석 사 학 위 논 문

Master's Thesis

시맨틱 정보를 이용한 패턴 매칭을 통한 동시성 결함 검출

Concurrency Bug Detection through Improved Pattern Matching Using Semantic Information

홍 신 (洪 申 Hong, Shin)

전산학과

Department of Computer Science

KAIST

2010

시맨틱 정보를 이용한 패턴 매칭을 통한 동시성 결함 검출

Concurrency Bug Detection through Improved Pattern Matching Using Semantic Information

Concurrency Bug Detection through Improved Pattern Matching Using Semantic Information

Advisor : Professor Kim, Moonzoo

by

Hong, Shin
Department of Computer Science
KAIST

A thesis submitted to the faculty of the KAIST in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Department of Computer Science

Daejeon, Korea
2009. 12. 17.
Approved by

Professor Kim, Moonzoo
Advisor

시맨틱 정보를 이용한 패턴 매칭을 통한 동시성 결함 검출

홍 신

위 논문은 한국과학기술원 석사학위논문으로 학위논문심사위원회에서 심사 통과하였음.

2009년 12월 17일

MCS 20073636 홍 신. Hong, Shin. Concurrency Bug Detection through Improved Pattern Matching Using Semantic Information. 시맨틱 정보를 이용한 패턴 매칭을 통한 동시성 결함 검출. Department of Computer Science . 2010. 70p. Advisor Prof. Kim, Moonzoo. Text in English.

Abstract

Many software systems today are concurrent programs as multi-core processors become popular. However, the correctness of an industrial-size concurrent program (e.g. operating system) is difficult to achieve by the traditional testing or model checking technique. In this research, we propose a light-weight concurrency bug detection technique based on bug pattern matching targeting for Linux kernel source code. In order to understand concurrency bugs (e.g. deadlock, data race), we first survey the previously reported bugs detected from Linux file systems, and then classify the bugs with respect to the five attributes: symptom, fault, resolution, synchronization primitives, and synchronization granularity. Second, we identify ten concurrency bug patterns. And then we develop the bug pattern detectors and applied to the Linux file systems. Finally, and foremost, we improve the accuracy of the concurrency bug detection technique by enhancing semantic information in pattern matching. We demonstrate the effectiveness of our technique through detection of concurrency bugs in the Linux file systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many software systems today are concurrent programs. As multi-core processors become popular, most software systems are designed and implemented as concurrent programs, and moreover some programs are converted into concurrent programs to improve its performance and responsiveness. Many recent embedded systems use concurrent programs in their software as the embedded processors are improved and the functional requirements are increased.

1.1 Concurrent Program

By the definition in [1], a concurrent program is a collection of interacting computational processes that may be executed in parallel. Each interacting computational processes communicates with other processes via message passing or shared memory. The concurrent program refers to a wide area of programs including multithreaded programs and distributed systems. In this research, we concentrate on multithreaded programs. Multithreaded programs refer to a type of concurrent programs with the following characteristics: (1) each interacting computational process is implemented as a thread, (2) A program has a shared memory space which can be accessed by all threads, (3) On a single process, threads are executed by interleaving with fair scheduling, and (4) Synchronization mechanisms such as conditional variables, locks, and semaphores are used. Many concurrent software systems are multithreaded programs written in C and Java.

In multithreaded programs, threads communicate through shared resources in shared memory space. A shared resource might be a data structure or an interface in shared memory space. Therefore, proper synchronizations on shared resources are necessary for correct program behaviors. For this purpose, critical sections are widely used. A critical section is a sequence of instructions (or a code block) for which atomicity is guaranteed. Atomicity of instructions refers to a property that the results of concurrent executions of the instructions are always the same as the result of an execution without any interleaving with respect to a starting state. This property enables programmers to program a complex code safely with sequential execution semantics.

One naive approach to implement critical sections is interrupt-disabling. However, this approach results bad performance, and also inappropriate for multi-core environments. In most multithreaded programs, locks are used for implementing critical sections. In this approach, all critical sections which manipulate one shared resource are guarded by one lock. This enables atomicity of the critical sections since no two code block guarded by one lock can be executed concurrently and there is no other code which manipulates the shared resource. This approach has better performance than the first approach since two critical sections which manipulate independent shared resource can be executed concurrently.

1.2 Concurrent Programming Problems

As concurrent programming becomes widespread, the correctness assurance of concurrent program becomes much important issue. However, it is much difficult to implement, verify, and debug concurrent programs than sequential programs. There are the following three reasons for the difficulties.

First, concurrent programming requires the understanding of whole program codes. In sequential programming, behaviors of a part of code depend on input data and other codes nearby. Unit testing is effective to sequential program analysis. However, in concurrent programs, behaviors of a part of code depend on not only input data but also all other code which can be executed concurrently with the target code. Therefore, compositional analysis (or understanding) of concurrent program codes is hard to achieve.

Second, the number of distinguishable interleaved executions from a concurrent program increases exponentially with respect to the number of processes in a program and the average process size. For this reason, it is unfeasible to apply traditional testing and model checking techniques to industrial size concurrent programs such as operating systems and database management systems.

Third, the complexity of concurrent program code is generally high since programmers apply many optimization techniques to improve program performance. Moreover, the most widely used programming language C does not support concurrency in the programming language level. Therefore, the code structure for controlling concurrency in C programs varies depending on programmers' style. For the above reasons, it is complicated to analyze the concurrent program codes in general.

1.3 Error, Bug, Failure, Fault

In this paper, we use the vocabularies for indicating software defects based on IEEE Standard 729 Glossary for Software Engineering Technology [2]. For a terminology 'error', the standard 729 describes four different meanings: (1) Error is the difference between a computed value, an observed, and a theoretically correct value. (2) Error is an incorrect step, process, or data definition. (3) Error is an incorrect result. (4) Error is a human action that produces an incorrect result.

We refine second and third meanings for further discussion. We will use 'bug' for second meaning, and 'error' for third meaning. We never use 'error' with first meaning, and also fourth meaning since programs are required to be aware of all possible human actions.

The IEEE standard introduces 'failure' and 'fault'. Failure is the inability of a program to perform its required functionality. And a fault is a defect in either hardware or software which causes a failure. Two terminologies 'bug' and 'faults' are conceptually similar, but 'bug' accents software, whereas 'fault' accents system including both software and hardware. Correspondingly, 'error' and 'fault' are similar but different. For this reason, we mainly use 'bug' and 'error' in this paper since the research targets software defects.

1.4 Concurrency Errors

Concurrency errors refer to unexpected program behaviors caused by incorrect synchronizations in concurrent programs. A distinction of concurrency errors from non-concurrency errors is that the condition where a concurrency error occurs in a program not only depends on input data but also depends on thread scheduling. For the same given data, an execution results concurrency errors but other execution with a different thread schedule may not result any concurrency errors.

There are mainly two types of concurrency errors: one is race condition, and the other is deadlock. A race condition refers to a situation that there exists an execution of a critical section results an unexpected state. Race conditions are mainly caused by incorrect implementations of critical sections, which do not guarantee the atomic executions. Race condition bugs can be classified into two types. One type is data race bug which indicates a case that a shared resource is not synchronized properly. The other type is atomicity violation. An atomicity violation bug is caused when a critical section which manipulates two or more shared resources is not always executed atomically.

In Section 6, we survey the currently developed race detection techniques.

A deadlock refers to a situation that an execution of a critical section is indefinitely stopped depending on thread scheduling. Deadlock is caused when a set of threads try to hold at least two locks simultaneously. In general multithreaded programs, one thread can hold one lock by one instruction. So, in order to acquire several locks, a thread first holds one lock and then acquires the other lock in a sequence. Deadlocks can happen when two threads try to acquire a set of locks concurrently without proper synchronization. Most simple example is as follow. There are two threads Thread1 and Thread2. Two threads try to acquire lock A and B. Thread1 first acquire lock A and then lock B. Thread2 first acquire lock B and then lock B. Deadlock happens in following execution scenario: (1) Thread 1 holds lock A, (2) Thread 2 holds lock B, (3) Thread 1 acquire lock B, (4) Thread 2 acquire lock A. Deadlock has been studied since concurrent programming was introduced. Coffman introduced a reasonable necessary deadlock condition called Coffman condition in 1971 [3]. An execution is deadlock if it satisfies the following four conditions:

- Mutual exclusion condition: A resource that cannot be used by more than one process at a time
- Hold and wait condition: Processes already holding resources may request new resources
- No preemption condition: No resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process.
- Circular wait condition: Two or more processes from a circular chain where each process wait for a resource that the next process in the chain holds.

Many researchers have developed the techniques to detect deadlock errors or potential deadlock. In Section 6, we survey these techniques.

Livelock refers to either starvation or infinite execution in general. Starvation is a situation where a task on a resource is not scheduled so that its progress is standstill. Infinite execution is a situation where a thread' s execution is not progressed even though it is scheduled. In most cases, infinite executions stuck in loops [3].

1.5 Approach

In this research, we propose a light-weight concurrency bug detection technique based on bug pattern matching. This technique aims to result fast and effective bug detections from industrial size multithreaded C program codes.

