do not know the actual time of an accident for these drivers, we know that the event of our interest has not happened for them up to a certain point in time. These individuals are called *censored data*. Regression analysis methods simply ignore these data because their values for T are missing. However, these censored data are still carrying valuable information, which can be taken into account and help us drastically improve the accuracy of our prediction method.

In the context of our reordering framework, our population is the set of all registered events that contain a message of interest. An individual is a single message from a source component to a destination component. We want to predict the time it will take for a certain message to recur.

3.4.1 Definitions

We define T as a non-negative continuous random variable which represents lifetime or time-to-event for individuals in a population. We denote the *probability distribution function (p.d.f.)* by $f(t)$ and the corresponding *cumulative distribution function (c.d.f.)* by $F(t)$ [58].

$$F(t) = \int_0^t f(x)dx \tag{3.1}$$

By definition, $F(t)$ is the probability that an individual experiences the event of interest before time t. Likewise, we can define the probability that an individual survives or does not experience the event of interest until time t as follows:

$$S(t) = P(T \geq t) = 1 - F(t) = \int_t^\infty f(x)dx \tag{3.2}$$

$S(t)$ is called *survival function* or *reliability function*. Note that $F(t)$ is a cumulative function which implies that $F(t)$ is a monotonic increasing function. This yields that $S(t)$ is a monotonic decreasing function. At $t = 0$ the probability that an individual survives is 1, and at the $t = \infty$, the probability that an individual survives is 0. From the equations 3.1 and 3.2, we can derive the following equation [58]:

$$f(t) = \frac{dF(t)}{dt} = -\frac{dS(t)}{dt} \tag{3.3}$$

Another important concept in survival analysis is *hazard function*. Given the fact that an individual has survived up to time t, the hazard function ($h(t)$) gives you the immediate rate of failure at t which is a number in $[0, \infty)$ range. It is defined as follows:

$$h(t) = \frac{f(t)}{S(t)} = \frac{-\frac{dS(t)}{dt}}{S(t)} = -\frac{d \log S(t)}{dt} \tag{3.4}$$

It is important to know that hazard function is a *rate* function rather than a probability function. The *cumulative hazard function* also can be simply described as follows:

$$H(t) = \int_0^t h(u)du = -\log(S(t)) \tag{3.5}$$

Using equations 3.4 and 3.5, we can define $f(t)$ and ($S(t)$ functions based on hazard function in the following way:

$$S(t) = e^{-H(t)} = e^{-\int_0^t h(u)du} \tag{3.6}$$

$$f(t) = h(t)e^{-H(t)} = h(t)e^{-\int_0^t h(u)du} \tag{3.7}$$

The last thing we define is the mean for response variable T and *mean residual life* at time t. The mean residual life for individuals at time t is the time left until an individual experiences the event of interest [58].

$$E(t) = \int_0^\infty tf(t)dt = \int_0^\infty S(t)dt \tag{3.8}$$

$$mrl(t) = \frac{\int_t^\infty S(u)du}{S(t)} \tag{3.9}$$

3.4.2 Non-Parametric Models

Non-parametric estimation of time-to-event T is used for expressive purposes. They are helpful for investigating the general structure of *survival function* and hazard function. One can then use a parametric survival analysis method with independent variables involved in the calculations. Because we have not used any non-parametric estimators or models, we do not bring any more explanation for them. For more information on and examples of non-parametric survival analysis models, please refer to provided references [58, 26, 38, 27].

3.4.3 Parametric Models

Parametric models assume a parametric form for hazard and survival functions. They also let us develop a regression model and include independent variables in calculations to predict the time-to-event T variable based on the values of dependent variables. Table 3.2 summarizes the most frequent parametric models used for parametric survival analysis.

Figures 3.4.4 and 3.4.4 show *probability density*, *survival*, *hazard* functions for *Exponential* and *Log-Logistic* distributions, respectively. These curves can be investigated more thoroughly to interpret the survival probability and hazard rate as time passes. For example, the Figure 3.4.4 illustrates that in exponential distribution, the chance of failure is great at the beginning, and then, it drops down greatly. Figure 3.4.4 demonstrates different shapes of survival or hazard functions for *Weibull* distribution. For example, if we have $\lambda = 1$ and $\alpha = 3$, the chance of failure is initially quite low, gradually rises, and eventually

Parametric Model	**h(t)**	**S(t)**	**f(t)**
Exponential	$\lambda, \lambda > 0$	$e^{-\lambda t}$	$\lambda e^{-\lambda t}$
Gompertz	$\lambda e^{\alpha t}$	$e^{-\frac{\lambda}{\alpha}(e^{\alpha t}-1)}$	$\lambda e^{\alpha t - \frac{\lambda}{\alpha}(e^{\alpha t}-1)}$
Weibull	$\lambda\alpha(\lambda t)^{\alpha-1}$	$e^{-(\lambda t)^{\alpha}}$	$\lambda\alpha(\lambda t)^{\alpha-1}e^{-(\lambda t)^{\alpha}}$
Log-Logistic	$\frac{\lambda\alpha(\lambda t)^{\alpha-1}}{1+(\lambda t)^{\alpha}}$	$\frac{1}{1+(\lambda t)^{\alpha}}$	$\lambda\alpha(\lambda t)^{\alpha-1}(1+(\lambda t))^{-2}$

Table 3.2: Common parametric models for parametric survival analysis [58, 38].

falls. If we have $\lambda = 1$ and $\alpha \leqslant 1$, the distribution becomes similar to exponential distribution. We would like to mention that the *Weibull* and Log-Logistic distributions are analogous.

One can investigate the shape of survival and hazard functions using a non-parametric survival analysis before choosing a parametric model. We then approximate the parameter values by optimizing the corresponding *likelihood* function.

For the sake of simplicity and coherency we have not included many details about other distributions and the methods for estimating the parameters. For a more in-depth explanation of survival analysis field and parametric models, please refer to the major textbooks [58, 26, 38, 27].

3.4.4 Weibull Regression Model

The next step is developing a regression model for time-to-event response variables. To recap, we have some observations. For each observation, we have measured a set of independent variables denoted by $\underline{x}$ and a dependent variable T, time-to-event, which is the lifetime of an individual until it experiences the event of interest. Based on these data we are going to calculate a regression model which lets us predict the lifetime of any individual using its independent variable values. For each parametric model, we derive a slightly different regression model.

Based on our observations provided in Chapter 6, we have chosen the *Weibull* distribution and corresponding parametric model. Thus, we will use a Weibull regression model.

For the purpose of our regression analysis, we define the *hazard function* as follows [58]:

$$h(t|x) = h(t)e^{\underline{x}'\beta} = \alpha\lambda^{\alpha}t^{\alpha-1}e^{\underline{x}'\beta} = \alpha(\lambda(e^{\underline{x}'\beta})^{\frac{1}{\alpha}})^{\alpha}t^{\alpha-1} = \alpha\tilde{\lambda}^{\alpha}t^{\alpha-1} \tag{3.10}$$

This is the same form we used for the hazard function that is now replaced by $\tilde{\lambda}$. This $\tilde{\lambda}$ has now the independent variables incorporated in it [58].

If T follows *Weibull* distribution with parameters λ and α, then $Y = log(T)$ follows an *extreme value distribution* with $\mu = -\log(\lambda)$ and $\sigma = \frac{1}{\alpha}$. Consequently, we will have the following equation [58]:

$$Y = \log(T) = \mu + \sigma Z \tag{3.11}$$

We can further incorporate our independent variables in the above equation simply as follows in which we have $\tilde{\mu} = -\log(\tilde{\lambda})$

$$Y = \log(T) = \tilde{\mu} + \sigma Z = -\log(\lambda) - \underline{x}'\sigma\underline{\beta} = \beta_0^* + \underline{x}'\underline{\beta}^* + \sigma Z \tag{3.12}$$

Now, we have a regression form which is very similar to the ordinary regression we know. We can estimate the parameters λ, $\sigma = \frac{1}{\alpha}$, and $\underline{\beta}$ by minimizing the error term Z. It is worth noting that Z does not follow a normal distribution. Thus, we cannot use the least-squares method.

As mentioned earlier in this section, we have considered the *Weibull* distribution and a parametric model for our adaptive buffering method. We would like to clarify the survival analysis we are using for our adaptive buffering method as follows:

Population
The set of all registered events/messages of interest.

Event of Interest
The occurrence of a registered event/message.

Time-to-event T
The period of time which takes for a registered event/message to happen again since its last occurrence.

Independent Variables
The source component (x_1) and destination component (x_2) of a registered event/message.

Finally, we can present our Weibull regression model for our reordering frame in the following form:

$$\log(T) = -\log(\lambda) - \frac{1}{\alpha}\beta_1 x_1 - \frac{1}{\alpha}\beta_2 x_2 \tag{3.13}$$

For more information on regression analysis in survival analysis and *measure of effect*, please refer to reference [58].

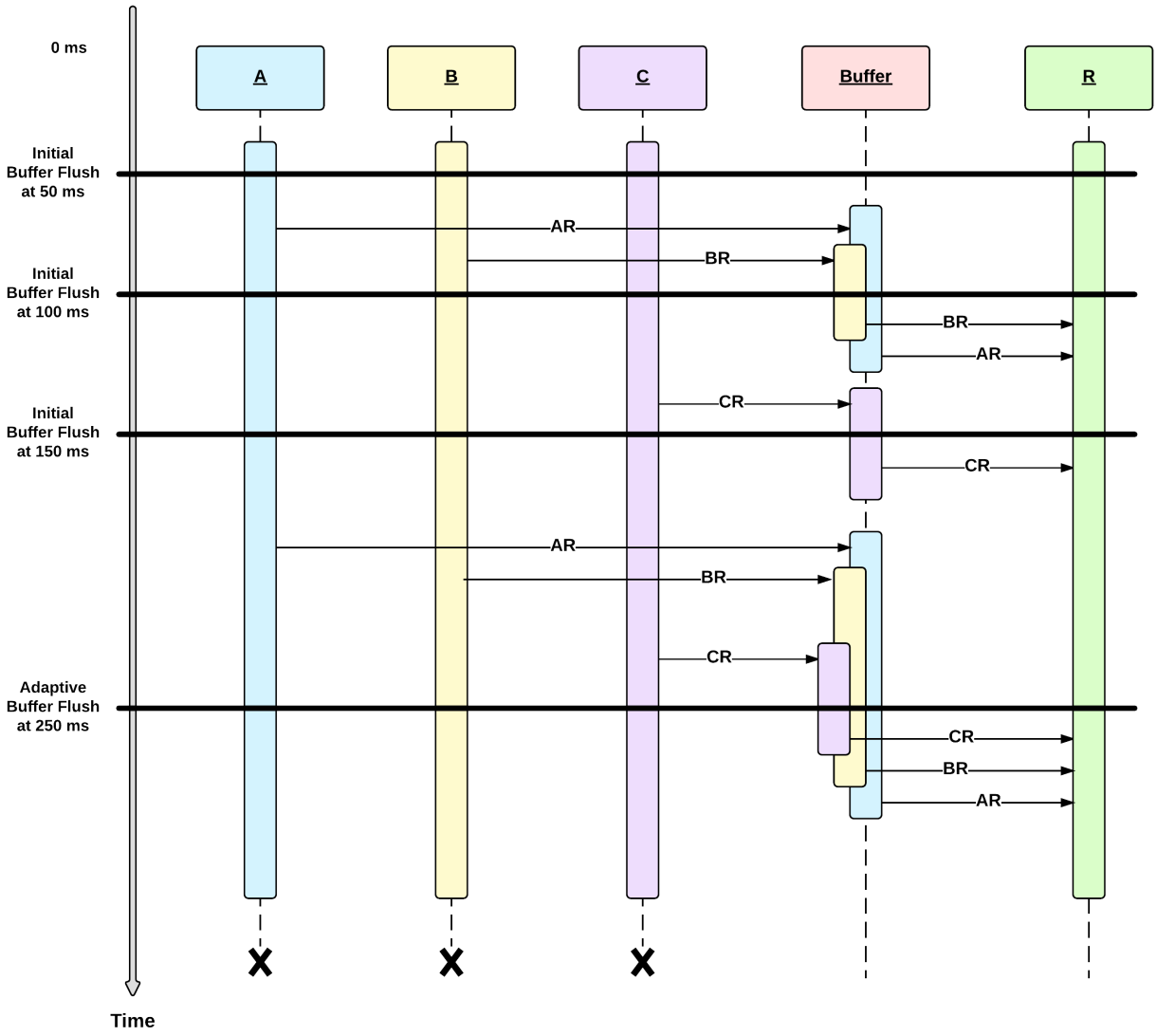

Figure 3.8: An example of adaptive buffering method for reordering messages.