1.6 Contributions

The contributions of this research are the followings:

- Classify concurrent bugs
  We report analysis result of previous concurrency bugs in Linux kernel. And we suggest the concurrency bug classification which can be used to assist systematic understanding of concurrency bugs and define bug patterns.

- Detect concurrency bugs using the studied bug patterns
  We suggest an approach to define concurrency bug patterns and use the defined bug patterns for bug detections. This approach can be used for other bug patterns. Moreover, it can be used for other techniques using bug patterns. We detected suspected bugs in Linux kernel using the defined bug patterns. We report the detected bugs to Linux maintainers.

- Enhance accuracy of bug pattern matching
  We developed the improved bug pattern matching technique by considering semantic information from codes. This technique effectively eliminates the false alarms from the syntactic bug pattern matching results. This technique can be utilized for other static analysis for concurrency bug detections.

1.7 Structure of this paper

We first explain the basic concepts of concurrency and concurrency bugs in Section 2. In Section 3, we introduce the concurrency bug classification developed by previous Linux bug surveys for better understanding of real concurrency bugs. We introduce five concurrency bug patterns and report two bug pattern detector implementations and the results of applying the tools to Linux file system code in Section 4. In Section 5, we suggest an improved bug pattern matching using semantic information for more accurate results. In Section 6, we explain the related works on various concurrency bug detection techniques. We conclude our research results and suggest future work in Section 7.

Chapter 2

Related Work

Many researches have been developed to detect concurrency bugs including race condition and deadlock. Each bug detection technique has a bug definition and a program analysis algorithm. A bug detection technique apply the algorithm to analyzes the behavior of target program to check whether the bug definition can be satisfied by a target program. In this section, we classify the concurrency bug detection techniques with respect to these two aspects: One is program analysis technique and the other is bug detection technique. And then we survey the previoulsy developed techniques in each attribute.

2.1 Program Analysis Techniques

2.1.1 Code pattern based techniques

FindBugs [4] detects bug canddiates by pattern matching Java binary with respect to the specified various Java bug patterns. FindBugs framework use BCEL Java binary parser for binary code pattern matching by class structure analysis, linear code analysis, control flow analysis, and data flow analsys. In [5], the authors introduce FindBugs approach for finding concurrency bugs in Java programs.

Dawson Engler and his research group analyze the previously reported bugs from Linux and OpenBSD codes in [6]. They first study the nature of bugs with respect to the distribution of bugs over operating system modules and the time duration between a bug introduction and the bug path. In following research [7], they suggest a bug detection approach using code pattern matching. In this work, they introduce a state-machine based bug specification language *MetaL*. A *MetaL* description specifies a set of execution path. Many bugs including both sequential bugs and concurrency bugs are detected by this techniques from Linux and OpenBSD.

ConTest infrastructure has a feature which utilize bug patterns to trigger concurrency errors during test runs. Farchi and his research team in IBM Haifa introduce this technique in [8]. This technique first finds a candidate concurrency bugs through code patterns. And then, it inserts noise injections at the candidate bug site in order to detect concurrency bugs with high probability in testing. Upon ConTest, this technique contributes to active testing of concurrent Java program. Farchi and his research team also adopt bug patterns for assisting code review process [9]. The authors extend the regular expression in Perl language for bug specifications and bug detections. As a preprocessing to code review by experts, this technique automatically attach the comments on a code which is corresponding to a given bug specification.

Opera Laboratory at UCSD(Previously at UIUC) published empirical study on concurrency bugs with meaningful discussion on bug patterns [10]. This research analyze previously detected concurrency bugs from 4 large opern source programs: MySQL, Apache, Mozilla, and OpenOffice. The authors advocate an idea to develop bug detection techniques motivated from common charactersitics of concurrency bugs.

2.1.2 Rule based techniques

The techniques belong to this category enforce a set of programming rule which guarantee that generated programs do not result any incorrect behavior. There exist both top-down approach and buttom-up approach to enforcing programming rules. Top-down approach assumes that programming rules are given before analysis. Buttom-up approach rather extract the programming rules used in a target program and then check whether the programming rule is consistently used around whole program.

Top-down approach

Most techniques of this approach extend the type systems of existing programming languages. The extended type systems check a program correctly follows a given programming rules to ensure that the desired properties holds for the program. The type systems are noramlly implemented as a part of compilers.

Flangan and Stephen N. Freund [11] introduce an extended Java type system to avoid concurrency errors including deadlock and data race. This type system requires every shared member has a specification of its synchronization object. Users should specify the specification and then the type system automatically check whether there is a possibility of deadlock and data race error. The authors implement this type system as rccjava which inputs annotations and automatically check the absence of deadlock and data race from an annotated Java program. Flangan and Shaz Qadeer extend the concept to dynamic analysis in [12]. They suggest an algorithm based on Lipton's reduction theory to ensures the absence of atomicity violation from an observed execution.

In [13], the authors introduce a similar type system approach to Flanagan's work for Safe Concurrent Java. This type system enforces partial-order of locking orders to avoid deadlock and also enforces lock disciplines to avoid data race. This type system adopts ownership types to consider encapsulations. And this system has a capability to extract type in order to reduce human labors.

Cyclone is a dialect of C for writing safe system programs. In [14], the authors extend Cyclone's type system to generate data race free multithreaded C programs. Cyclone is a C style programming language which supports additional language features for safe and effective programming such as region based memory management and parametric polymorphism. The extended Cyclone input a lock name for every pointer type, a lock name for every lock type and a locking constrain for every function as annotations to guarantee absence of data race.

NesC [15], a dialect of C language for safe programming, avoid data race error in compiler level by enforcing simple invariant. It checks all variables accessed in any asynchronous code should be accessed in atomic blocks.

These type system based program analysis techniques guarantee the certian properties from target programs. However, there are two shortcomings of this approach to be applied to general programs. First, these systems restrict programmers to write codes in simple manner strictly. Second, these methods require programmers to specify the additional information related to synchronization used in a code. These two shorcomings are unfeasible for targeting system programs. In system programs, fine tuning of synchronization operations are common in order to improve performance. Moreover, the size of program is normally too large for programers to give the additional information manually.

Buttom-up approach

The techniques in this approach first extract suspected programming rules with respect to programming rule templates. And then, the techniques check whether the programing rules are consistently and completely applied to the target program or not. The most popular programming rule templates are lock discipline and partial order in nested lock acquiring order. A lock discipline is a rule to enforce the existence of a lock which synchronizes every access to a shared variable. It gurantees the absence of data race of the shared variable. Partial order in nested lock acquiring order (simply lock ordering) is a rule that enforce the existence of partial order relation in nested lock acquiring order. This rule gurantees the absence of deadlock.

Dawson Engler and Ken Ashcraft develop RacerX [16] which extract lock discipline and lock ordering to detect data race bugs and deadlock bugs respectively. RacerX extract programming rules in use by traversing inter-procedural control-flow graph. The absence of consistent lock discpline and the absence of partial order relation in a program are interpreted as data race bug and deadlock bug respectively. RELAY also checks the existence of lock discipline to detect data race bugs. The fundamental concept of RELAY is similar to that of RacerX. But RELAY adopts lock analysis, function summary, and symbolic execution techniques for better accuracy in the analysis result.

These techniques are effectively detect programming rule violations from program source code. The major limitation of these techniques is the extensibility. These techniques only can verify the correct use of given programming rules.

2.1.3 Stateful model checking

Model checking techniques exhaustively check the possible behaviors of a model to provide sound and complete verifications. Software model checking techniques aim to verify software by automatic model constructions from source code and aslo by abstraction techniques which alleviate state explosion. Complete analysis is desirable for verifying concurrent programs since concurrent programs have non-determinism in nature. However, state-of-art software model checking techniques are still not scalable enought to verify large-size concurrent software. Most techniques still concentrate on sequential program verification.

TCBMC [17] is a bounded model checker for multithreaded programs. TCBMC translate a given multithreaded C program into the corresponding SAT formula with a given bound as CBMC works. TCBMC additionally input a context-bound which is a upper bound of the number of context switching in an execution.

KISS transforms a given concurrent program into a sequential program which simulates a partial behavior of the concurrent program. KISS verifies the transformed sequential program using SLAM model checker. Users can specify requirement properties as assertion statements. In [18], the authors report experimental results of applying KISS to Windows device drivers with several thousand lines of codes. In the experiments, test harness was manually constructed before the model checking.

Blast employs thread modular abstraction technique for verifying multi-threaded programs [19]. This technique model checks concurrent execution of one thread and environment model at a time, and then refine the environment model by the result of the model checking until the environment model reaches to a fixed point and every thread satisfy the requirement property with the environment model.

MOKERT framework adopts SPIN model checker with semi-automatic model extractor MODEX. In this framework, an abstract model is extracted from C codes using Modex tool and then the model is verified using SPIN model checker. This framework supports counter example replay module through which the validity of counter example and the abstract model can be checked.