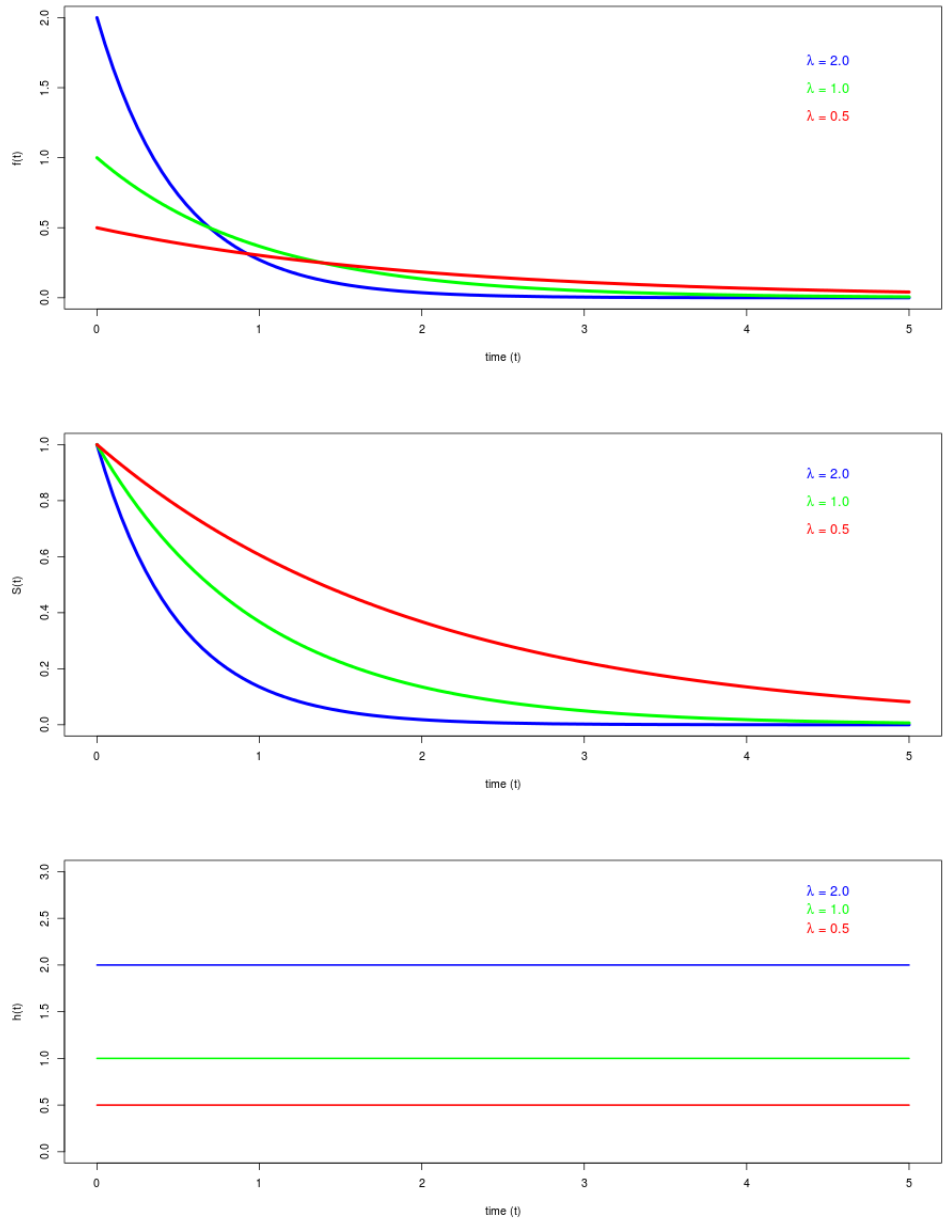

Figure 3.9: Examples of probability, survival, and hazard functions for Exponential distribution.

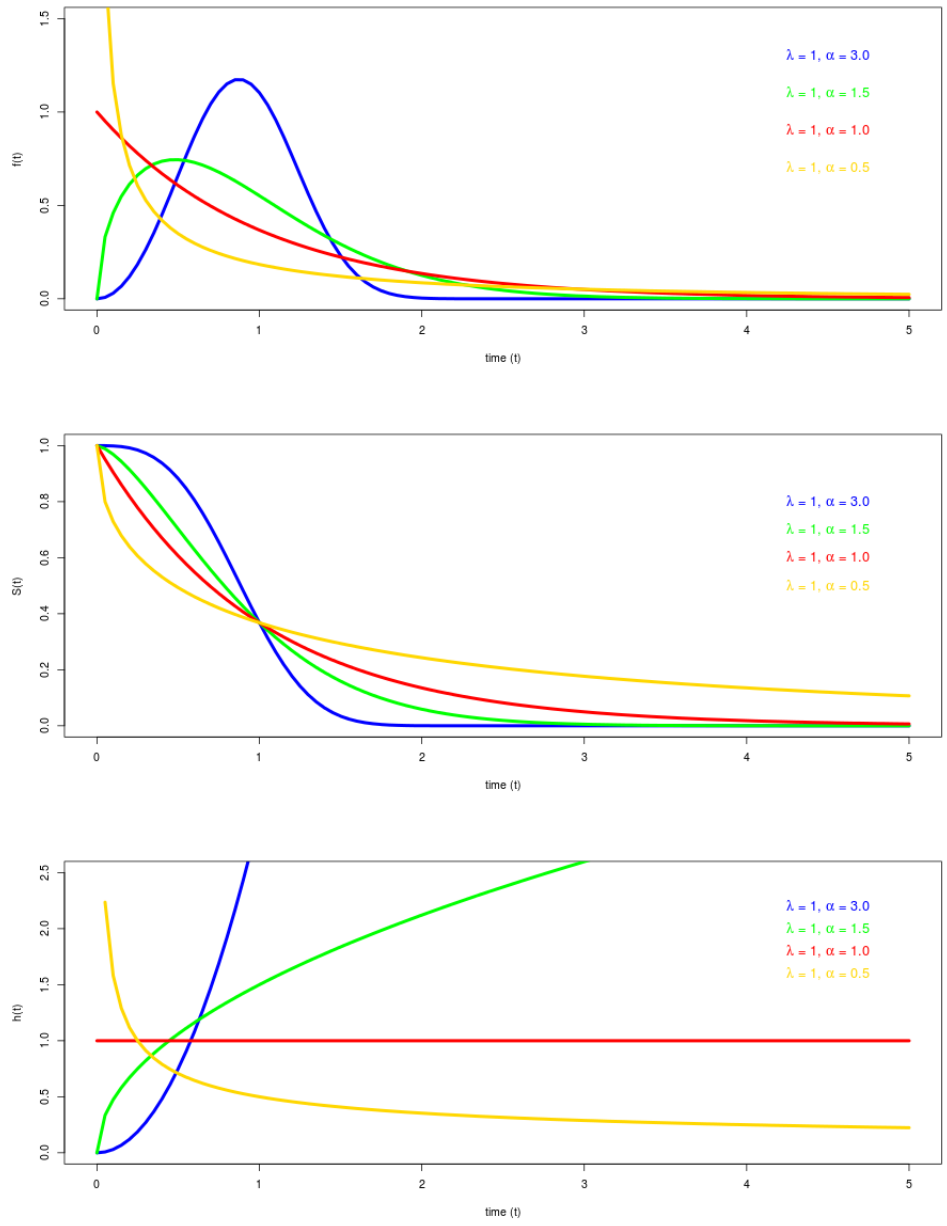

Figure 3.10: Examples of probability, survival, and hazard functions for Weibull distribution.

Chapter 4

Evaluation

In this chapter, we will explain how we are going to evaluate our reordering framework. We will consider and define some metrics that enable us to measure the effectiveness of our reordering framework in a quantitative manner. These metrics also provide a ground for the reordering framework to be compared with other similar approaches. After defining and clarifying our metrics, we arrive at a solution for comparing different reordering methods.

4.1 Metrics

In this section, we will introduce, define, explain our criteria one by one. Before going through these criteria more in-depth, we would like to introduce them briefly. Table 4.1 presents our criteria in short.

Criterion	**Measure**	**Unit**
Ordering Coverage	Ratio of covered orderings to all orderings	%
Coverage Rate	Number of orderings per time	$\frac{1}{ms}$
Slowdown Overhead	Extra run time in compare to naive case	ms
Memory Overhead	Extra memory usage in compare to naive case	KB

Table 4.1: Summary of quantitative criteria for evaluating the reordering framework.

4.1.1 Ordering Coverage

The first and the most important metric in which we are interested is how many different message orderings we can enforce for a given component. The more message orderings we can enforce, the more thorough our testing will be, and the more software defects and faults we can hopefully reveal.

We measure this criterion by keeping track of and counting distinct message orderings among all possible message orderings. We then report the covered ratio of message ordering in percentages. We will formally define our *ordering coverage* metric as follows:

Ordering Criterion [1]
Ordering criterion, C, requires all message orderings possible between a certain number of components to be tested.

Test Requirements [1]
The set of our test requirements, TR, is consisted of all possible message orderings between a certain number of components to be tested.

Ordering Coverage Level [1]
Having defined our testing criteria and our test requirements set, the ordering coverage level is the ratio of the number of test requirements (message orderings) satisfied to the size of test requirements set.

Consider the example we have shown in Chapter 3. We have three sender components (A, B, and C) as well as a receiver component (R). We are only interested in message AR (a), BR (b), and CR (c). Our ordering criterion implies the following test requirements set. Using testing coverage criteria for analyzing the effectiveness of statechart-based testing techniques is done in [6].

$$TR = \{abc, acb, bac, bca, cab, cba\}$$

If we manage to only force four message orderings such as abc, acb, bca, and cba, our ordering coverage level will be calculated as follows:

$$Ordering\ coverage = \frac{4}{|TR|} = \frac{4}{6} = 66\%$$

Testing coverage criteria have been presented in many of the works related to testing software. Authors in [2] have investigated four data flow criteria and compared their coverage levels. In [36], authors presented a new testing coverage criteria for testing graphical user interfaces (GUIs). Various testing coverage criteria have been used in [33] to model testing effort and software reliability. In [17], testing coverage criteria have been used for validating software components.

4.1.2 Coverage Rate

It is also important how quickly we can cover the different message orderings. The faster we can try as many message orderings as possible, the faster we can catch software defects and errors. We may be able to enforce a large number of different message orderings in the long run. But, if we managed to do the same thing in a shorter amount of time, it would be more useful and practical. So, we can scale up our reordering framework for real big applications.

We measure this criterion as the number of different message orderings we have covered per unit of time. This measurement is done while we are simultaneously monitoring distinct message orderings. We just need to keep track of time as well. In the example we brought forth in the previous section, let us assume that we have covered four message ordering after $100ms$. The coverage rate would be calculated as follows:

$$Coverage\ rate = \frac{4}{100ms} = 0.04\ \frac{1}{ms}$$

Authors in [49] have used testing coverage level versus time to predict the reliability of software products in a quantitative manner. In another similar work [24], authors used testing coverage for describing the software reliability growth process. In [67], different coverage criteria have been measured within a certain amount of time.

4.1.3 Slowdown Overhead

Besides how quickly and effectively we can disclose software defects and faults in our application, it is also significant to know how much overhead our testing method incurs to applications compared to when we are running them without any reordering mechanisms in place.

We measured this criterion once by running our target applications without any testing mechanism implemented and once by running the same applications with our reordering framework in place. We then reported the difference in running time between these two scenarios. For instance, assume that the running time of a sample application without any reordering testing method is $100ms$. When we exert our reordering framework for testing purposes, running time of the same application is now $120ms$. We will report the slowdown overhead metric as follows:

$$Slowdown\ overhead = 120ms - 100ms = 20ms$$

This metric has been the first and the most important evaluation metric for many solutions and mechanisms which work during runtime. *FastTrack* [14], a race detection tool which instruments programs, uses the slowdown measure for evaluating their solution. *Pulse* [31], a dynamic mechanism for detecting deadlocks, uses this metric as well. Authors in [68] have used slowdown overhead for evaluating their kernel-level logging mechanisms. In [13], authors proposed a new way for detection of anomalies, and they have compared their solution with other similar works in terms of runtime overhead. Slowdown metric also was considered for comparing test suites using coverage criteria in [19].

4.1.4 Memory Overhead

Following up slowdown measurement, we are also interested in learning how much more memory the target applications need if a reordering method is in place. This criterion would become very important in embedded systems in which resources are limited and precious.

Similar to slowdown measurement, we ran the target application one time natively without any extra mechanism implemented and one time with our reordering framework in place. Then, we measured the memory usage in both cases and reported the difference. Assume that when we ran an application without any reordering testing methods, the application used up $600KB$ of memory, and when ran the same application using our reordering framework, it consumed $800KB$ of memory. We will then report the memory overhead metric as follows:

$$Memory\ overhead = 800KB - 600KB = 200KB$$

This metric also has been presented and used in a variety of works. Authors have used extra memory usage in [10] for evaluating their *mTags* framework for labelling processes

and tracking their interactions. In [11], authors have used memory use as one of their evaluation metrics for their labeling framework. Authors in [68] used the memory overhead to evaluate their kernel-level logging solution. In [13], authors have also compared their solution with other similar solutions in terms of extra required space.

4.2 Reordering Methods Comparison

In this section, we will investigate the advantages and disadvantages of each of the reordering methods we have presented in chapter 3 as part of our evaluation. In chapter 3, we portrayed three different approaches for reordering message-based communications in message-passing systems. These methods are called *blocking*, *buffering*, and *adaptive buffering*. For details about how each of these methods work, refer to section 3.2.1, section 3.2.2, and section 3.2.3.

4.2.1 Qualitative Comparison

In the qualitative comparison of reordering methods, we will discuss the pros and cons associated with each method, and we will postpone the quantitative comparison of these methods to next section. By qualitative comparison, we mean what a reordering method can do, and what it cannot do.

Blocking Reordering

In the blocking method, each message/event gets blocked until all its dependencies take place. Figure 3.2.1 illustrates this concept. One can argue what if one of the dependencies for a blocked message/event does not happen at all. In this case, the blocked message/event stays blocked forever, and the sender component starves! So, in the blocking method our processes of interest may face deadlock or starvation. All messages should be released at some point. This even becomes more critical in embedded systems in which we should guarantee an upper bound for message delivery. So, the question remains: why do we need the blocking method if it does not guarantee message delivery? If we know all messages of our interest occurs frequently for certain and there is no circular dependency between them, we can use the blocking method to guarantee enforcement of the exact ordering we seek. The blocking method can guarantee a certain order between messages of interest.

Reordering Method	Guarantee Ordering?	Guarantee Upper Bound?
Blocking	Fully	No
Buffering	Partially	Yes
Adaptive Buffering	Partially++	Yes

Table 4.2: Summary of qualitative comparison of reordering methods.

Buffering Reordering

In order to guarantee an upper bound for delivery of messages, we have introduced the buffering method. In the buffering method, each message/event gets buffered and get released when a timeout comes. Figure 3.2.2 depicts the concept of the buffering method for reordering messages. We have a set of timeouts which happen with a fixed period/frequency, and at each timeout, we release all buffered messages partially reordered. The buffering method solves the problem we had with blocking method. It guarantees an upper bound for messages. No message/event stays blocks forever, and there would be no deadlock. However, as we discussed earlier, an appropriate value for timeout period/frequency is a challenge and specific to running context of each application. The buffering method does not guarantee the specific ordering we are interested in. We may not have seen all messages of interest before a timeout. Figure 3.2.2 exemplifies this problem. Therefore, we need a dynamic mechanism for adjusting the timeout values.

Adaptive Buffering Reordering

In the adaptive buffering method we dynamically change the timeout to accommodate all messages of interest before the timeout, so we can reorder the messages fully rather than partially. By observing the pattern of message occurrences and adjusting the timeout period accordingly, we can hopefully enforce the exact ordering of interest between messages. As we explained in chapter 3 and section 3.4, we perform a survival analysis regression on-the-fly to determine the new timeout. The adaptive buffering method tries to enforce orderings of interest exactly as it is specified, and consequently, it lets us test more orderings. Like the basic buffering method, the adaptive buffering method prevents deadlocks from happening and guarantees delivery of messages after a certain amount of time. Yet, it does not guarantee that our ordering of interest will happen for certain. Table 4.2 summarizes the comparison of qualitative features of different reordering methods.

4.2.2 Quantitative Comparison

Our metrics defined in section 4.1 give us some measurements for comparing the reordering methods in a quantitative manner. For each reordering method, we measure *ordering coverage*, *coverage rate*, *slowdown overhead*, and *memory overhead* metrics. We expect the buffering method to have more overhead than the blocking method due to its timeout mechanism. We also expect the adaptive buffering method to have more overhead than the buffering method due to its survival analysis mechanism. We will bring the actual experimental results in chapter 6.

Chapter 5

Implementation

In this chapter, we review our reordering framework which implements our proposed methods for reordering messages in a message-passing system. We adopt the QNX Neutrino 6.5.0 as a message-passing system. In the context of QNX operating system our components are system processes.

We will review the architecture of our reordering framework, the user-space programming application interface, and the command-line utilities in a very general way. We do not detail our implementation due to the fact that the QNX source code is proprietary.

5.1 QNX Neutrino 6.5.0

The QNX Neutrino is a fully POSIX-compliant and Unix-based Real-Time Operating System (RTOS). Portable Operating System Interface (POSIX) is indeed a set of standard specifications defined by Institute of Electrical and Electronics Engineers (IEEE) [22] for promoting compatibility and portability for all operating systems. POSIX empowers third-party applications to be easily portable to any other POSIX-compliant operating system ideally only by recompiling the source code. POSIX documents are actively maintained and developed by IEEE Computer Society [16].

QNX is an instance of message-passing systems in which we have some processes communicating only through message-passes. In QNX inter-process communications are done via message-passing, even system calls are done using message-passing. The QNX Neutrino microkernel (*procnto*) itself is a very minimal process. It only contains a set of basic objects and fundamental functionalities like process and thread scheduling, memory management,

Software Tool	Version	Workstation
QNX Neutrino RTOS	6.5.0	Host/Target
Ubuntu 64-Bit	14.04 LTS	Development
QNX Software Development Platform	6.5.0	Development
VirtualBox	4.0.0 +	Development
Git (Version Control System)	1.9.1 +	Development

Table 5.1: The toolchain for development and deployment.

interrupt handling, and so forth. All other services such as device drivers are implemented as separate processes in user-space. QNX has been ported to a wide range of devices and architectures and can scale up very well thanks to its unique architecture. [57, 21].

5.2 Development Environment and Toolchain

For developing our framework and implementing our proposed methods we require two separate workstations as follows:

- Host/Target workstation
- Development workstation

The host or target workstation is set to run an instance of QNX Neutrino operating system. The kernel or more precisely the microkernel of this QNX instance has been modified and includes our framework implementation. Also, our framework library for application developers and our shell utility tools for testers have been added to user-space of this QNX instance. The development workstation is the main environment in which we carry out the actual implementation of our framework and compile our modified version of QNX Neutrino. Table 5.1 summarized the applications and tools we have used for implementing the reordering framework.

5.2.1 Development Operating System

For the development workstation the Ubuntu Linux distribution is preferred. Although other Linux distributions like rpm-based distribution will work well, they are not recom-

mended. Furthermore, the 32-bit version of Ubuntu is preferred, but 64-bit version also works fine. If you choose to use a 64-bit Ubuntu, you need to install the *"ia32-libs"* package for running the Momentics IDE. If you are not able to install this package using your default package manager, follow these steps:

```
sudo -i
cd /etc/apt/sources.list.d
echo "deb http://old-releases.ubuntu.com/ubuntu/ raring main
restricted universe multiverse" > ia32-libs-raring.list
apt-get update
apt-get install ia32-libs
```

5.2.2 Installing QNX SDP 6.5.0

The core of our toolchain for compiling our source code and developing QNX applications is *QNX Software Development Platform 6.5.0*. For acquiring this software package, go to the http://www.qnx.com/download/, navigate to "QNX Software Development Platform 6.5.x", and download "QNX Software Development Platform 6.5.0" for "Linux Hosts".

You need to follow instructions provided in a document named "QNX Software Development Platform Installation Guide [6.5.0]" [56]. It is recommended that you install the binary package using `sudo -E` rather than installing it as *root* user. It is also recommended that you install the package in the default path, emph/opt/qnx650/. The QNX SDP will install the necessary libraries, headers, and tools for developing QNX applications on your development workstation. It also includes an integrated development environment named *QNX Momentics IDE* which lets you easily develop, test, run, and debug your applications.

After installing the package, make sure that you have your environmental variables set in your *.bashrc* or *.zshrc* appropriately. These settings are certainly required if you want to work with your development workstation through an *ssh* session. Table 5.2 outlines the major environmental variables alongside their values.

5.2.3 Setup Host Workstation

For your host/target workstation, you have two options: you can either set up a real physical workstation or a virtual machine which runs an instance of QNX Neutrino 6.5.0. For running the host workstation as a virtual machine, you need to install the VirtualBox.

Variable	Value
QNX_CONFIGURATION	/etc/qnx
QNX_JAVAHOME	/opt/qnx650/_jvm
QNX_HOST	/opt/qnx650/host/linux/x86
QNX_TARGET	/opt/qnx650/target/qnx6
MAKEFLAGS	-I/opt/qnx650/target/qnx6/usr/include
LD_LIBRARY_PATH	/opt/qnx650/host/linux/x86/usr/lib
PATH	/etc/qnx/bin:/opt/qnx650/host/linux/x86/usr/bin:$PATH

Table 5.2: The specification of QNX SDP environmental variables.

To this end, visit https://www.virtualbox.org and follow the instruction for installation. Then, you can import a pre-configured QNX instance using the "Import Appliance ..." option. In your QNX instance *settings* section, set the network mode to *Bridged Adapter*. Then, make sure that your virtual instance gets a valid IP address, and you can *ping* it from another machine on your network.

It is highly recommended to allow *ssh* sessions from your host workstation to your development workstation and vice versa without entering a password using public-private key pairs. In your host workstation, which runs an instance of QNX Neutrino, you have to enable the ssh server by running it from its full path, `/usr/sbin/sshd`.

5.3 Reordering Framework Implementation

The reordering framework has been implemented as a kernel module in QNX Neutrino operating system and runs in kernel-space. The actual framework has been implemented in a separate file, and the control is transferred to it from the kernel module. User-space applications can access the kernel module and call its available operations indirectly using a static library provided.

There is also another important component which is the *service process*. This process implements some features of the reordering framework which cannot be run in kernel-space. Any kernel-space code has a limitation that no POSIX API function can be called. Figure 5.3 shows the components and the architecture of the reordering framework.

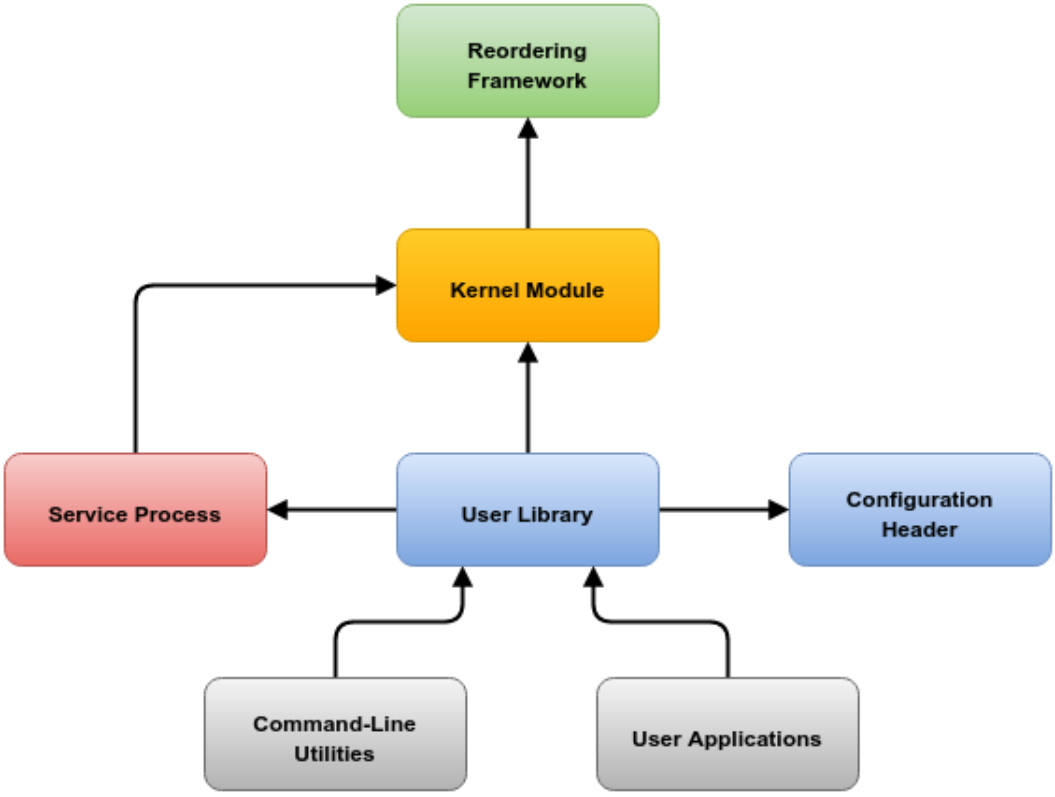

Figure 5.1: The reordering framework architecture.

5.3.1 Compiling Source Code

If you have established your development environment correctly, and your environmental variables are set to right values, compiling the source code for modified QNX Neutrino can be easily accomplished. You only need to run the "*build.sh*" script, and after about 20 to 25 minutes your source code will be completely compiled. You will find the final image under the "*qmu*" directory with "*ifs*" extension.

If you get a compiling error related to "*trunk/lib/io-pkt/utils/r/route*" sub-directory, delete that path completely and compile the source code again! Under the "*doc*" path, you will find a much more detailed document in Latex format which can be helpful in establishing proper working environment.

5.3.2 Using Reordering Framework

In order to make the reordering framework into effect, you need to copy the "*qnxtagging.ifs*" image under the "*qmu*" in your source directory to the "*/.boot*" directory in your host/-target workstation. You can do this using *scp* command. Then, you can restart your host machine, and you will see an entry for the new QNX booting image. If you boot up your QNX instance with that image, you will have the reordering framework running on your host workstation. Also, you will have access to reordering command-line utilities, examples, and experiments.

If you make any change to user-space header files, you need to update their corresponding files in host/target QNX instance as well. When you make such changes, you can update all files by running the "*update.sh*" script from "*script*" path in your source code directory. The "*load.sh*" script does the same by being run on a host/target machine.

For developing applications enabled with reordering features, you can either do it on your host or development workstations. When developing applications on a host/target machine, you simply need to include the "*libtag.h*" header file and link the "*/usr/lib/libtag.a*" library when compiling.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/neutrino.h>

#include <libtag.h>
#include <sys/tagging.h>


int main(int argc, char* argv[])
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

The above code snippet shows an example of C application which can make use of reordering Application Programming Interface (API) functions. You should compile this program as follows:

```
gcc example.c /usr/lib/libtag.a
```

You can also develop reordering-enabled QNX applications on your development machine and in QNX Momentics IDE. To this end, you have first to enable your QNX software development platform with the reordering framework libraries. Again, running the "*update.sh* script propagates all changes to all targets you need. Then, you have to add reordering library for building you project. Go to your project properties, select "*QNX C/C++ Project*", navigate to "*Linker*" tab, and select "*Extra libraries*" from the drop-down menu. Now, yo can add the library as "*tag*". Figure 5.3.2 pictures this step.

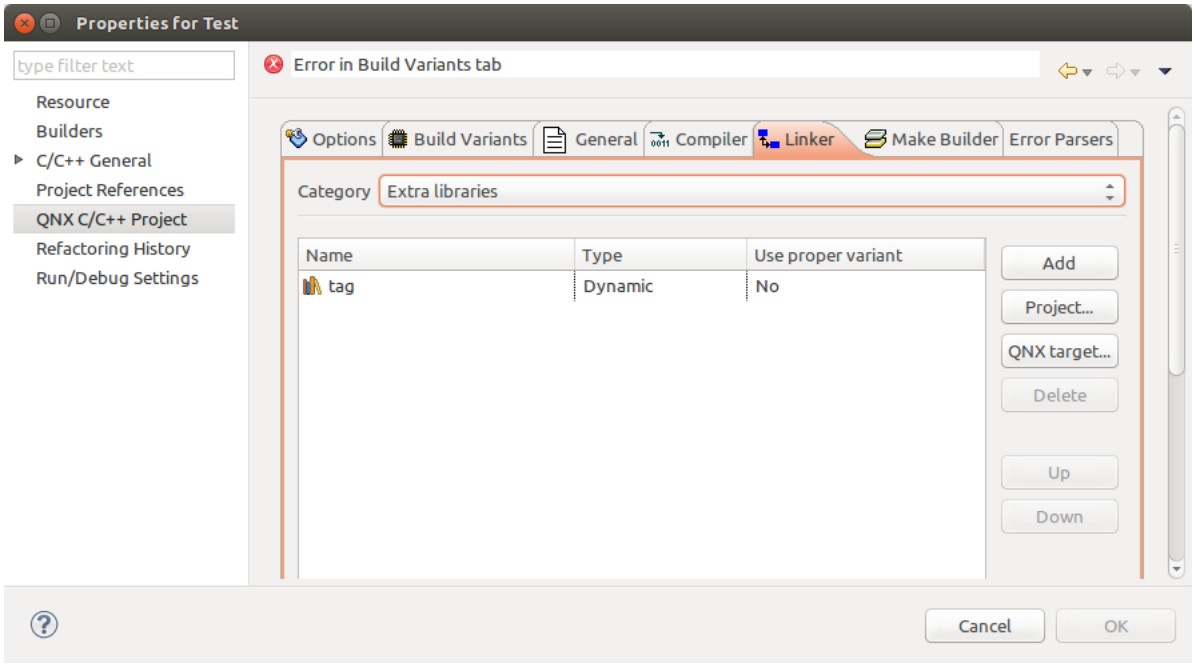

Figure 5.2: Adding the reordering library to a QNX Momentics project.