2.1.4 Dynamic analysis

Dynamic analysis techniques (a.k.a runtime analysis) aim to verify a certain property of a program by evaluating its actual executions. By observing internal states during target program executions, the dynamic analysis techniques can use accurate information of program behaviors. In dynamic analysis, it is possible to achieve value-sensitive and alias-sensitive analysis with much less computation cost than in static analysis.

Dynamic analysis extends traditional testing to check meaningful properties using intermediate state information in program executions. Concurrent executions are not only determined by test cases but also by thread scheduling.

Dynamic analysis includes the technique for controlling thread scheduling.

Dynamic analysis techniques can be categorized into three independent layers. First layer is systematic testing layer. This layer is to execute target programs within policies. These policies aim to reach error states effectively. Second layer is information extraction layer. The information on the internal behaviors of the target programs is extracted to be used for the program correctness checking. At third layer, the monitors generate abstract model of the target program from the extracted information and then verify the abstract model to detect possible errors in the program.

Dynamic analysis techniques share the limitations of testing inherently. Dynamic analysis cannot support complete analysis for target programs since it uses monitored partial behavior of the target programs. The other limitation is that dynamic analysis techniques are difficult to be applied unless target programs are complete. Dynamic analysis techniques require executable environments and test cases. However, these can be delivered only at later phase of software development especially for embedded software.

For checking industrial concurrent programs such as Linux file systems, our bug pattern detection approach has the advantages for the following points. First, the bug pattern detection technique can be applied for incomplete program source code so that it may be applied for early phase of software developments. Second, the bug pattern detection technique is scalable for the complexity of target programs whereas dynamic analysis techniques are not. Systematic testing techniques including ConTest [8] and AtomRace [20] investigate concurrency bug patterns in systematic testing layer to simulate common concurrency error execution patterns.

In this section, we survey the techniques for stateless systematic testing techniques and information extraction techniques. Former one is used for first layer, and later one is used for second layer. The techniques for third layer will be discussed in the next subsection.

Stateless systematic testing

Systematic testing techniques regulate thread scheduling to reach error states effectively in testing. By non-determinism in thread scheduling, the number of distinguishable thread scheduling scenarios increases exponentially with respect to target program size. Traditional testing methods cannot verify concurrent programs effectively.

The motivations underneath systematic testing techniques are to increase high coverage and explore a property of thread scheduling which is commonly observed from concurrency errors. Systematic testing techniques use the information of internal structures of target programs. For this point, systematic testing is different from traditional black box testing. Systematic testing is similar with software model checking since these two techniques verify implementation directly. However, systematic testing does not use the visited state information for controlling thread scheduling whereas software model checking does.

One motivation is to explore as many distinguishable thread scheduling scenarios as possible in a given time. For two distinguishable thread scheduling scenario, one thread scheduling is reducible to the other thread scheduling if their resulting states are the same. There are many distinguishable thread scheduling scenarios which result the same program state or program behavior. For effective testing, it is desirable to explore thread scheduling scenarios that are not reducible to each other. Exploiting different states would increase the probability of reaching error states in a given amount of time. Using partial order reduction techniques, it is possible to avoid exploration of reducible thread scheduling scenarios. Verisoft [21] uses partial order reduction concept in systematic testing of concurrent programs. Verisoft controls message passing among processes and non-deterministic choice in a process to test non-reducible scheduling scenarios. A dynamic model checking technique Inspect improves dynamic partial order reduction technique to additionally consider locking behavior of target programs [22].

The other motivation is to explore error-prone thread scheduling scenarios to detect a certain type of errors effectively. CHESS [23] uses context-bounded search which limit the number of context-switching for a thread scheduling scenarios in a testing. This approach is motivated by an observation that most of concurrency bugs cause concurrency errors within a small number of context-switches. CHESS allows users to specify requirement property as linear temporal logic formulae. And CHESS can check whether an execution satisfies the Good Samaritan property which is a desirable fairness property in concurrent programs or not. ConTest [8] is an infrastructure for concurrency program testing. This technique inserts noise injection probes to target programs to generate concurrency errors in high probability. A noise injection probe is a piece of code which does not affect original program behaviors except for execution timing by yielding and sleeping for an amount of time. ConTest uses concurrency bug patterns. Many concurrency bugs are caused by undesirable context switching in a code. ConTest insert noise injection probes at critical points of pattern matched codes to induce errors. CalFuzzer [24] is an active testing framework which enforces error-prone thread scheduling scenarios. CalFuzzer understands candidate bugs in a program by applying dynamic analysis to the program, and then guide thread scheduling in testing to generate the error by the candidate bugs. CalFuzzer instruments target program source code to control thread scheduling.

Information extraction

Dynamic analysis techniques transform target programs to access the internal state information during executions for verifying the target programs. Dynamic analysis techniques statically analyze a target program to indicate for which variables correctness properties will be checked. And then, additional programming features are inserted into the target program to monitor the value of the interested variables in dynamic analysis. Deadlock detection techniques track the behavior of synchronization objects such as mutex, semaphore, and etc. Race condition detection techniques target the behaviors of both shared variables and synchronization objects in use. Information extraction should not change any state (semantic) behavior of taret programs, but it degrades the time performance of target programs since the transformed program executes additional code for the monitoring. For efficient dynamic analysis, novel information extraction techniques to reduce the runtime cost of monitorings have been developed. Choi et al suggest a novel algorithm to reduce useless insertion of monitoring probes in object-oriented programs by static analysis techniques. Velodrone [25] imporve the performance of happen-before monitoring by additionally considering lockset information at each probe execution.

Dynamic analysis techniques insert monitoring probe codes by several program instrumentation techniques. Eraser [26] uses ATOM [27] binary modification system to instrument target programs. Velodrome [25] uses BCEL[28] Bytecode Engineering Library to instrument Java intermediate code. ConTest infrastructure [8] use its own instrumentation engine specialized for multi-threaded Java program testing. In [29], the tool uses its own compiler for instrumenting Java intermediate codes. The concurrency bug detection techniques developed by Scott D. Stoller et al. [30] In [30], the technique transforms target programs' source code to extract internal behavior during program executions. MultiRace [31] instrument the source code of target C / C++ programs to use page fault handler for logging interesting read and write operations.

2.2 Bug Detection Techniques

2.2.1 Deadlock

A deadlock is a situation where threads stay in waiting state indefinitely so that their executions are stopped. Major source of deadlocks is circular waiting situation which is that thread A is waiting for thread B's execution meanwhile thread B is waiting for thread A's execution. The deadlock problem related to the major source has been studied since concurrent programming concept was first introduced. As a minor source of deadlock, a program can result deadlock situations when a thread disable preemptive scheduling by incorrect interrupt disabling. However, a program controls scheduling freely in general. Only in specific circumstances such as operating system kernel can make this type of deadlock. Therefore, most deadlock detection techniques concentrate on the major type deadlock.

Coffman condition [1] is a practical necessary condition of deadlock occurrence, which is suitable for lock based multithreaded programs. In order to check fourth condition of Coffman's deadlock condition, lock graph which represent dependency relations among threads and resources has been used in dynamic analysis and model checking. Coffman introduces a sufficient condition of deadlock occurrence in an execution. The following is four conditions.

- Mutual exclusion condition
  A resource that cannot be used by more than one process at a time
- Hold and wait condition
  Processes already holding resources may request new resources
- No preemption condition
  No resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process.

- Circular wait condition

  Two or more processes from a circular chain where each process wait for a resource that the next process in the chain holds.

Lock graph can be used to check whether an execution satisfies Coffman's condition or not. This method is post-mortem since it reports deadlock bugs only if actual deadlock error is occurred in testing. However, post-mortem deadlock detection helps debugging only if deadlock errors are revealed in testing. Potential deadlock detection technique is desirable since achieving high coverage of multithreaded program in testing is in general difficult.

Many synchronization mechanisms in use satisfy mutual exclusion, hold and wait, and no preemption conditions. Most potential deadlock detection techniques check a program can reach a state of any circular waiting. Otherwise, it cannot report any possibility of deadlock bugs which might be located in uncovered by testing. In general, checking the presence of deadlock from a given program is undecidable. Recently, researchers developed the technique by (1) checking partial ordering in lock order relation, and (2) checking deadlock in abstract synchronization models for checking possible deadlocks.

Static potential deadlock detection techniques

There exists a lock ordering relation from lock A to lock B if the program acquires lock B while it holds lock A in an execution. It is known that if there exists partial ordering in lock ordering relations, deadlock will never occur. This condition is a stronger condition of the absence of deadlock. However, enforcing partial ordering is used as deadlock avoidance policy in large program including Linux file system. RacerX [16] statically traverse inter-procedural control-flow graph of a program while it records lock ordering relation. It reports not partially ordered lock ordering relations as deadlock bugs by constraint solving. RccJava [32] checks the presence of a partial order relation in a program by type checking in compiler level. It does not allow any program unless a strict partial order relation exists in the lock ordering relation of the program.

Dynamic potential deadlock detection techniques

Dynamic potential deadlock detection techniques construct an over-approximated abstract model which reflects synchronization behavior of a target program based on the monitored information. And then, the techniques perform deadlock checking in the abstract model. The modified lock graphs are used for abstract models in these techniques. Researchers suggest ideas of reducing false alarm caused by the gap between an abstract model and the original program in deadlock detection techniques.