For directions regarding how to run and debug your application in Momentics IDE remotely on a host operating system, you can refer to another document named "10 Steps to Developing a QNX Program: Quickstart Guide [6.5.0]" [55].

5.4 Application Programming Interface

We have also provided a programming library for developers. Developers can use our reordering framework for testing purposes, enforcing message orders, or other intentions. Basically, they need to include one header file before compiling their source code with the corresponding library linked. We first introduce our programming library and the APIs and then, we would like to go briefly through another header file which specifies the configuration and structures for the reordering framework.

The reordering framework starts with tokenizing our processes of interest. Among all existing processes in the system, we can narrow down our attention just to a limited set of processes in which we are interested. There are limited number of tokens available for assigning to processes. Tokens can be assigned and freed.

```
int tag_reorder_available_tokens(char* tokens);
int tag_reorder_set_token(char token);
int tag_reorder_get_token(char* token);
int tag_reorder_free_token();
int tag_reorder_set_token_for(int pid, int tid, char token);
int tag_reorder_get_token_for(int pid, int tid, char* token);
int tag_reorder_free_token_for(int pid, int tid);
int tag_reorder_free_token_all();
```

The first function gives you a list of available tokens for assigning. The `tokens` should be an array of type `char` with size `TAG_MAX_TOKEN`. The next three functions are used for assigning a token to the calling process, getting the token of the calling process, and freeing the token of the calling process. The next three functions are used to assign a token to any arbitrary process, get the token of any arbitrary process, and free the token of any arbitrary process. The `token` argument in these six functions is a single `char` as token. The combination of `pid` and `tid` arguments are actually specifying the process of interest. Moreover, finally, the last function frees all assigned tokens.

Having tokenized the processes of interest, the next step is registering the events of interest. An event is a communication between two processes. Each event is specified by a source process (sender) and a destination process (receiver). An event could occur either in form of a *message-passing* (synchronous) or in form of a *pulse* (asynchronous).

```
int tag_reorder_register_event(char token_src, char token_dest, int* id);
```

```
int tag_reorder_reset_event(int id);
int tag_reorder_reset_event_all();
int tag_reorder_unregister_event(int id);
int tag_reorder_unregister_event_all();
```

The first function in this set registers an event. An event is specified by tokens of source and destination processes. The `id` is an `int` variable which will have the id of the registered event after returning from the function. The next two functions are used for resetting the current state of an event and the state of all registered events, respectively. The current state of an event says if the event has occurred and, if yes, when. Finally, the last two functions void registration of an event or all of the registered events.

```
int tag_reorder_lookup_event(char token_src, char token_dest, int* id);
int tag_reorder_get_event(int id, TAG_REORDER_EVENT* event);
int tag_reorder_get_event_all(TAG_REORDER_EVENT* events);
int tag_reorder_get_event_state(int id, TAG_REORDER_EVENT_STATE* state);
int tag_reorder_get_event_state_all(TAG_REORDER_EVENT_STATE* states);
```

The first function in this set looks up the id of an event by specifying the tokens of its source and destination processes. The next function retrieves the specification of an event by its id. The `event` is a single instance of the structure `TAG_REORDER_EVENT`. The third function in this set retrieves the specification of all registered events. The `events` is an array of the structure `TAG_REORDER_EVENT` with size of `TAG_MAX_EVENT`. The last two functions provide the current state of an event or the current state of all events. The `state` is a single instance of structure `TAG_REORDER_EVENT_STATE` and the `states` is an array of structure `TAG_REORDER_EVENT_STATE` with a size of `TAG_MAX_EVENT`.

Every event can have an associated *notification*. A notification is a pulse from the *tagging* process to a desired process on a desired channel with a desired code. Using this mechanism you have the option of receiving notifications whenever your events of interest occur.

```
int tag_reorder_add_event_notif(int id, int pid, int chid, int code);
int tag_reorder_get_event_notif(int id, TAG_REORDER_EVENT_NOTIF* notif);
int tag_reorder_get_event_notif_all(TAG_REORDER_EVENT_NOTIF* notifs);
int tag_reorder_remove_event_notif(int id);
int tag_reorder_remove_event_notif_all();
```

The first function adds a notification for an already registered event. The `id` specifies the event of interest, the `pid` and `chid` specify a process and a channel that the process is waiting on for receiving pulses, and the `code` specifies the code of pulse to be sent for notifying. The next two functions retrieve the specification of notification for an event or notifications for all events. The `notif` is a single instance of the structure `TAG_REORDER_EVENT_NOTIF`, and the `notifs` is an array of the structure `TAG_REORDER_EVENT_NOTIF` with size of `TAG_MAX_EVENT`. The last two functions remove associated notifications for an event or notifications for all events. Figure 5.4 shows an example in which we have assigned some tokens to our processes of interest, registered those message we are interested in as events, and added a notification for one of the events to be sent to another process on a specified channel.

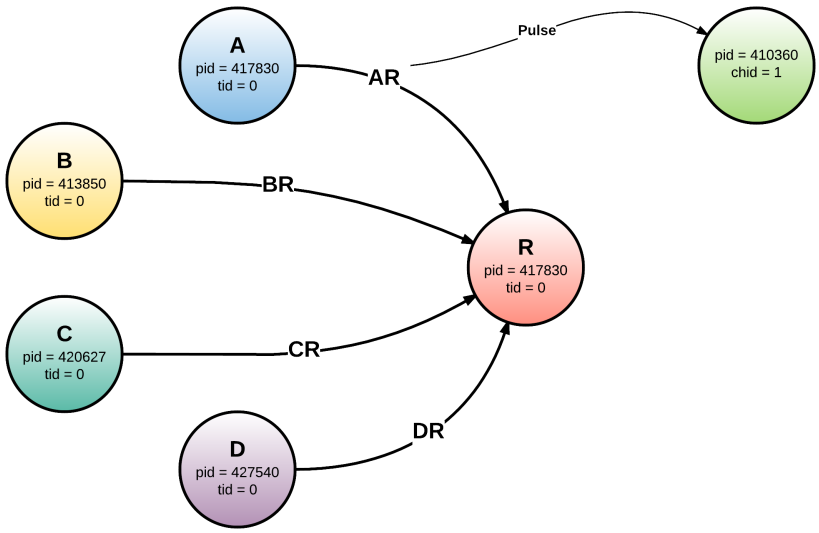

Figure 5.3: An example of tokenized processes, registered events, and added notifications.

Having registered the events of interest, the next step is specifying constraints for events. A dependency is a form of constraint for events. The occurrence of an event could be dependent on the occurrence of another event ahead. If AB is dependent on AC, it implies that the communication from process A to process C should happen before the communication from process A to process B.

```
int tag_reorder_add_constraint(int id_dependent, int id_dependency);
int tag_reorder_remove_constraint(int id_dependent, int id_dependency);
int tag_reorder_clear_constraints(int id);
```

```
int tag_reorder_clear_constraints_all();
```

The first function in this set adds a constraint for an event in the form of a dependency. The event with id `id_dependent` will be dependent on the event with id `id_dependency`. The next function removes a constraint in the form of a dependency. Moreover, the last two functions clear all dependencies of an event or all dependencies existing in the system.

```
int tag_reorder_get_dependents(int id, TAG_REORDER_EVENT* dependents);
int tag_reorder_get_dependencies(int id, TAG_REORDER_EVENT*
    dependencies);
int tag_reorder_get_dependencies_str(int pid, int tid, char*
    dependencies);
```

The first function retrieves all events which are dependent on an event. The `id` specifies the event that we want to make all events dependent on. The `dependents` is an array of the structure `TAG_REORDER_EVENT` with size of `TAG_MAX_CONSTRAINT`. The next function similarly retrieves all events which an event is dependent on it. The `id` specifies an event that we want to get all its dependency events. The `dependencies` is an array of the structure `TAG_REORDER_EVENT` with size of `TAG_MAX_CONSTRAINT`.

If an event has unsatisfied constraints (dependencies), the source process will yield the processor, and its state is changed to *TAG_WAIT* until all unsatisfied events get satisfied at some point. The last function in the above set gives you all unsatisfied events for a process to continue at any time. The `pid` and `tid` combination are actually specifying a process as source of an event, and the `dependencies` argument is an array of type `char`.

As we have explained earlier in this chapter, one of the the reordering framework component is a process named *tagging*. This process takes care of some of the tasks that cannot be done in kernel space.

```
int tag_reorder_service_set(int pid, int chid);
int tag_reorder_service_get(int* pid, int* chid);
```

The *tagging* process can set and save its address for other processes using the first function in the above set. The `pid` is the process id of the tagging process, and the `chid` is the channel id which the tagging process is waiting on for receipt of commands. The

second function in the set lets the user-space applications to retrieve the address of the tagging process. Developers do not need to deal with these functions at all.

```
int tag_reorder_release_event(int id);
int tag_reorder_release_dependents(int id);
int tag_reorder_schedule_release_events();
```

Here are more functions that developers typically find futile. The first function allows an event to occur regardless of its unsatisfied constraints (dependencies). The second function let all the events that are dependent on an event to occur. The `id` specifies the event that we want to release the events dependent on it. Finally, the last function releases all the events which are currently held due to unsatisfied events. This function breaks the deadlock case, and it does the topological sort for the dependency graph of events. So, in the case of deadlock, all the events will occur, and as many as constraints (dependencies) possible get satisfied.

```
int tag_reorder_enable_timeout(int timeout_ms);
int tag_reorder_enable_adaptive_timeout(int initial_ms);
int tag_reorder_disable_timeout();
```

The first function in the last set of functions enables the *buffering* method. The `timeout_ms` is the fixed timeout for the buffering method in milliseconds. Every *timeout_ms* milliseconds, all on-hold events get occurred to prevent deadlocks. The second method enables the *adaptive buffering* method. The `initial_ms` is an initial timeout in milliseconds for the adaptive buffering method. This timeout will be adapted to the context of running processes dynamically.

When you include the "*libtag.h*" as a developer, another header file, "*tagging.h*", is included as well. This file contains the configuration and definitions needed for the reordering framework. If you look at the "tagging.h" file, you will find the following definitions and decelerations. These definitions comes in three different forms:

- Configuration macros
- Input and output macros
- Structures and data types

```
#define   TAG_MAX_TOKEN             256
#define   TAG_MAX_EVENT             512
#define   TAG_MAX_CONSTRAINT         16
#define   TAG_MAX_ADAPT_DATA        512
```

These macros determine the configuration (length) of circular buffers needed for the reordering framework. The first one says how many different tokens are available for assigning to processes/threads. The second macro defines the maximum number of events we can register within the reordering framework context. The third macro similarly specifies the maximum number of constraints that we can place on registered events. Finally, the last macro determines the size of the buffer for data needed for adaptive analysis in the reordering framework.

```
#define   TAG_NULL                   0
#define   TAG_FALSE                  0
#define   TAG_TRUE                   1
```

These three macros are general-purpose macros for null, false, and true values respectively. You can pass them as input parameters or accept them as output parameters when calling the reordering API functions.

```
#define   TAG_REORDER_NO_EVENT      -1
#define   TAG_REORDER_NO_PROCESS    -1
#define   TAG_REORDER_NO_CHANNEL    -1
#define   TAG_REORDER_PULSE          1
#define   TAG_REORDER_MESSAGE        2
```

These are reordering-specific macros which are self-descriptive. The first three macros respectively say if there is no event, no process, or no channel. The last two macros say if an event is a pulse or a message.

```
#define   TAG_RET_YES                1
#define   TAG_RET_OK                 0
#define   TAG_RET_NO                -1
#define   TAG_RET_ERROR             -10
#define   TAG_RET_NO_THREAD         -23
```

```
#define  TAG_RET_NO_PROCESS       -24
#define  TAG_RET_NO_CONNECT       -25
#define  TAG_RET_NO_CHANNEL       -26
#define  TAG_RET_NO_TOKEN         -60
#define  TAG_RET_TOKEN_TAKEN      -61
#define  TAG_RET_TOKEN_EXISTS     -62
#define  TAG_RET_NO_EVENT         -63
#define  TAG_RET_EVENT_EXISTS     -64
#define  TAG_RET_NO_NOTIF         -68
#define  TAG_RET_CYCLIC_CONST     -69
#define  TAG_RET_TIME_ERROR       -70
#define  TAG_RET_TIMER_ERROR      -71
#define  TAG_RET_INVALID_TOKEN    -82
#define  TAG_RET_INVALID_EVENT    -83
#define  TAG_RET_INVALID_CONST    -84
#define  TAG_RET_INVALID_SERVC    -85
```

These macros are reordering-specific macros for returning from reordering API functions. They are pretty straightforward. For example, *TAG_RET_CYCLIC_CONST* says adding the specified constraint will cause a cyclic dependency.

```
typedef struct tag_time
{
    long  sec;
    long  nsec;
} TAG_TIME;
```

This structure specifies a timestamp with nanoseconds precision.

```
typedef struct tag_reorder_event
{
    char  token_src;
    char  token_dest;
} TAG_REORDER_EVENT;
```

This structure defines an event within the context of the reordering framework. Each event is determined by a source and a destination. Source and destination can be any process/thread with a token.

```
typedef struct tag_reorder_event_state
{
    int       occurred;
    TAG_TIME  time;
} TAG_REORDER_EVENT_STATE;
```

This structure characterizes the last state of an event. It determines if an event has ever occurred and, if yes, when.

```
typedef struct tag_reorder_event_data
{
    int  src;
    int  dest;
    int  time;
} TAG_REORDER_EVENT_DATA;
```

This structure encapsulates event data needed for adaptive reordering analysis. It has the specification of an event (source and destination) besides the last timestamp that the event has occurred at.

```
typedef struct tag_reorder_event_notif
{
    int  pid;
    int  chid;
    int  code;
} TAG_REORDER_EVENT_NOTIF;
```

Every event may have an associated notification as well. This structure defines a notification for an event. A notification is simply an asynchronous pulse to a desired process (*pid*) on a desired channel (*chid*) with a desired code (*code*).

5.5 Runtime Support

In addition to the programming application interface for developers, we have also provided a command-line tool for system administers and application testers. The command for using this tool is called *reorder*. All forthcoming commands and options are case-sensitive.

```
reorder help
reorder config
```

The *help* option prints the list of all commands and options alongside a brief description for each. The *config* option prints the full configuration of the reordering framework. It gives you a list of available macros and structures that you can use for programming purposes. You do not need to remember any values; you only need to remember names.

```
reorder token list
reorder token set  <PID> <TID> <TOKEN>
reorder token get  <PID> <TID>
reorder token free <PID> <TID>
reorder token free all
```

These commands which start with *token* options manage tokens in the reordering framework. The *list* option shows all the tokens available for assigning at any time. Currently, all possible tokens are capital letters of English alphabet (A-Z). The *set* option assigns a token to a process/thread. For this purpose you should refer to a process/thread by its *pid* and *tid*, and then specify a token for assigning to that process/thread. The *get* option shows the token assigned to a process/thread. You can again refer to a process/thread by its *pid* and *tid*. The *free* option release the token of a process and frees that token for assigning to other processes/threads. You specify a process/thread by its *pid* and *tid*, and its token gets freed. Finally, the *free all* option releases and frees all assigned tokens.

```
reorder event reg   <EVENT>
reorder event reset <EVENT>
reorder event reset all
reorder event unreg <EVENT>
reorder event unreg all
```

In the context of the reordering framework, an event is defined as a message from a source process to a destination process. Among all process exist in the operating system, we pin down our attentions only to a limited set of processes. We do this by assigning tokens to those processes of interest; we refer to each process by its token. Similarly, among all possible interactions between tokenized processes, we focus our attention on events of interest. An event is specified using its source and destination tokens, and each registered

event has a unique id number.

The *reg* option registers a new event in the reordering framework. You specify an event using its source and destination tokens. For example, *AB* implies a message from the process with token A to the process with token B. The *reset* option resets the last state of a given event. A state event comprises if the event has recently occurred and if yes, the timestamp of the last occurrence. The *reset all* option resets the state of all registered events. The *unreg* option unregisters a previously registered event, and the *unreg all* option unregisters all registered events.

```
reorder event list
reorder event get <ID>
reorder event lookup <EVENT>
```

The *list* option shows all the registered events in the reordering framework. The *get* option shows the specification of an event specified by its id number, and the *lookup* options shows the specification of an event specified by its source and destination tokens.

```
reorder notif add <EVENT> <PID> <CHID>
reorder notif get <EVENT>
reorder notif list
reorder notif remove <EVENT>
reorder notif remove all
```

Within the context of the reordering framework, each registered event can have an associated notification. A notification for a given event is consist of a *process id*, a *channel id*, and a *code*. When that event happens, an asynchronous pulse will be sent to the specified process, on the specified channel, and with the specified code.

The commands start with the *notif* option manage event notifications. The *add* option adds a notification for an event. It is necessary to specify the events using their tokens and the process and the channel through which the notification must be sent. The *get* option shows the notification for an event if applicable. The *list* option shows all added notifications in the reordering framework. The *remove* option removes the notification for an event; the *remove all* option removes all added notifications.

```
reorder const add <EVENT> <EVENT>
```

```
reorder const remove <EVENT> <EVENT>
reorder const clear  <EVENT>
reorder const clear all
reorder const list <EVENT>
```

Another important notion in the reordering framework is *constrain*. After tokenizing processes and registering events, we can define constrains on event occurrences. Currently, there is only one type of constraint which is the *dependency* constraint. We can potentially have other types of constrains such as the *delay* constraint. A constraint makes occurrence of an event conditional. Consequently, an event does not occur until all its constraints are satisfied.

These commands, which start with the *const* option, manipulate constraints in the reordering framework. The *add* options defines a new dependency constraint between two events. The first event is dependent event and the second one is dependency event. For instance, *reorder const add AB AC* implies that the event AB is dependent on the event AC which means that the event AC should happen before the event AB. The *remove* option removes a dependency constraint between two events. The *clear* option deletes all dependency constraints for a given event, so the event become unconditional. You may guess that the *clear all* option deletes all defined constraints in the reordering framework.

```
reorder release <EVENT>
reorder release depens <EVENT>
reorder release all
```

These commands, which start with the *release* option, are for administrative and testing purposes. An event with unsatisfied constraints will be *onhold* until all its constraints are satisfied. Using the first command we can force an event to happen regardless of its constraints. The first command does this. The *depens* options releases all events which are dependent on a given event. The *release all* option releases all events on hold.

```
reorder timeout enable <TIMEOUT>
reorder timeout adapt  <INITIAL>
reorder timeout disable
```

As you may recall from previous chapters, one of the reordering features is buffering and adaptive buffering methods for reordering messages. The *timeout enable* option enables

buffering method with a fixed timeout in milliseconds. The *timeout adapt* enables *adaptive buffering* with an initial timeout in milliseconds. Finally, the *timeout disable* option disables any buffering method.

```
reorder log
reorder log clear
```

These commands are for working with logging features of the reordering framework. This feature is for debugging the reordering framework source code, and it may not be available in final or production version of the reordering framework.

Chapter 6

Experiments and Case Studies

In this chapter, we introduce and explain our experiments and then our case studies in which we apply our different reordering methods. We then demonstrate our results and present them using diagrams and tables. Finally, we deliver the outcome of measuring our quantitative metrics presented in Chapter 4.

6.1 Experiments

6.1.1 Experiment Setup

As we have implemented our reordering framework on the QNX operating system, our experiments are designed and set to be run in QNX as well. Our message-passing system is the QNX operating system, and our components are system processes in the operating system. In the QNX operating system, inter-process communications are done using message-passing. Messages can be sent either synchronous or asynchronous.

Figure 6.1.1 shows how our experiments have been set up. We have one receiver process R and ten other sender processes A to J. Each sender process sends messages to receiver process R. More specifically, each sender process has a loop that requires a specified wait time before it is able to send a message to process R. The amount of time each process waits is calculated as follows:

$$delay = base + random$$

The component *base* is a constant number which has been set to $100ms$ in our experiments. The component *random* is a random number in a pre-specified range. We ran all our experiments once with *random* in $[0, 2]$ range and once with *random* in $[0, 5]$. Also, we feed our random number generator with the same seeds each time we run the experiment. A fixed seed ensures that our sender processes behave deterministically each time we run them, so our results are consistent and comparable.

Tokens A to J have been assigned to sender processes, and token R has been assigned to receiver process. Each message from a sender process to a receiver process is registered as an event. Our experiment driver defines dependencies between these events on-the-fly in such a way to generate all possible orderings between messages. So, the sequence of orderings we are going to test will be similar to the following:

$$abcdefghij$$
$$abcdefghji$$
$$abcdefgihj$$
$$abcdefgijh$$
$$abcdefgjhi$$
$$abcdefgjih$$
$$\ldots$$

Our experiment driver also keeps track of all message orderings seen by R. Basically, each time the process R sees an ordering of messages from sender processes, the experiment driver logs and writes that ordering in a data file. After our experiment is completed, we will process the log files for distinct orderings. The timestamp at which an ordering completes is also recorded.

6.1.2 Experiment Results

Figures 6.3 and 6.4 show the results of experiments we have conducted. Both of these experiments have been carried out on a physical (non-virtual) QNX machine. You can find the hardware and software specification of this machine in Appendix A.

In each diagram we have brought six different experiments. We ran the experiment one time without any reordering method in place, one time using the blocking method,

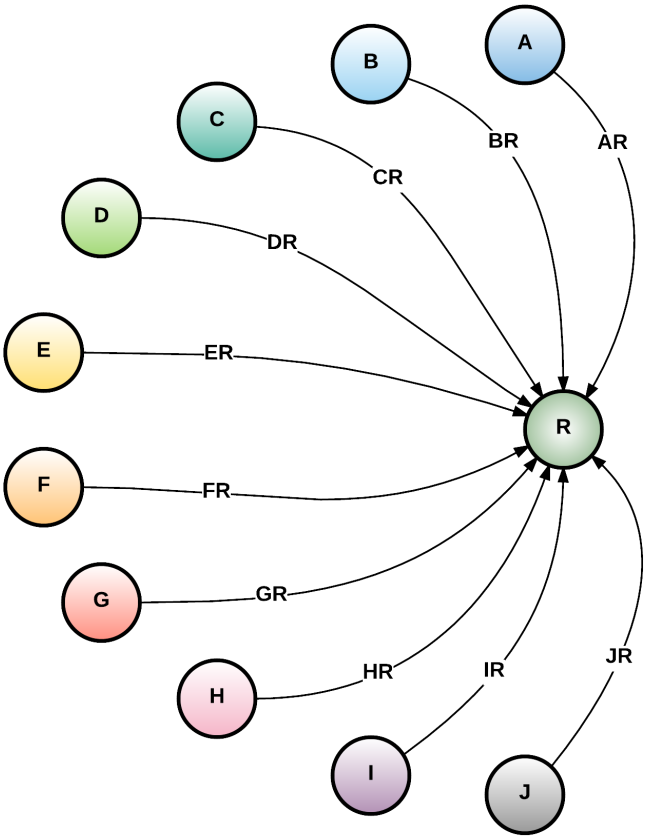

Figure 6.1: The configuration of reordering experiments.

three times using the buffering method, and one time using the adaptive buffering method. In the first diagram, all experiments have been done with *base* delay equal to $100ms$ and a *random* delay in $[0, 2ms]$ range. In the second diagram, all experiments have been done with *base* delay equal to $100ms$ and a *random* delay in $[0, 25ms]$ range. Table 6.1 summarizes all the experiment configurations. Each column is a configurable parameter for an experiment. Each row shows a reordering method used for experimenting. Each cell within each row shows the different options for that parameter. There are 24 experiments in total, and the next two figures visualize the results of these experiments. You can find the completed details of our reordering experiments in Appendix B.

You may ask why we chose the aforementioned configuration parameters. The *base* delay does not make a difference in the final results and conclusion because in the long

Reordering	Base Delay	Random Delay	Timeout (ms)	Host
Random (Naive)	100	[0,2], [0,5]	N/A	Physical, Virtual
Blocking	100	[0,2], [0,5]	N/A	Physical, Virtual
Buffering	100	[0,2], [0,5]	50, 100, 200	Physical, Virtual
Adaptive Buffering	100	[0,2], [0,5]	200	Physical, Virtual

Table 6.1: Different configurations of all reordering experiments.

run each process has minimum delay regardless of how fast or how slow it sends its messages. What matters is the *variance* of each process delay. The *random* delay component causes this variance. If you notice, our random delay values are small in comparison to our base delay. This assumption is valid in fact since processes do not expose very stochastic behaviors with high variance. In fact, since processes mainly follow a deterministic behavior, we introduced our reordering framework to for testing them. Otherwise, if following a completely random behavior, we would not require any reordering method for testing purposes.

The only difference between experiments in Figure 6.3 and results in Figure 6.4 is that in the former, the *delay* component is set to $2ms$ and in the latter, the *delay* is set to $5ms$. The *Random* experiment is the naive case in which we do not apply any reordering methods. So, the orderings are completely random. Then, we use the blocking method. The buffering method is employed three times with three slightly different configurations for fixed timeouts. Since adaptive buffering method adjusts its timeout dynamically, the initial choice for timeout does not matter, and we used this method only once.

As you see in the result figures, the blocking method has the best performance, which is no surprise given the blocking method systematically applies all orderings possible and guarantees that each ordering. For the buffering method we see that if we increase the fixed timeout value, the coverage will slightly increase. Finally, if the processes expose a more random behavior with a higher variance, the performance of the *random (naive)* experiment will obviously increase.

We repeated our reordering experiments using only seven sender processes (A to G). Figure 6.5 shows the corresponding results. Using seven processes, the total number of message orderings would be $7! = 5040$. The random/naive experiment hits 1232 distinct orderings (24% coverage) after 30 minutes whereas the blocking method hits 100% coverage, and buffering methods (including adaptive buffering) hit 99% coverage.

Experiment	Base Delay	Random Delay	Timeout	Run time	Orderings
Random (Naive)	100ms	2ms	N/A	3600s	11835
Random (Naive)	100ms	2ms	N/A	3600s	12164
Random (Naive)	100ms	2ms	N/A	3600s	12007
Random (Naive)	100ms	2ms	N/A	3600s	11947
Random (Naive)	100ms	2ms	N/A	3600s	12159
Random (Naive)	100ms	2ms	N/A	3600s	11731
Random (Naive)	100ms	2ms	N/A	3600s	12313
Random (Naive)	100ms	2ms	N/A	3600s	11192
Random (Naive)	100ms	2ms	N/A	3600s	12262
Random (Naive)	100ms	2ms	N/A	3600s	11672
Buffering	100ms	2ms	100ms	3600s	34214
Buffering	100ms	2ms	100ms	3600s	34208
Buffering	100ms	2ms	100ms	3600s	34220
Buffering	100ms	2ms	100ms	3600s	34208
Buffering	100ms	2ms	100ms	3600s	34212
Buffering	100ms	2ms	100ms	3600s	34206

Table 6.2: The results of running reordering experiments multiple times.