It is possible to construct an abstract model which represents synchronization behavior of a target system. We can detect deadlock in the abstract model in order to detect deadlock in the target program.

GoodLock algorithm constructs a lock tree for each thread, and then detects two opposite lock ordering in two threads' lock tree. In the detection, the algorithm check the presence of any guarding lock, a lock always held before two locks, to exclude false alarms. In [33], the generalized GoodLock algorithm for arbitrary number of threads is introduced for dynamic potential deadlock detection. In [30], the authors suggest an approach to consider deadlock related to conditional variables and semaphores. In [34], another generalized GoodLock algorithm which is parallel to [30] is introduced. The authors adopts the generalized GoodLock algorithm in deadlock detection using Java Path Finder software model checker and suggest a method to validate a potential deadlock detection using Java Path Finder [35].

2.2.2 Race condition

A race condition refers to a situation where a program execution reaches to unexpected states due to concurrency. Race conditions are caused when the synchronizations for critical sections are incorrect so that the atomicity of the critical sections does not hold. Many race condition detection techniques have been proposed, but it is still difficult to assure the absence of race condition because most programming languages including C and Java do not support any language feature to declare critical sections. For this reason, race condition detection techniques first identify the critical sections from a target program. Depending on the assumptions used for identifying critical sections, race condition detection techniques are categorized into two types: one type is called datarace detection, and the other is atomicity violation detection. For each type of race condition detections, many algorithms to check atomicity of critical sections have been introduced. In this section, we first survey the approaches and techniques for identifying critical sections, and then various techniques for datarace detections and atomicify violation detections.

Defining critical sections

Identifying which parts of codes are written as critical sections is not trivial in C and Java programs. Each race detection technique deploys its own technique to define critical sections based on the properties of target program domains. The accuracy of a race condition detection technique highly relies on the accuracy of critical section definitions. If a technique regards much code as critical sections than actual, the techniques may report false positives. In opposite, false negatives would be resulted if a technique regards less code as critical sections than actual.

One approach to define critical sections is regarding every shared variable access as a critical section. This approach is naive but most widely accepted in race condition detection techniques. The race condition detection techniques with this particular approach are called as datarace detection techniques. These techniques define a data race as a situation where a shared variable is accessed by at least two different threads concurrently where at least one access is writing. This approach is neither sound nor complete but its usefulness is proven by many techniques for finding programming errors by missing proper lock operations.

The other approaches assume that a critical section consist of a consequence accesses rather than a single access. The race detection techniques based on this assumption are called as atomicity violation detection techniques. There are mainly three approaches to define a sequence of accesses as a critical section for atomicity violation detection techniques. First approach is to expect users to specify code blocks intended to be critical sections. Atomizer [36] and Velodrome [25] allow users to specify code blocks as critical sections using additional keyword atomic.

Second approach is to employ a certain type of existing code blocks as critical sections. In [37], the technique assumes that each method is intended to be a critical section. In [38], the technique assumes that a code block guarded by a synchronize construct as an atomic code block. In [39], non private methods, synchronized private methods, synchronized blocks inside non-synchronized private method except for main method, and thread spawning methods are regarded as critical sections.

Third approach is to infer the critical sections by analyzing data dependency of a target program code. This approach is motivated by the observation that atomicity violations might occur if two statements with data dependency does not belong to a critical section. In [40], the technique inputs user given dependency relations among object members and then check whether dependent members are manipulated without possibility of atomicity violations. The technique reports a possibility of atomicity violation if any two dependent members are not updated in a critical section. MUVI [41] infers data dependency in a program by a heuristic algorithm which estimates the presence of data dependency by code distance information. Based on inferred dependency information, the technique reports a possibility of atomicity violation if the shared variables of a dependency are not updated consistently in a program. In [42], the technique detects a specific type of atomicity violations called stale-value concurrency error. This technique analyzes Java programs to find methods with two adjacent synchronized blocks where later one depends on former one via local variables. The technique reports a possibility of atomicity violation since the executions where context-switching may occur between two synchronized blocks. AtomRace [20] has the code patterns of critical sections. This tool infers critical sections from a given code by pattern-matching the code patterns.

Checking race condition of critical sections

There are two major categories in the techniques for checking atomicity of critical sections with respect to critical section definitions. First categories targets on datarace detection. The techniques in the other categories aim to detect atomicity violations. Datarace detection technique concentrates on finding two unsynchronized accesses to a shared variable which can be executed simultaneously. Atomicity violation detection techniques find two critical sections which can be executed concurrently and apply additional techniques to verify these two critical sections are executed atomically for any interleaved execution. We first survey data race detection techniques and then atomicity violation detection techniques.

Datarace detection techniques

In datarace detection techniques, datarace error is defined by the following three conditions :

- There exists two accesses where at least one access is writing, and
- These two accesses manipulate the same shared variable (or memory location), and
- These two accesses can be executed concurrently.

These conditions are deployed in various ways depending on the purpose of datarace detection techniques.

In dynamic analysis techniques, it is possible to check first condition for any two accesses since the extract exact memory location of an access is available. However, in static analysis techniques, it is difficult to check whether any two statements would access the same variable in real executions. Many techniques over-approximate to check whether two statements access the same variable or not using type information, escape analysis, and points-to analysis.

In order to check third condition, many dynamic analysis techniques compute happen-before relation of each access in a monitoring execution. For an execution with happen-before relation, two accesses without happen before ordering are considered to be executed concurrently. This technique never results false positive if happen before relation is correctly measured. Vector clock is widely used for computing happen before relation.

The other widely used technique is lockset algorithm which checks a sufficient condition of third condition. This detection technique is motivated by a programming idiom that every statement which accesses a shared variable is guarded by one lock consistently over a program. This programming idiom is called lock discipline and widely used in lock based concurrent programs. Lockset algorithm tracks the set of locks held by the current thread at an execution point to check whether there exists at least one lock which is consistently held when a thread accesses the shared variable. Since lockset algorithm adopts a sufficient condition of absence of datarace, many techniques using this algorithm investigate additional false positive filtering techniques for increasing accuracy of results.

There exist approaches which employ both happen before relation and lockset algorithm to improve dynamic datarace detection techniques. The static analysis technique such as Chord uses neither happen-before relation nor lockset algorithm. This technique first statically find a set of candidate datarace accesses pairs from a Java program, and then check whether two candidates accesses can be executed concurrently by various static analysis techniques including reachability analysis, thread-escape analysis, and lock analysis.

Eraser [26] infers lock discipline from a multithreaded C program execution to check whether the program has data race bugs. A program has a lock discipline if every shared variable is consistently synchronized by at least one lock. Eraser understands the absence of lock discipline of a shared variable means that two concurrent accesses to the shared variable can result data race in an execution of the program. In order to infer lock discipline, the tool records each variable's lockset, a set of locks held at a time of the variable accessing through an execution. Eraser tool uses ATOM binary modification system to instrument a binary program to execute the lockset algorithm in run time. Eraser tracks a lockset for every 32-bit sized memory locations of global variables and heap variables.

MultiRace [31] detects data race from executions of C++ programs using Djit+ algorithm which is based on happen-before execution model and an improved lockset algorithm. Considering memory characteristic of object oriented programs, the memory granularity in data race detection is the size of objects rather than fixed size (double words in Eraser). This technique aims to reduce runtime detection cost by using region trap library instead of using information extraction by inserting probes.

Racer [43] is a dynamic data race detection tool for Java program. It adopts Eraser's lockset algorithm with the modified state machine suitable for Java memory model. And the authors use aspect-oriented programming concept for information extraction. The functionality of aspect is extracting runtime information and the authors define three point cuts: locking points, unlocking points, accessing variables suspected to be shared ? where the aspects are added.

RCAnalyzer [44] finds shared variables which have conflicting accesses (read accesses with at least one write access) from a given program and then check whether a lock is consistently held at every access to each shared variable. RC-Analyzer is implemented upon Bauhaus infrastructure which supports various pointer analysis and control flow analysis techniques. RCAnalyzer handles inaccuracy of each analysis by ranking each warning. This approach is similar to our approach in the point that the technique first finds a pair of conflicting codes and then checks whether two codes can be executed in parallel by lock analysis. But, RCAnalyzer indicates candidate bugs by inferring lock discipline, however in our approach, we indicates candidate bugs by pattern matching.

Chord [45] statically analyzes Java programs to find data race bugs. Chord first set the set of conflicting accesses as a set of candidate bugs. And then it refines the set of candidate bugs using alias analysis, thread-escape analysis, call-graph construction, and lock analysis. Chord considers the execution semantics of thread creation, thread join, and lock synchronization. The criteria of Chord are similar to our approach. Chord targets for open programs and results counter examples. Chord concentrates on finding data race bugs meanwhile our approach is to find general concurrency bugs including atomicity violations. Chord is built upon Soot Java optimization framework.