Data Replication

One more thing we could do to show that our results and thus our conclusions are consistent is replicating experiments. For example, we could have run each experiment (random/-naive, blocking, buffering, and adaptive buffering) ten times and include all of them in the diagrams. However, our experimental results show that if we run our experiments many times, they do not expose a different behavior; we almost get the same results every time as expected.

Table 6.2 summarizes the results of replicating random/naive and buffering experiments. We ran the random/naive experiment ten time and the buffering experiment six times.

We can quantify the variation in this data set using the statistics presented in Table

Statistic	Mean	Median	Variance	Standard Deviation
Random (Naive)	5888.2	5971	137276.6	370.5086
Buffering	34211.33	34210	26.66667	5.163978

Table 6.3: The calculated statistics for replicated reordering experiments.

Experiment	Coverage	Coverage Rate	Slowdown	Memory Overhead
Random (Naive)	24.42%	0.684 s^{-1}	N/A	N/A
Blocking	100%	2.800 s^{-1}	0.29%	*constant*
Buffering (50ms)	99.64%	2.790 s^{-1}	0.29%	*constant*
Buffering (100ms)	99.98%	2.799 s^{-1}	0.29%	*constant*
Buffering (200ms)	100%	2.800 s^{-1}	0.29%	*constant*
Adaptive Buffering	99.72%	2.792 s^{-1}	0.28%	*constant*

Table 6.4: The measured metrics for evaluating reordering methods quantitatively.

6.3. As it can be seen in this table, the *mean* and *median* values are very close, especially for the buffering experiment. Also, the *variance* and *standard deviation* values are enough small in compare to corresponding ordering values in Table 6.2. As you may notice, the variation in data related to the buffering experiments is very negligible.

6.1.3 Evaluation Metric Measurements

Table 6.4 shows the result of measuring evaluation metrics presented in Chapter 4. For each experiment, we have measured four evaluation metrics. As you can see, the coverage of our reordering methods is highly close to full coverage. More interestingly, the slowdown overhead of our reordering methods is very negligible. The memory overhead is also constant which means that regardless of the input size of a test case, the extra memory needed is always the same. This is due to static memory allocation for our reordering framework.

The first two metrics, ordering coverage and coverage rate, are directly derived from our experiment results depicted in figures 6.3 and 6.4. The details of slowdown measurements are provided in Appendix C.

6.2 Case Studies

6.2.1 Case Study Setup

Our case studies design is similar to our experiments. We have monitored all message-passing communications between all processes in QNX operating system for a certain period. We discovered that frequent message-passing happens between processes of *Photon* utility. Photon is a UI shell for QNX that enables a user to work with QNX in a graphical environment. It is similar to GNOME, but very minimalistic. It is composed of a number of processes, each responsible for one specific aspect. The main process is *usr/photon/bin/Photon*, and other Photon processes send messages to this process intermittently.

In our case study we have selected five Photon sender processes and assigned tokens A to F to these processes. We also assign token R to the Photon main process. Next, we register messages from sender processes to receiver processes. We also add notifications for these events to alert us whenever one of these events occurs. These notifications enable us to track events and the order in which they occur. Likewise, the case study driver keeps track of orderings between these messages and records the ordering alongside the timestamps of occurrence. Figure 6.2.1 demonstrates this setup for our case study. The only subtle point is that for some reason that we later on explain, we excluded the process */usr/photon/bin/shelf* from our case studies when running.

We would like to mention that the blocking method does not work for our case study (Photon) since there are some underlying dependencies between message which were explicitly coded. Therefore, we are not able to force any ordering of interest. Instead, we use our buffering method which is designed to deal with these cases.

We start running our case study and then we work with Photon UI as usual. We do some interactions like typing in the terminal, clicking on the UI, and browsing the system. We do our best to repeat these actions each time we run the case study driver. For the purposes of our case study, we would like to show that our new methodology for testing works and we can apply it to real applications in use.

6.2.2 Case Study Results

Figures 6.6 and 6.7 illustrate the outcome of our case studies. We once ran the case study without any reordering method in place; we just watched the orderings seen by Photon process R (black dots). We then repeated the case study with the buffering method in

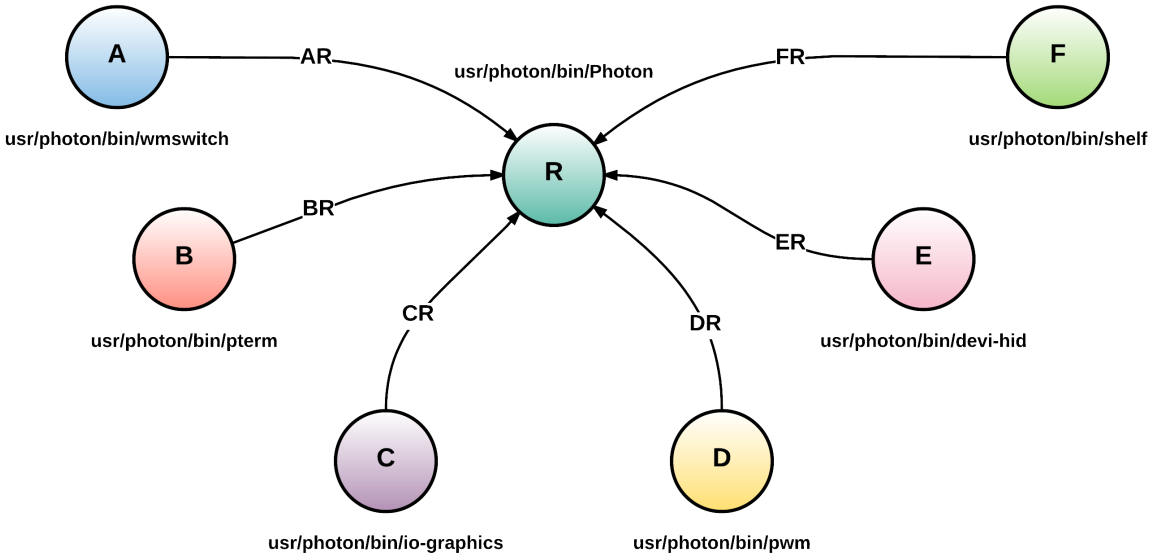

Figure 6.2: The configuration of reordering case studies.

effect. We have tried the buffering method with three different fixed timeouts. As the results implicate, this timeout value does not significantly impact the final coverage. A short timeout is recommended to decrease the overall overhead.

6.2.3 A Potential Bug in Photon

Using our reordering framework, we might have found a bug in Photon utility. If we include the Photon process */usr/photon/bin/shelf* in our case studies, this processes stops responding after seeing a few orderings. Remember that our reordering framework does not override any explicit dependency between messages coded in the processes; it only forces a different ordering if it is possible for the operating system to allow. Consequently, we believe this could be a software defect in the Photon application. It could be due to the fact that one of the processes may assume that it always receives a particular message after receiving a message from another process without explicitly enforcing this order.

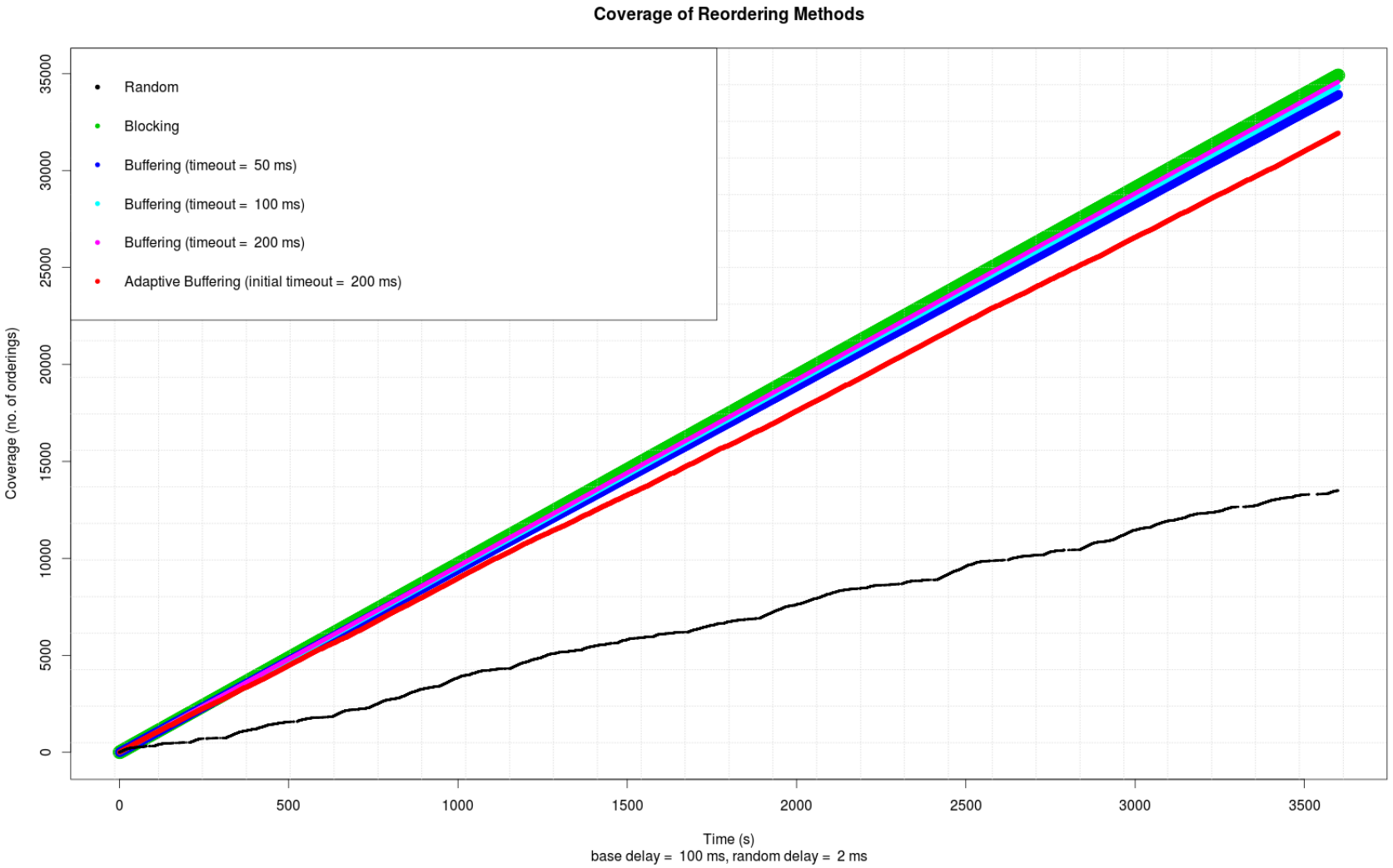

Figure 6.3: The results of running 6 reordering experiments for 10 processess on physical QNX machine.

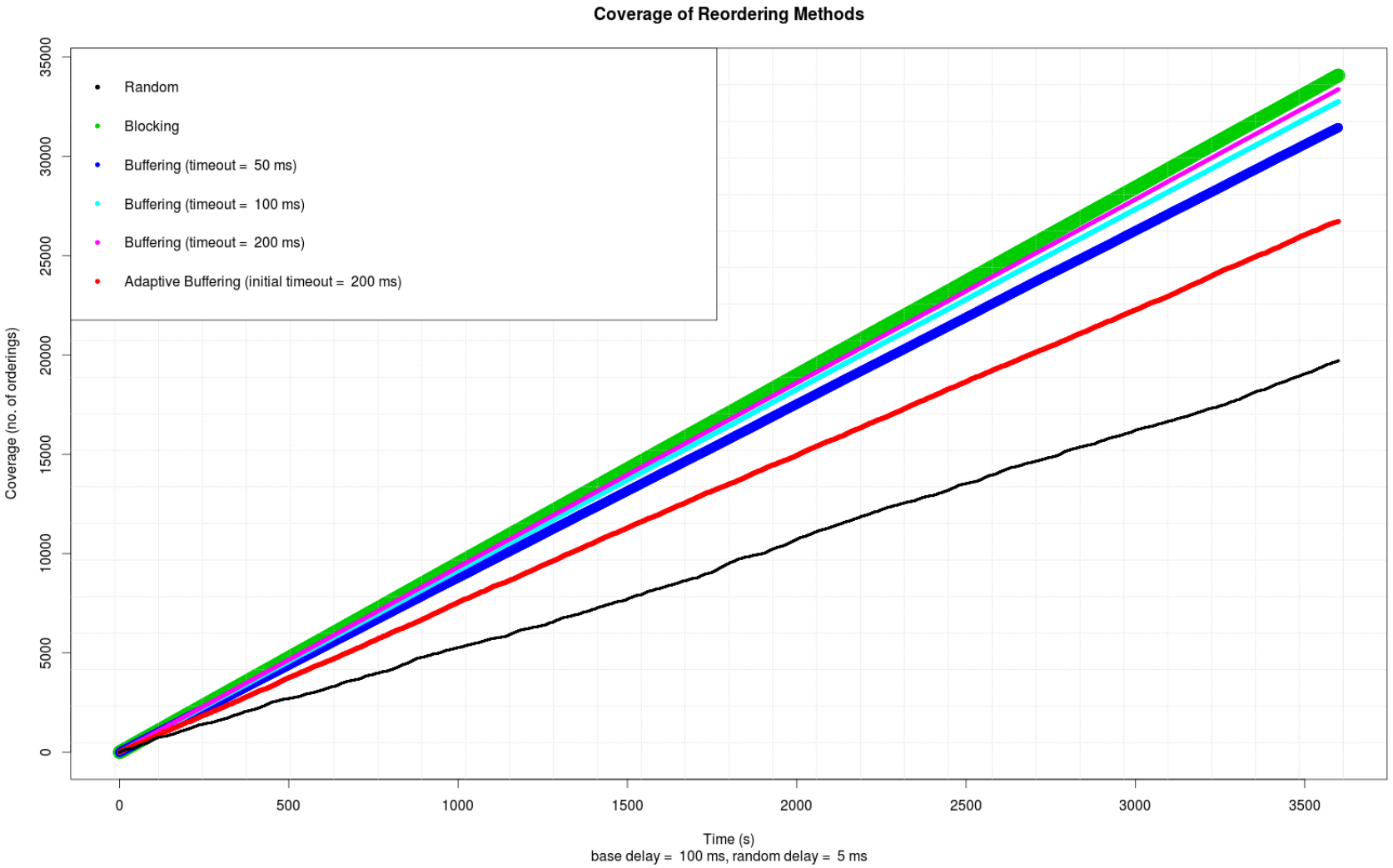

Figure 6.4: The results of running 6 reordering experiments for 10 processes on physical QNX machine.