RacerX [16] checks the presence of consistent lock discipline for each shared variables. RacerX traverse the inter-procedural control-flow graph in flow-insensitive manner while it maintains the held lock in the path of traversal. RacerX updates lockset of a shared variable whenever it reaches the accessing statement in the traversal. It reports an error when there is no consistent lock discipline for a shared variable. In order to reduce the false alarms caused by inaccurate lockset analysis, RacerX uses many heuristics in lockset analysis and bug ranking.

Choi et al. [29] suggest a method for efficient dynamic data race detection using preceding static analysis. The static phase of this technique indicates only necessary points to monitor using weaker-than relations. The definition of datarace in this technique is (1) the two accesses are to the same memory location and at least one of them is writing, (2) the two accesses are executed by different threads, (3) the two accesses are not guarded by a common synchronization, and (4) there is no enforced execution orderings such as ordering by thread start and join operations.

Atomicity violation detection

After the critical sections in a program are recognized, the atomicity violation techniques checks whether each critical section is executed atomically or not. There are mainly three approaches to check the atomicity of a code block. First approach is based on type system check whether a program follows a safe guaranteed manner or not. Second approach is to check the absence of interference in an execution of an atomic block. Third approach is to check the serializability of atomic block executions. The goal of atomicity violation checking is to check whether execution of atomic block starting from any state always result the same state with all possible interference of concurrent threads. However, this problem is NP complete. Therefore each approach investigates a strong but useful condition to ensure atomicity.

Atomicity is also important issue in transaction control in database systems, so that researchers of database system have tried to solve the same problem. Actually, the atomicity violation checking is rediscovery of concurrency control techniques for database transactions in the programming language point of view. First approach corresponds to two phase locking in transactions. Second approach corresponds to serial scheduling techniques. And Third approach corresponds to conflict-serializability analysis techniques.

- Checking safe locking patterns

Two interleaved executions are equivalent if two there reaching (final) states are the same with respect to a starting state. Using Lipton's reduction theory, we can check the equivalence of two interleaved executions. If an interleaved execution is reducible to a serial executions where once atomic block is started, only the execution of the atomic block should appears until the atomic block is finished, we can say that the interleaved execution does not violate atomicity. Atomizer verifies an interleaved execution can be reducible to one of serial executions. In general, Lipton's reduction results many false positives. Atomizer introduces a modified reduction theory by using lock discipline information. Atomizer assumes that a program which has a consistent lock discipline for every shared variables and uses the lock discipline information in the atomicity violation checking. However, this technique only accepts a program with two phase locking patterns. Therefore, it restricts synchronization patterns.

- Checking exclusive execution of atomic blocks

In [38], the authors define a view as a set of variables which is synchronized by a lock at least once in a program. And then, the technique searches view consistency. View consistency is a situation that there exist two variables in a view which is accessed in different two synchronization blocks. AtomRace dynamically monitors an interleaved execution. AtomRace reports when it observes a situation that one thread is executiong an atomic block and the other thread executing an atomic block is scheduled before the thread finishes the execution of atomic block.

- Checking serializability checking

Serializability holds for critical sections if every interleaved execution of the critical section result the same as a serialized critical section execution. The techniques in the previous approaches restrict interleaved executions to avoid race condition. However, well-designed critical sections allow as many interleaved execution as possible. Therefore, serializability is much appropriate condition to check the atomicity violation. Since checking serializability for an arbitrary code is undecidable, many techniques adopt conflict-serializability, a strong condition of serializability. An interleaved execution is conflict-serializable if there exists a serial execution where the orders of conflicting accesses are the same. In conflict-serializability, it allows interleaving of two atomic code block executions as long as the orders of conflicting variable accesses are the same in the two threads. Two methods are widely used for the decision procedures: one is conflict-graph, and the other is accessing patterns. Velodrome [25] maintains a conflict graph using happen before relation of an execution. Liqiang Wang and Scott D. Stoller introduce an algorithm [39] to check conflict-serializability and view-serializability using commit-nodes which is similar to conflict graph. They also propose an algorithm to validate conflict-serializability based on execution patterns in [37].

Chapter 3

Concurrency Bug Classification

3.1 Linux File System

Linux kernel is a large multithreaded program. Each kernel thread is either executing kernel code or executing user application. A thread with a user application can execute kernel code as the application program invokes a system call. When a user program invokes a system call, the thread which executes the user application program will execute the corresponding system call handling function. Since user applications cannot allow to access kernel memory spaces, there is no possibility to occur concurrency error with a kernel program execution and a user application program execution. There are two ways for a thread to execute kernel programs. One ways is that a kernel program creates a kernel thread for handling a special task such as garbage collector or timer. The other way is that a user application program invokes a system call. In this case, the thread which executes the user application program will temporary executes the corresponding system call routine.

In traditional Linux kernel, only one system call routine can be executed at a given time in order to avoid concurrency errors. Big kernel lock which is acquired by `lock_kernel` and relased by `unlock_kernel` has been used for the atomic execution of each system call execution. However, from Linux 2.5, concurrent execution of system call has been allowed. Many synchronization mechanisms are used to synchronize concurrent accesses to shared resources. In most cases, an object has its synchronization mechanism as its members.

There is the following synchronization mechanisms commonly used in Linux file systems. Each synchronization mechanism has its own purpose, implementations and usages.

- Binary lock
  A binary lock is used for guarantee mutual exclusive accesses of a set of resources. A binary lock may synchronize a set of variables or a set of data structures. A binary lock can be implemented as a spin lock or a semaphore whose counter is one. Before processing a shared variable, the accessing thread acquires the protecting lock of the shared variable. After the processing, the accessing thread releases the protecting lock. In many cases, operations on a set of shared variables are integrated into a function. Then, a thread which invokes the function is asked to acquire the proper protecting locks before the function invocations.

- Readers writer lock
  This synchronization mechanism allows concurrent read accesses on a shared data structure by multiple threads but mutually exclusive write access on the shared data structure. There exists a library which support readers/writer lock interface. However, kernel developers implement the readers/writer lock using two binary locks and a counter.

- Atomic instructions
  Atomic operations support cheap synchronization mechanism. This operations are used for frequently accessed shared variables such as reference counters. There are library functions which execute machine instructions for atomic operations. The following operations execute two operations atomically.

- Barriers
  Linux support memory barriers as library functions. A barrier operation

enforces the explicit operation ordering in compiled code. Concurrency behaviors might differ depending on the instruction execution orders in a thread. The execution order for operations on flag variable and count variables are sensitively intended so that disordering in compiling might result incorrect programs. Barriers are used for avoiding this type of errors.

- Thread operations
  Linux kernel programs can create a new kernel thread and communicate to the thread via thread operations. These thread opereations are supported as library functions. It is possible to create a thread using `kthread_create` and `kthread_run` functions. And `kthread_stop()` is to join on a target thread after sending killing signal to the thread.

3.2 Survey of Reported LFS Bugs

We review bug reports to understand real concurrency bugs in system programs. We search the patch information which is related to concurrency issues from Linux file system codes. There are two reasons for targeting bug reports from Linux file system. First, Linux file system use many concurrency features. Second, patch information is available via Linux change log documents.

We searches Linux change log from kernel 2.6.1 to 2.6.28 using keywords. The keywords `file system`, `vfs`, `ext`, `fs` are used for collecting patches for file system related parts. And the following keywords are used for collecting concurrency related patches: `deadlock`, `livelock`, `race`, `concurrent`, and so on. Since Linux change logs are written in plain text, we write a simple script program to finding the patch which contains the keywords of interest. Within more than 200 Linux change log documents, we found that 3And within the file system related patches, we found that about 40 patches are directly related to both concurrency issues and file systems.

We chose Linux 2.6 as the domain of survey for the reason that Linux 2.6 is changed from Linux 2.4 (Linux 2.5 is unofficial) for supporting full concurrency. We expect that many unrevealed concurrency bugs are issued as the kernel becomes supporting full concurrency.

Linux kernel supports a bug reporting system, Bugzilla. We did not rely on the bugzilla, since not all of bug finding and fixings are reported through the bug reporting system. However, we do agree that bug should be systematically managed.

3.3 Classificaion According to Five Aspects

We construct the classification of concurrency bugs in order to assist systematic understanding of concurrency bus. The classification is developed based on based on the observations from the survey result. We analyze the bugs from the survey result by observing the descriptions in the bug reports and code changes. We found that the bugs have the following five aspects: Symptom, Fault, Resolution, Synchronization primitives, and Synchronization granularity.

- Symptom
  The symptom of a bug is a candidate state of an error execution casued by the bug. There are 4 attributes in this aspect: Data race resulting machine exception, Data race resulting faulty states, Deadlock, and Livelock. Data race results invalid state in an execution so that it is observed in either machine exception or faulty states of the internal data structure. As explianed in the chapter 2, deadlock result threads not to be progressed due to the blocking. Livelock results that the related threads are not progressed or delayed due to the infinite loop or scheduling problem.