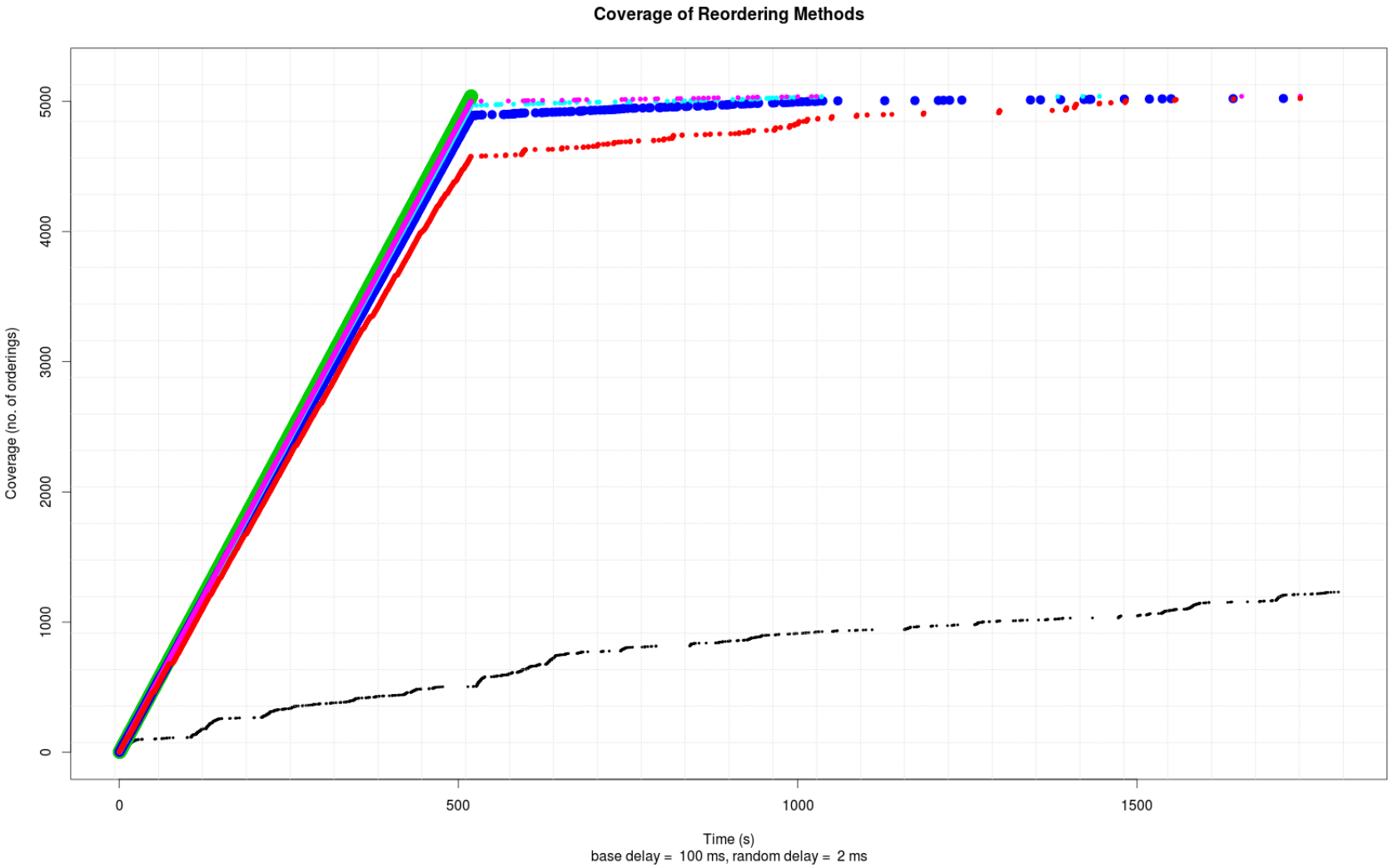

Figure 6.5: The results of running 6 reordering experiments for 7 processes on virtual QNX machine.

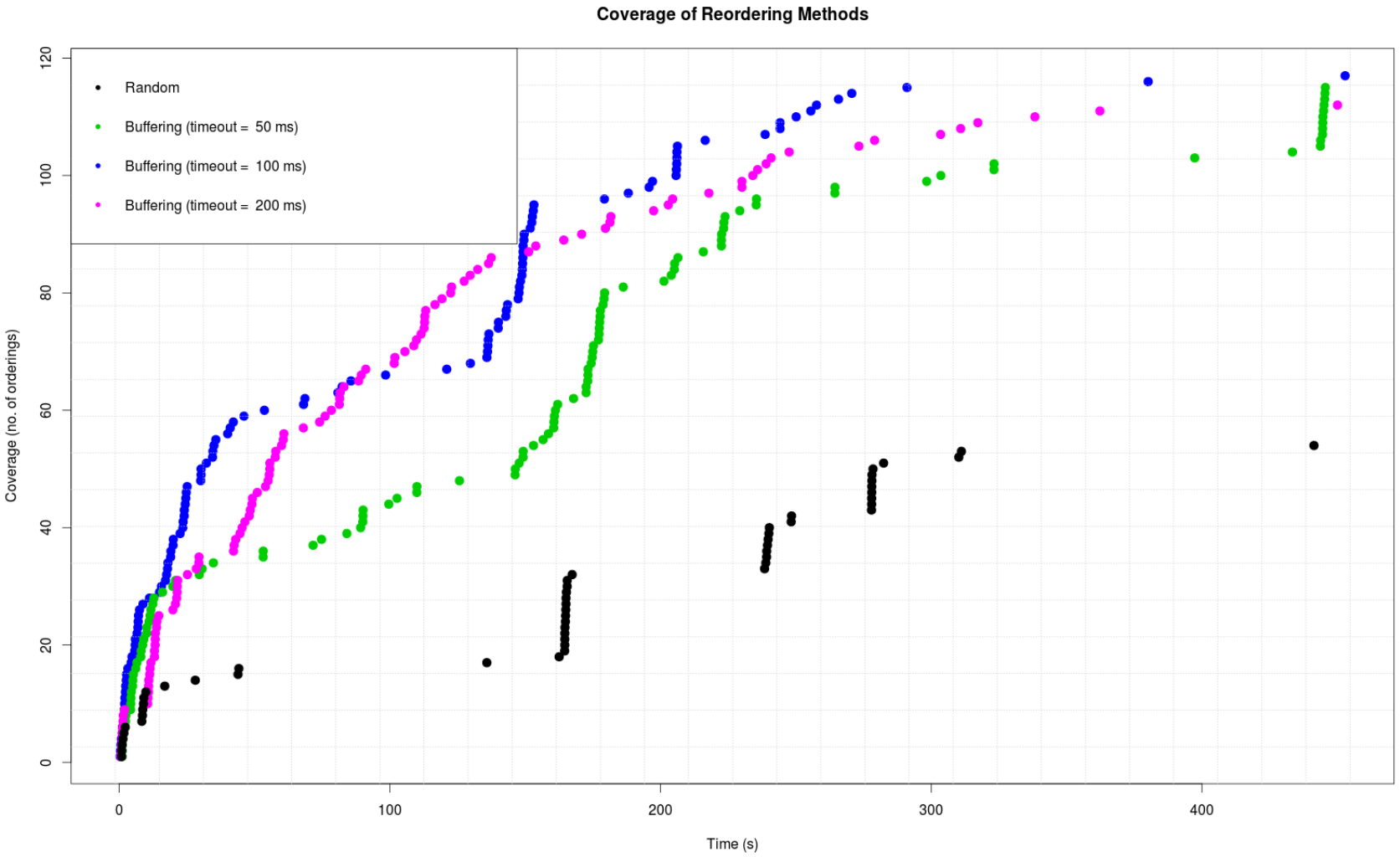

Figure 6.6: The results of running 4 reordering case studies on physical QNX machine.

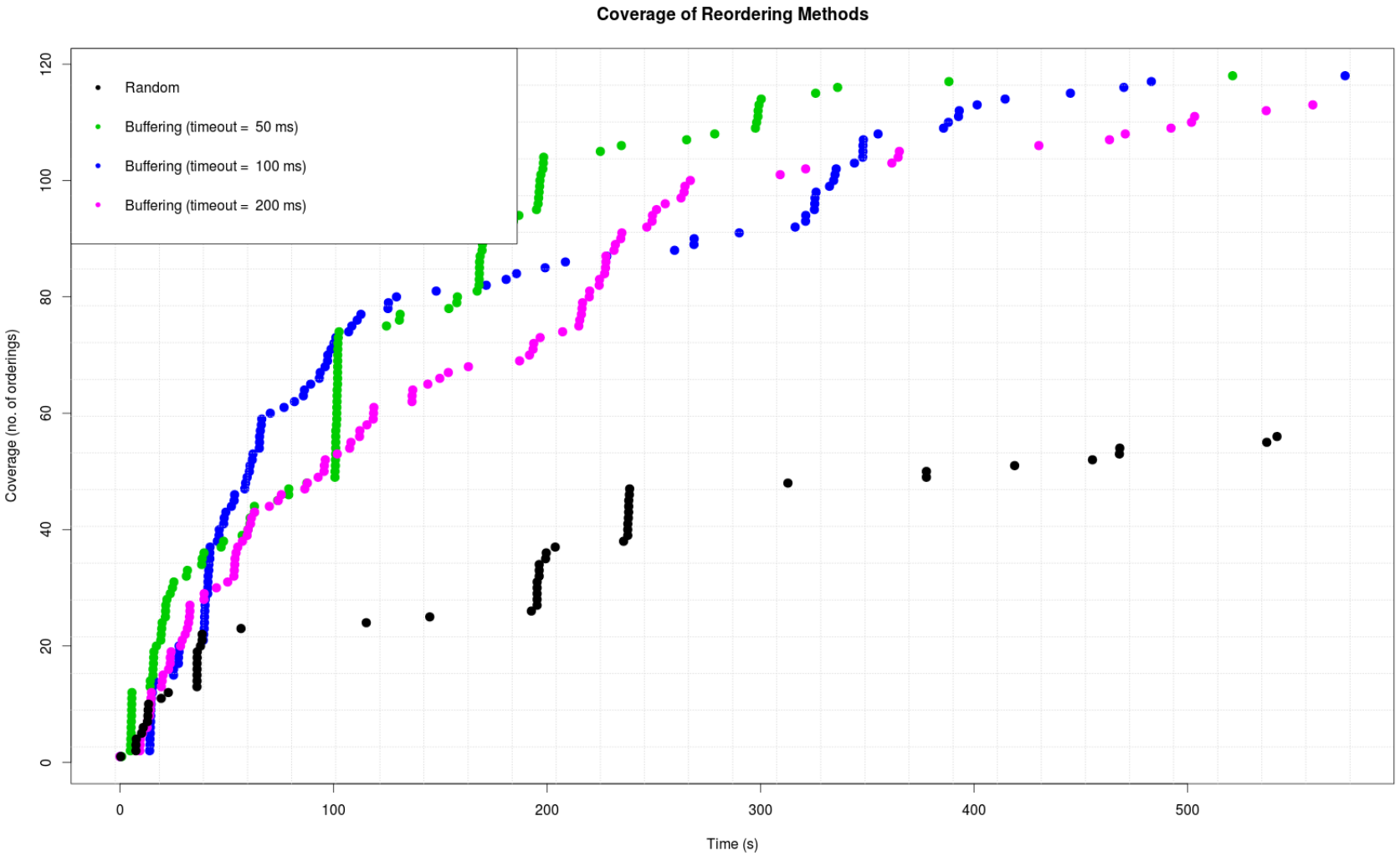

Figure 6.7: The results of running 4 reordering case studies on virtual QNX machine.

Chapter 7

Discussion and Conclusion

7.1 Summary

In message-passing systems every communication between components of the system is done by sending and receiving messages. The order in which messages are delivered is not guaranteed. Thus, a component which receives messages cannot make an assumption about the order of receiving messages. How can we test such a component without having access to its source code to investigate the possibility of software defect? We proposed a new approach for testing software components and finding software defects in message-passing systems without even having access to source codes. For a given receiver component, we can change the ordering of incoming messages. Furthermore, we can try as many distinct orderings as possible and see how the receiver component reacts. If the receiver component make an assumption about the message ordering without explicitly forcing it, our method can reveal such a software defect.

Our reordering framework enables us to limit our attention to a number of components and interactions. We can define dependencies between messages. If message a is dependent to message b, message a does not occur until message b happens. We proposed three different ways for reordering messages in a message-passing system. The *blocking* method blocks each message until all its dependencies occur ahead. The *buffering* method buffers a message until either its dependencies get satisfied or a pre-specified timeout happens. The *adaptive buffering* method works similar to the basic buffering method as it dynamically adjusts the timeout values on-the-fly.

7.2 Single Receiver vs. Multiple Receivers

Based on our problem statement declared in Section 1.3, we consider a given receiver component and define the notion of ordering with respect to that component. It is worth mentioning that our reordering framework is a general-purpose tool for enforcing dependencies between events/messages in message-passing systems. A linear dependency graph is only one use-case for our general dependency graph. So, in the future works we can consider more than a single receiver component at the same time.