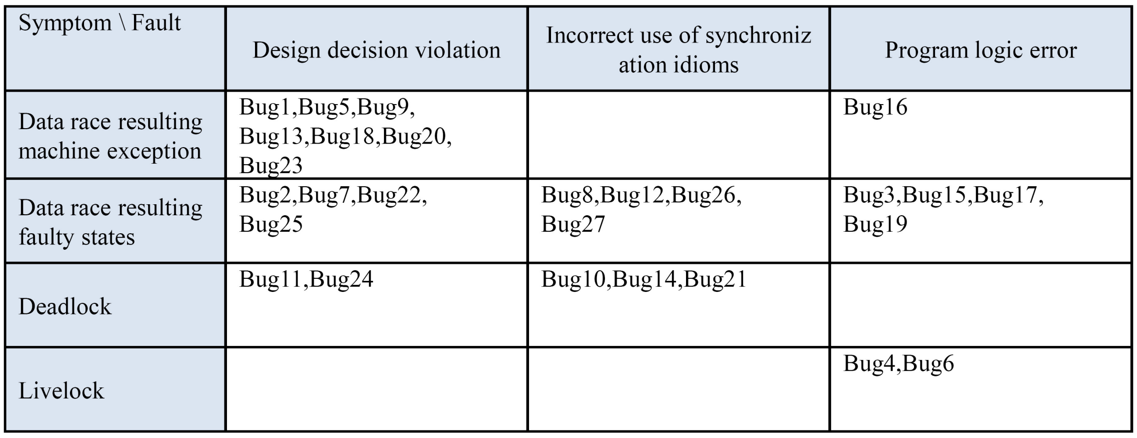
- Fault
  This aspect aims to represent the cause of a bug. There are 3 attributes

in this aspect: Design decision violation, Incorrect use of programming idiom, and Program logic error. Three attributes are corresponding to three level of programming. Design decision violation refers to a bug where programmers do not follow high-level programming criteria. Incorrect use of programming idiom refers to the situation that programmsers mistakenly adopt programming idioms. Program logic error indicates a situtation where the low-level program logic is incorrectly implemented.

- Resolution
  This apsect reprsent that by what program changes a bug is alleivated in the bug patch. There are 2 orthogonal elements which consist of an attribute. First attribute represents the actions : Insert, Remove, Change, and Reorder. Second attribut represents the type of target operations: Synchronization operation, Normal operations, Both. A bug's patch may consist of several code changes. Then the resolution of the bug represent the most significant code change.

- Synchronization primitives
  This aspect of a bug represent the synchronization primitives which contribute to the bug. This aspect has 6 attributes: Instruction, Barrier, Thread operation, Binary Lock, Complex Lock, Semaphore. Instruction refers that a shared variable is used as flag or counter to control concurrency in a bug. Complex lock refers to the lock with derived semantics. This attribute include releasing-on-block semantics, nested locking, try-lock, etc.

- Synchronization granularity
  The synchronization granularity of a bug is decide by the related shared data structure. This aspect has 4 attributes: Kernel-level, File system-level, File-level, and Inode-level. Kernel-level refers that the data structure exists in a kernel level so that the bug may effect to all kernel area.



Symptom \ Fault	Design decision violation	Incorrect use of synchroniz ation idioms	Program logic error
Data race resulting machine exception	Bug1,Bug5,Bug9, Bug13,Bug18,Bug20, Bug23		Bug16
Data race resulting faulty states	Bug2,Bug7,Bug22, Bug25	Bug8,Bug12,Bug26, Bug27	Bug3,Bug15,Bug17, Bug19
Deadlock	Bug11,Bug24	Bug10,Bug14,Bug21	
Livelock			Bug4,Bug6

Figure 3.1: Classification with respect to symptom and fault

According to the five aspects, we derived the attributes for each aspect and then classify 27 Linux file systems bugs from the survey respectively. Figure 3.1 and Figure 3.2 depicts the distribution of the bugs with respect to the aspects. The description of the 27 bugs and their classification results are summarized in the Appendix.

Chapter 4

Concurrency Bug Pattern

Bugs are usually caused by programmers' misunderstandings in programming language, library usages, or design decisions. One mistunderstanding can be shared by various programmers, so that various parts written by the programmers may contain similar bugs. If we find a bug in a program, there is high change to other unrevealed bugs caused by the same reason exist as the found bug in the other part or other version (later modification) of the program. Therefore, bug patterns which capture characteristics of bugs can be utilized for bug detection in large and legacy program development.

In this research we characterize 10 concurrency bugs patterns from the previously reported concurrency bugs of Linux. And then we build the bug pattern detectors of 10 patterns for automatic bug detection. Using the bug detection techniques, we found 8 unreported concurrency bugs from recent Linux kernel. This bug detection technique do guarantee neither soundness nor completeness of analysis result. However, the approach to find bugs using bug patterns is valuable since the bug pattern matching can enable fast, intuitive and convenient bug findings. This technique is cost-effective since it can be applied in early stage of program development and also it do not require any manual effort from end-users.

The following subsections introduce 10 concurrency bug patterns. And then we report the bug detection result by applying the bug patterns in recent Linux kernel. We expect that the bug patterns can be useful for both developing automatic bug detection by computer and code inspection process by human.

```
int data; // shared data

int f(){
  // false most of time
  if (test(data) == true) {//test
    lock_global_lock() ;
    if (test(data)==true)//test&set
    data = newvalue ;
    unlock_global_lock() ;
  }
}

          /* Correct code */
```

```
int data; // shared data

int f(){
  // false most of time
  if (test(data) == true) {//test
    lock_global_lock() ;
    data = newvalue ;
    unlock_global_lock() ;
  }
}

          /* Buggy code */
```

Figure 4.1: Canonical form of Misused Test and Test-and-Set

4.1 Studied Bug Patterns

4.1.1 Misused Test and Test-and-Set

Test and Test-and-Set is a well-known locking idiom. This idiom aims to reduce runtime cost by avoiding the executions of locking operations by dirty reading shared variables. In code optimizations, test and test-and-set pattern would be applied for simple locking codes. However, programmers may make mistake when they apply the pattern. Programmers mistakenly forget to write the second test after lock acquiring. This mistake results a data race bug. In Linux 2.6.11.1, a bug of this pattern is reported from ext3 file system. The programmer misused test and test-and-set for optimization purpose. The code is as follow.

```
int data       // shared data
int  func() {
      lock(m) ;
      write(data) ;
      unlock(m) ;
        …
      lock(m) ;
      /* no write(data) here */
      read(data) ;
      unlock(m) ;

          /* Correct code */
```

```
int data       // shared data
int  func() {
      lock(m) ;
      write(data) ;
      unlock(m) ;
        …
      lock(m) ;
      /* no read(data) here */
      unlock(m) ;

          /*  Incorrect code  */
```

Figure 4.2: Canonical form of Unlock before I/O operation bug pattern

4.1.2 Unlock before I/O operations

I/O operations are incomparably slower than memory operations. If a thread which holds a lock executes I/O operations, other threads which acquire the same lock cannot be executed until the thread releases the lock after all I/O operations. In order to increase utilization, a function intentionally releases its holding lock before I/O operations and then acquires the locks again to finish its work. However, in many cases, programmers may assume that a code block originally guarded by a pair of lock acquiring and releasing would be executed as if no context-switching happens while its execution.

We found that the original programmer of `__sync_single_inode()` considered the possibility of data race and write additional code to handle this situation in `__sync_single_inode()`. The one who modified the `__sync_single_inode()` at Linux kernel 2.6.6, did not consider the case. However, this bug was recognized and fixed in Linux kernel 2.6.7.

4.1.3 Unintended Big Kernel Lock releasing

Big kernel lock is one lock shared in whole Linux kernel for synchronization. Big kernel lock is unordinary semantics called Releasing-on-block. With releasing-on-block semantics, big kernel lock is preempted when a thread which holds big kernel lock gets into waiting state. And when the thread is awakening, it first acquires big kernel lock again and then goes on the execution. This semantics is to avoid situations where a thread with holding big kernel lock becomes waiting. This situation may cause deadlock or severe performance degradation since every system calls had relied on synchronization by big kernel lock in earlier Linux kernel. However, this semantics may cause data race since the atomic execution of a code block guarded by big kernel lock is not guaranteed. Programmers would assume that no other code block guarded by big kernel lock can be scheduled while a code block guarded by big kernel lock is executing. But this assumption is not true if a code block guarded by big kernel lock executes a possibility waiting operation which can make the current thread waiting. Most memory management operations such as `kmalloc()` and `kmem_cache_alloc()` can result waiting state. Linux kernel 2.6.24 includes a patch which correct a data race bug caused by this reason.

4.1.4 Unsynchronized data passing to child thread

A parent thread can transfer data to its child thread by passing a reference to a shared variable. In most case, parent threads write data on shared variables. But, parent thread may write data after thread creation. In such case, proper synchronization is necessary to avoid a scheduling which executes the child thread right after the thread creation. However, programmers do not consider this possible scheduling scenario and do not program proper synchronization. A bug of this type was located at GFS2 file system in Linux kernel 2.6.23.

```
int data;          //shared data
                     protected by BKL
int f(){
   kmalloc() ;     //may release BKL
   lock_kernel(); // BKL acquire
   data = newvalue;
   …
   assert(data == newvalue);
   unlock_kernel(); //BKL release
}

            /* Correct Code */
```

```
int data;          //shared data
                     protected by BKL
int f(){

   lock_kernel(); // BKL acquire
   data = newvalue;
   kmalloc() ;    // may release BKL
   //A context-switch might occur
   assert(data == newvalue) ;
   unlock_kernel(); //BKL release
}
            /* Incorrect Code */
```

Figure 4.3: Canonical form of Unintended Big Kernel Lock releasing

```
int child(int * arg) {
    read(*arg) ;
        …
}
int  func() {
   int arg ;
   thread_create(child, &arg) ;
   /* no write to arg */
        …
}

            /* Correct code */
```

```
int child(int * arg) {
    /* no locking */
    read(*arg) ;
        …
}
int  func() {
   int arg ;
   thread_create(child, &arg) ;
   /* no locking */
   write(arg) ;
}
            /* Incorrect code */
```

Figure 4.4: Canonical form of Unsynchronized data passing to child thread

```
int func1(...) {
               ...
  if (atomic_dec_and_test(x)) {
               ...
  }
               ...
}


int func2(...) {
               ...
  if (atomic_dec_and_test(x)) {
               ...
  }
               ...
}
          /* Correct code */
```

```
int func1() {
               ...
  if (atomic_dec_and_test(x)) {
               ...
  }
               ...
}


int func2() {
               ...
  if (atomic_read(x) == 1) {
    atomic_dec(x) ;
               ...
  }
               ...
}
          /* Incorrect code */
```

Figure 4.5: Canonical form of Use atomic instruction in non-atomic ways

4.1.5 Use atomic instructions in non-atomic ways

Linux supports atomic type variables and atomic operations which manipulate atomic type variables. Test-and-set instructions are supported as atomic operations such as `atomic_test_and_dec()` and `atomic_dec_and_lock()`. Atomic type variables are widely used for the purpose of counters. Use of atomic instructions are desirable since the time-cost of atomic instruction operation is much less than other synchronization mechanisms. However, the program may result unintended behavior unless atomic variables are manipulated consistently. Programmers mistakenly use two seperate atomic instructions to code test-and-set whereas `atomic_dec_and_test(x)` or `atomic_inc_and_test(x)` is used in other parts of the program, it may result atomicity violations.

4.1.6 Waiting already finished thread

Linux kernel supports thread operations as library functions. `kthread_stop()` function is to signal a thread to stop, and then wait until the signaled thread is

```
int parent(...) {
  th = kthread_run(child, arg);
  ...
  kthread_stop(th) ;
}


int child(arg) {
               ...
  while(!kthread_should_stop())
  {
    ...
    if (err)
       continue ;
    ...
  }
}
          /* Correct code */
```

```
int parent(...) {
  th = kthread_run(child, arg);
  ...
  kthread_stop(th) ;
}


int child(arg) {
               ...
  while(!kthread_should_stop())
  {
    ...
    if (err)
       break ;
    ...
  }
}
          /* Incorrect code */
```

Figure 4.6: Canonical form of Waiting already finished thread

finished. Therefore, whenever one thread invokes `kthread_stop()` to stop the other thread, the other thread must not be finished. A thread which invokes `kthread_stop()` will indefinitely wait if the target thread already finish its execution. In order to avoid this situation, a thread designed to be signaled by `kthread_stop()` must not finish its execution until `kthread_shoud_stop()` returns true. However, if there is an execution path where a thread execution is finished without checking `kthread_shoud_stop()`.

The correct code in the canonical form is a conventional programming pattern for parent and child threads. However, the child thread of incorrect code can be be finished without checking `kthread_should_stop()` if `err` is satisfied.

```
int wait_function()
{
  while (atomic_read(x)) {
    ...
    memory_barrier() ;
  }
}

int notify_function()
{
  ...
  atomic_write(x) ;
  memory_barrier() ;
  ...
}
        /* Correct code */
```

```
int wait_function()
{
  while (atomic_read(x)) {
    ...
   /*no memory_barrier()*/
  }
}
int notify_function()
{
  ...
  atomic_write(x) ;
  memory_barrier() ;
  ...
}

        /* Incorrect code */
```

Figure 4.7: Canonical form of Busy-waiting on atomic variable without memory barrier

4.1.7 Busy-waiting on atomic variable without memory barrier

Handshaking between two thread can be implemented via busy-waiting: One thread busy-waits on a shared variable until the other thread assigns a specific value to the shared variable. In symmetric Multi-processor circumstances, these two thread might be executed on two different processors with two different memory caches. Busy-waiting thread should contain at least one memory barrier to enforce cache update for considering this circumstance. Unless, it may iterate unnecessary for reading out-of-date value in a CPU cache, so that it may decrease the performance. This pattern recognizes memory barriers for memory barrier library functions(e.g. `smp_mb()`), yield functions(e.g. `sched()`), and lock operations.

```
int func1() {
  v = malloc() ;
  v->field = x ;
  memory_barrier() ;
  list_add(v->list, shared_list);
}

int func2(){
  ...
  list_read(shared_list) ;
  ...
}

/* Correct code */
```

```
int func1() {
  v = malloc() ;
  v->field = x ;
  /* no memory barrier */
  list_add(v->list, shared_list);
}

int func2(){
  ...
  list_read(shared_list) ;
  ...
}

/* Incorrect code */
```

Figure 4.8: Canonical form of No memory barrier after object initialization

4.1.8 No memory barrier after object initialization

The execution order of two instructions without dependency might be changed by CPU and compiler optimization. However, the execution order of a dynamically allocated variable initialization and its linking to a shared data structure (in this case linked list) must be strict although there is no visible dependency. The absence of any memory barrier between the initialization and the linking to the linked list may result re-ordered execution. And it may cause race condition where other concurrent threads can read uninitialized value. This bug pattern captures the specific situation where the object is a dynamically allocated list element.

```
int func1() {
  lock(m) ;
  ...
  unlock(m) ;
  wait(c) ;
  lock(m) ;
  ...
}

int func2(){
  lock(m) ;
  ...
  unlock(m) ;
  ...
  complete(c) ;
}
```

/* Correct code */

```
int func1() {
  lock(m) ;
  ...
  wait(c) ;
  ...
}

int func2(){
  lock(m) ;
  ...
  unlock(m) ;
  ...
  complete(c) ;
}
```

/* Incorrect code */

Figure 4.9: Canonical form of Waiting with lock held

4.1.9 Waiting with lock held

Linux supports conditional variable synchronization mechanism as `complete` type variable and `complete()` and `wait_for_complete()` functions. The execution of a thread which invokes `wait_for_complete()` on a `complete` variable is blocked until the corresponding thread invokes `complete()`. There is deadlock possibility if a thread waits on a complete variable while it holds a lock. If the thread to wake-up waiting thread acquires the lock before `complet()` invocation, these two threads reach deadlock situation.

4.1.10 Releasing and re-taking outer lock

A function releases and re-takes a lock which is acquired by its caller for improving performance. This function may result deadlock where the releasing lock is not the most recently acquired lock. For this situation, the lock ordering might be violated for the lock re-taking operation.

```
int func1() {
  lock(outer) ;
  lock(inner) ;
  func2() ;
  unlock(inner) ;
  unlock(outer) ;
}

int func2(){
  ...
  unlock(outer) ;
  ...
  lock(outer) ;
}
```

/* Correct code */

```
int func1() {
  lock(outer) ;
  lock(inner) ;
  func2() ;
  unlock(inner) ;
  unlock(outer) ;
}

int func2(){
  ...
  unlock(inner) ;
  unlock(outer) ;
  ...
  lock(outer) ;
  lock(inner) ;
  ...
}
```

/* Incorrect code */

Figure 4.10: Canonical form of Releasing and re-taking outer lock

4.2 Automatic Bug Pattern Detection

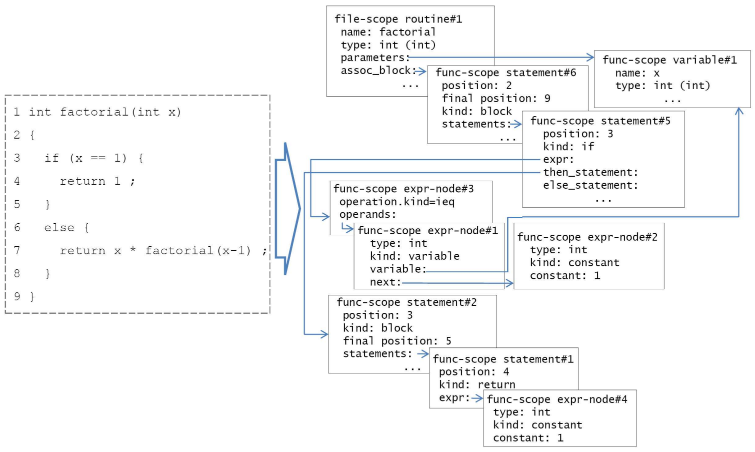
We build the automatic bug pattern detectors for ten bug patterns. The bug pattern detectors are implemented upon EDG C/C++ parser to identify specific abstract syntac tree structure of a target program. The tools demonstrate that bug patterns can be effectively used for finding bugs with simple programming. Using these bug detectors, we found unreported concurrency bugs from a recent Linux kernel source code.