7.3 Achievements and Conclusion

We adopted QNX Neutrino 6.5.0 as our message-passing system. QNX is a micro-kernel in which every inter-process communication is done by message-passing. We implemented our reordering framework on QNX. We provided a programming application interface for developers to use in their programs. We also provided a command-line utility that lets system testers make use of our reordering framework without any programming required.

Our experiment results showed that our new methodology for testing increases the coverage criteria to a significant degree in comparison to a random (naive) case. Our case study outcomes showed that our reordering framework applies to real applications in use. In particular, we believe we found a potential software fault in the Photon utility in QNX Neutrino 6.5.0.

We believe our new approach for testing applications by changing the orderings of messages in message-passing systems can reveal software defects which have not been previously discovered or easily detectable using common testing methods and techniques.

7.4 Future Works and Suggestions

Our reordering framework distinguishes messages only by source and destination. Consequently, we treat all messages from the same source to the same destination equally. In real message-passing systems, messages from the same source and destination have more characteristics like *priority*. We can take these properties into account as well. Another area that we can extend our framework to is message dependencies. Currently, a message dependency can be the occurrence of another message. We can extend dependencies to the occurrence of other events in the system as well. More specifically, in the context of the QNX operating system we suggest the following extensions:

1. Priority of messages should be taken into account. Those messages with high priority should not be blocked or buffered too long.

2. Messages with the same source and destination can be further distinguished based on their destination *channels* and their *types*.

3. *SEGEVENTS* can be employed for delivering event notifications asynchronously instead of sending *pulses*.

4. The idea of stopping and continuing a message can be similarly applied to *SEGEVENTS*. Consequently, we can reorder the *SEGEVENTS* as well.

5. Reordering tests can be designed using a graphical user interface. A more intuitive interface would be more pleasurable for testers.

APPENDICES

Appendix A

QNX Machines

Specification	**Capacity**	**Unit**
Architecture	x86	N/A
Processor	3198	MHz
Memory	2039	MB

Table A.1: Specifications of physical QNX machine.

Specification	**Capacity**	**Unit**
Architecture	x86	N/A
Processor	3368	MHz
Memory	3584	MB

Table A.2: Specifications of virtual QNX machine.

Appendix B

Reordering Experiments

Reordering	**Senders**	**Base (ms)**	**Random (ms)**	**Run Time (s)**	**Host**
Random	10	100	[0,2]	N/A	Virtual
Random	10	100	[0,5]	N/A	Virtual
Blocking	10	100	[0,2]	N/A	Virtual
Blocking	10	100	[0,5]	N/A	Virtual
Buffering	10	100	[0,2]	50	Virtual
Buffering	10	100	[0,5]	50	Virtual
Buffering	10	100	[0,2]	100	Virtual
Buffering	10	100	[0,5]	100	Virtual
Buffering	10	100	[0,2]	200	Virtual
Buffering	10	100	[0,5]	200	Virtual
Adaptive	10	100	[0,2]	200	Virtual
Adaptive	10	100	[0,5]	200	Virtual
Random	10	100	[0,2]	N/A	Physical
Random	10	100	[0,5]	N/A	Physical

Blocking	10	100	[0,2]	N/A	Physical
Blocking	10	100	[0,5]	N/A	Physical
Buffering	10	100	[0,2]	50	Physical
Buffering	10	100	[0,5]	50	Physical
Buffering	10	100	[0,2]	100	Physical
Buffering	10	100	[0,5]	100	Physical
Buffering	10	100	[0,2]	200	Physical
Buffering	10	100	[0,5]	200	Physical
Adaptive	10	100	[0,2]	200	Physical
Adaptive	10	100	[0,5]	200	Physical
Random	7	100	[0,2]	N/A	Virtual
Blocking	7	100	[0,2]	N/A	Virtual
Buffering	7	100	[0,2]	50	Virtual
Buffering	7	100	[0,2]	100	Virtual
Buffering	7	100	[0,2]	200	Virtual
Adaptive	7	100	[0,2]	200	Virtual

Table B.1: List of configurations of all reordering experiments.

Appendix C

Slowdown Measurements

Experiment	**Rep. 1**	**Rep. 2**	**Rep. 3**	**Rep. 4**	**Mean**
Random (Naive)	102.512 ms	102.511 ms	102.511 ms	102.511 ms	102.511 ms
Blocking	102.807 ms	102.808 ms	102.809 ms	102.809 ms	102.808 ms
Buffering (50ms)	102.806 ms	102.809 ms	102.804 ms	102.802 ms	102.805 ms
Buffering (100ms)	102.803 ms	102.804 ms	102.804 ms	102.804 ms	102.804 ms
Buffering (200ms)	102.806 ms	102.805 ms	102.807 ms	102.807 ms	102.806 ms
Adaptive Buffering	102.803 ms	102.800 ms	102.799 ms	102.800 ms	102.800 ms

Table C.1: List of measured running times for reordering experiments on virtual QNX machine.

Appendix D

Reordering Case Studies

Reordering	**Timeout (ms)**	**Run Time (s)**	**Host**
Random	N/A	600	Virtual
Buffering	50	600	Virtual
Buffering	100	600	Virtual
Buffering	200	600	Virtual
Random	N/A	600	Physical
Buffering	50	600	Physical
Buffering	100	600	Physical
Buffering	200	600	Physical

Table D.1: List of configurations of all reordering experiments.

Appendix E

Artifacts

Artifact	**Requirements**	**Description**
qnx.ova	VirtualBox 5.0.0+	QNX Virtual machine image
qnx-tagging.ifs	QNX 6.5.0/6.6.0	Reordering-enabled QNX boot image
reorder_experiment	Reordering Framework	Reordering experiments driver
reorder_casestudy	Reordering Framework	Reordering case studies driver
qnx-reordering.git	Git	Complete QNX/Reordering repository
trunk	QNX SDP 6.5.0	Modified source code of QNX Neutrino
doc	LaTeX	Quick start documentation
results	Any Spreadsheet	Experiments and case studies Results
README.md	Any Text Editor	Repository readme file
script/update.sh	Bourne Shell	Populating compiled files to targets
script/load.sh	Bourne Shell	Fetching compiled files to a QNX host
script/gencsv.js	Node.js, MongoDB	Generating CSV files from JSON files
script/genplot.r	RStudio	Generating plots from CSV files

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[2] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, volume 26202649. MIT press Cambridge, 2008.

[4] Francoise Balmas. Displaying dependence graphs: A hierarchical approach. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01*, pages 261–270, 2001.

[5] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold, 2 edition, 1990.

[6] Lionel C Briand, Yvan Labiche, and Yihong Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Proceedings of the 26th International Conference on Software Engineering*, pages 86–95. IEEE Computer Society, 2004.

[7] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Addison-Wesley Professional and Pearson Education Limited, 1 edition, 2003.

[8] Sagar Chaki, Sriram K Rajamani, and Jakob Rehof. Types as models: model checking message-passing programs. In *ACM SIGPLAN Notices*, volume 37, pages 45–57. ACM, 2002.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[10] Augusto Born de Oliveira, Ahmad Saif Ur Rehman, and Sebastian Fischmeister. mtags: augmenting microkernel messages with lightweight metadata. *ACM SIGOPS Operating Systems Review*, 46(2):67–79, 2012.

[11] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 17–30. ACM, 2005.

[12] Michael Factor, Eitan Farchi, Yossi Lichtenstein, and Yosi Malka. Testing concurrent programs: A formal evaluation of coverage criteria. In *Computer Systems and Software Engineering, 1996., Proceedings of the Seventh Israeli Conference on*, pages 119–126. IEEE, 1996.

[13] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 62–75. IEEE, 2003.

[14] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

[15] Wesley K Fuchs, Yennun Huang, and Yi-Min Wang. Progressive retry method and apparatus for software failure recovery in multi-process message-passing applications, December 1996. US Patent 5,590,277.

[16] Bill Gallmeister. *POSIX.4 Programmers Guide: Programming for the Real World*. O'Reilly Media, 1 edition, 1995.

[17] Jerry Gao, Raquel Espinoza, and Jingsha He. Testing coverage analysis for software component validation. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 1, pages 463–470. IEEE, 2005.

[18] Hector Garcia-Molina and Annemarie Spauster. Message ordering in a multicast environment. In *9th International Conference on Distributed Computing Systems*, pages 354–361, 1989.

[19] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313. ACM, 2013.

[20] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[21] Dan Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[22] IEEE. Institute of electrical and electronics engineers.

[23] British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST). *Standard for Software Component Testing*. British Computer Society, 2001.

[24] Shinji Inoue and Shigeru Yamada. Two-dimensional software reliability assessment with testing coverage. In *Secure System Integration and Reliability Improvement, 2008. SSIRI'08*, pages 150–157. IEEE, 2008.

[25] Doug Kimelman and Dror Zernik. On-the-fly topological sorta basis for interactive debugging and live visualization of parallel programs. In *ACM SIGPLAN Notices*, volume 28, pages 12–20. ACM, 1993.

[26] John P. Klein, Hans C. van Houwelingen, Joseph G. Ibrahim, and Thomas H. Scheike. *Handbook of Survival Analysis*. Chapman and Hall/CRC, 1 edition, 2013.

[27] David G Kleinbaum and Mitchel Klein. *Survival Analysis*. Springer, 3 edition, 1996.

[28] Bogdan Korel. The program dependence graph in static program testing. *Inf. Process. Lett.*, 24(2):103–108, 1987.

[29] Bettina Krammer, Matthias S Müller, and Michael M Resch. Mpi application development using the analysis tool marmot. In *Computational Science-ICCS 2004*, pages 464–471. Springer, 2004.

[30] David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[31] Tong Li, Carla Schlatter Ellis, Alvin R Lebeck, and Daniel J Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *USENIX Annual Technical Conference, General Track*, volume 44, 2005.

[32] Panos E. Livadas and Theodore Johnson. An optimal algorithm for the construction of the system dependence graph. *Inf. Sci.*, 125(1-4):99–131, 2000.

[33] Yashwant K Malaiya, Naixin Li, Jim Bieman, Rick Karcich, and Bob Skibbe. The relationship between test coverage and reliability. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 186–195. IEEE, 1994.

[34] Brian Marick. New models for test development. *Testing Foundations*, 1999.

[35] Oliver A. McBryan. An overview of message passing environments. *Parallel Computing*, 20(4):417–443, 1994.

[36] Atif M Memon, Mary Lou Soffa, and Martha E Pollack. Coverage criteria for gui testing. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 256–267. ACM, 2001.

[37] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, (9):46–55, 2009.

[38] Rupert G Miller Jr. *Survival Analysis*, volume 66. John Wiley & Sons, 2011.

[39] Christian Murphy, Kuang Shen, and Gail Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 189–200. ACM, 2009.

[40] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[41] Robert HB Netzer, Timothy W Brennan, and Suresh K Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 31–40. ACM, 1996.

[42] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2 edition, 2004.

[43] The Institute of Electrical and Inc. Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[44] Object Management Group (OMG). Unified modeling language (uml), 1997-2015.

[45] Rahul Pandita, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Guided test generation for coverage criteria. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[46] David J Pearce and Paul HJ Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics (JEA)*, 11:1–7, 2007.

[47] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 142–151, 2008.

[48] Dennis Peters and David L Parnas. Generating a test oracle from program documentation: work in progress. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 58–65. ACM, 1994.

[49] Hoang Pham and Xuemei Zhang. Nhpp software reliability and cost models with testing coverage. *European Journal of Operational Research*, 145(2):443–454, 2003.

[50] Rüdiger Rackwitz. Reliability analysis: a review and some perspectives. *Structural safety*, 23(4):365–395, 2001.

[51] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4 edition, 2011.

[52] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Siti Zaiton Mohd-Hashim. A comparative study on automated software test oracle methods. In *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*, pages 140–145. IEEE, 2009.

[53] Stephen F Siegel and Ganesh Gopalakrishnan. Formal analysis of message passing. In *Verification, Model Checking, and Abstract Interpretation*, pages 2–18. Springer, 2011.

[54] SRS Souza, Silvia Regina Vergilio, PSL Souza, AS Simao, and Alexandre Ceolin Hausen. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience*, 20(16):1893–1916, 2008.

[55] QNX Software Systems. 10 steps to developing a qnx program: Quickstart guide [6.5.0], 2004-2007.

[56] QNX Software Systems. Qnx software development platform installation guide [6.5.0], 2004-2007.

[57] QNX Software Systems. Qnx software development platfrom 6.5.0 docs, 2004-2007.

[58] Mara Tableman and Jong Sung Kim. *Survival Analysis Using S: Analysis of Time-to-Event Data*. Chapman and Hall/CRC, 1 edition, 2003.

[59] Juichi Takahashi, Hideharu Kojima, and Zengo Furukawa. Coverage-based testing for concurrent software. In *Distributed Computing Systems Workshops, 2008. ICDCS'08. 28th International Conference on*, pages 533–538. IEEE, 2008.

[60] Richard Turner. Toward agile systems engineering processes. *Crosstalk. The Journal of Defense Software Engineering*, pages 11–15, 2007.

[61] Jeffrey S Vetter and Bronis R De Supinski. Dynamic software testing of mpi applications with umpire. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 51–51. IEEE, 2000.

[62] Richard Vuduc, Martin Schulz, Dan Quinlan, Bronis De Supinski, and Andreas Sæbjørnsen. Improving distributed memory applications testing by message perturbation. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 27–36. ACM, 2006.

[63] Neil Walkinshaw, Marc Roper, and Murray Wood. The java system dependence graph. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 55–64, 2003.

[64] Yi-Min Wang, Yennun Huang, and Kent W Fuchs. Progressive retry for software error recovery in distributed systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 138–144. IEEE, 1993.

[65] Yi-Min Wang, Yennun Huang, and C Kintala. Progressive retry for software failure recovery in message-passing applications. *Computers, IEEE Transactions on*, 46(10):1137–1141, 1997.

[66] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[67] Michael W Whalen, Ajitha Rajan, Mats PE Heimdahl, and Steven P Miller. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36. ACM, 2006.

[68] Karim Yaghmour and Michel R Dagenais. Measuring and characterizing system behavior using kernel-level event logging. 2000.