4.2.1 Code analysis using EDG C/C++ parser

In this research, we pattern match target Linux code using EDG C/C++ front end parser [46]. This parser returns the intermediate language of a target code so that we can traverse the target program code as graph like data structure. In this section, we briefly mention how we implement the pattern matching and code analysis algorithm based on EDG C/C++ front end parser.

We employ EDG C/C++ front end parser for program code analysis framework. For the following two reasons, we estimate that EDG C/C++ front end parser has advantage over other tools such as CIL. The one reason we select EDG C/C++ front end parser in the bug pattern matching technique is that it supports various dialect C grammars. Linux program code is written in GNU C grammar. The parser successfully constructs the intermediate language from any Linux code. The other reason for using EDG C/C++ front end is that we can write the algorithms in C/C++ language which is much familiar to many programmers than functional languages like ML.

In abstract syntax tree of target programs, we can access each function, statement, expression, and variable as one data object. Each data object has a set of fields which indicate the type and syntactic characteristic of the data object defined in a target code. And each data object has a set of links to the other data object so that the data objects in a program construct a hierarchical



```
int factorial(int x)
{
  if (x == 1) {
    return 1 ;
  }
  else {
    return x * factorial(x-1) ;
  }
}
```

Figure 4.11: Example of the intermediate language generated by EDG C/C++ front end parser

structure.

The data structure for a function has the members to record its scope, its type of return value, its type of parameters, etc. And also it has the members to indicate a statement object at the entry point of the function. The statement object has information of the first statement of the function, and construct a linked list with other statement objects in the function body using `next` field. And it also points to a set of expression objects. Figure ? depicts the data structures constructed from the example code.

4.2.2 Bug detection to Linux file systems

We apply the constrcuted bug pattern detectors to find concurrency bugs in Linux kernel 2.6.30.4. The tools also can be applied to other parts of Linux. As a primarily step, we first target 9 Linux file systems including both traditional one and recently developed one: Ext-2, Ext-3, Ext-4, NFS, ReiserFS, SysFS, ProcFS, UDF, and BtrFS. The source codes of a file system implementation and virtual file system source codes are merged into one source code. And

Pattern	Ext2	Ext3	Ext4	NFS	Reiser FS	SysFS	Proc FS	UDF	BtrFS	Total
Use Atomic Instructions in Non-Atomic Ways	0	0	0	1	1	0	0	0	3	5
No Memory Barrier After Object Initialization	0	0	0	0	0	0	0	0	7	7
Waiting Already Finished Thread	0	0	0	0	0	0	0	0	12	12
Unsynchronized Data Passing to Child Thread	0	0	0	0	0	0	0	0	3	3
Misused Test and Test-and-Set	13	19	18	15	18	12	18	15	17	145
Busy-waiting on Atomic Variable without Memory Barrier	0	0	0	0	2	0	0	0	6	8
Unlock Before I/O Operations	0	0	0	1	0	0	0	0	0	1
Unintended Big Kernel Lock Releasing	1	0	0	1	4	0	0	2	1	9
Releasing and Re-taking Outer Locks	0	0	1	0	1	0	0	0	0	2
Waiting with Lock Held	0	0	2	0	0	1	0	0	0	3

Table 4.1: Suspected bug reports by the bug pattern matching

then GCC preprocess the source code with a given kernel compile option.

We validate the suspected bug reports from the bug detectors by manual code inspection on the related codes. If a bug report is highly suspected to be real one, we report the issue to the Linux maintainers. BtrFS maintainers confirmed one bug detected by unsynchronized data passing to child thread four bugs detected by joining already finished thread. The later four bugs are caused by the same programming mistakes and then appeared in four different lines. The patch for fixing the bugs was reported to the Linux maintainers.

4.2.3 Bug detection to other parts of Linux

As a furher work, we currently apply the bug dectors to other parts of Linux kernel including device drivers and network stacks. In preliminary experiments to these modules, we report suspected bugs to correpsonding Linux maintainers. And six bugs are confirmed as a realistic by the maintainers. Figure 4.13 summarizes these bugs.

Id.	Location	Module	Related bug patterns
1	Device driver	mtd/ubi	Unsynchronized data passing to child thread
2	Network stack	atm	Misused test and test-and-set
3	Network stack	ax25	Misused test and test-and-set
4	Network stack	rds	Use atomic instructions in non-atomic ways
5	Network stack	netfilteripvs	Use atomic instructions in non-atomic ways
6	Device driver	scsi/qla4xxx	Misused test and test-and-set

Table 4.2: Confirmed bugs from device drivers and network stacks of Linux

Chapter 5

Improved bug pattern matching using semantic information

A syntactic bug pattern matching can find the codes which may results the corresponding bug in executions. However, the detected code may not generate the error in any executions of the program. The possibility of a detected code to actually occur the error depends on the possible contexts where the code would be executed. In syntactic bug pattern matching, the techniques pattern matches a code without considering whole program structure. However, in order to understand possible contexts, the global (whole) programming understanding is unavoidable.

Reducing false alarm is important issue in static analysis. Generally manual bug validation requires as much human labor as manual bug detection. Especially, for concurrency bugs, whole programming understanding is necessary for bug validation which is extremely hard for human code inspection.

For a syntactic bug pattern match, the improved bug pattern matching technique additionally consider (1) a code which can be scheduled in other thread to occur concurrency errors, (2) a set of locks which may be held when the pattern match is executed, (3) a possible shared variables in the pattern match in any executions. And we propose an idea of considering path condition as context.

We developed an improved bug pattern matching for automatic validation of concurrency bugs detected from syntactic bug pattern matching. In this research, the technique is bound to the bug pattern matching. But this technique can be also applied to other static analysis techniques for concurrency bug detection.

The improved bug pattern matching technique does not guarantee any soundness or completeness of bug detection. But, the bug pattern matching results would give users better understanding of bug report and will be helpful for their manual bug validation process. A bug pattern matching result suggests a possible scheduling error scenario with important context information.

5.1 Multiple Code Pattern Matching

In general, concurrency errors do not happen in single thread execution. At least two concurrently executing threads are involved in concurrency bugs. Syntactic bug pattern match only finds a code which may be executed by one thread in a concurrency error execution. A syntactic bug pattern match is false alarm unless there is corresponding code which can be executed by other thread concurrent at a time.

We extend the syntactic bug patterns into the corresponding multithreaded bug patterns. For a syntactic bug pattern, we additionally specify the corresponding code patterns. Syntactic bug pattern is pattern-matched for each function. A multithreaded bug patterns has (1) a set of code patterns, and (2) code pattern sets which may result the corresponding errors in their concurrent executions.

In order to track whether a program location can be reached in a thread execution or not, users should specify the function names which can be started for a thread starting routine. In a closed C program, threads starting routines include main function and interrupt handlers. In an open program such as a Linux file system implementation, users can specify the system call handlers and Linux APIs as threads starting routines.

The detector traverse inter-procedural control-flow graph starting from every function in a given thread starting routine names. First, the detector pattern-matches the major code pattern for each reachable function. If a match is found, the detector traverses the inter-procedural control-flow graph to find the minor code pattern match. If the detector finds a match for the minor code pattern, it reports the pair of the major match and the minor match as a candidate bug.

In inter-procedural control-flow graph, it is necessary to consider dynamic thread creations. In our analysis, the tool recognizes the thread creation function (such as `pthread_create()`, `kthread_start()`) to apply the traversal. However, the following two scenarios are impossible: A major (minor) code pattern match is detected from function a, and then in further traversal from function A, there exists function B which creates a thread with function C. And then a minor (major) code pattern is detected at a function D which is reachable from function C.

These two matches will be never executed concurrently since the synchronization by thread creation. In this research, we do not consider thread join operation which is necessary for much accurate analysis.

5.2 Lock Analysis

Two code pattern matches of a bug pattern cannot be executed concurrently if two code pattern matches are always guarded by a lock. To remove this type of false alarm, the detector traverses the inter-procedural control-flow graph starting from one of thread starting routines while it maintains the lockset. The lock analysis technique computes the set of locks held when a thread starts to execute a code at a particular program location. If there exists one lock which is always held at one program location and also at the other program location, the statements at these two program locations cannot be executed concurrently by the lock synchronization.

Eraser [26] proposes an approach to consider lockset at a program location to understand the concurrent behavior of a program. This approach has been widely adopted by many concurrency bug detection techniques in both static analysis and dynamic analysis. Our lockset analysis technique is similar to the lock analysis of RacerX [16]. RacerX aims to check lock disciplines of shared variables using lockset. But our approach computes the locksets at the starting point of each detected bug for checking whether two codes in a bug pattern can be executed concurrently or not.

The lockset analysis technique traverses the inter-procedural control-flow graph to maintain the lockset at each program locations. The algorithm includes a lock to the set of currently held lock when it reaches a statement known as the lock acquiring statement. In similar manner, the algorithm excludes a lock from the currently held lock set when it reaches a statement known as the lock releasing statement. In general concurrent programs, many distinguishable locks are used through various acquiring and releasing operations. Users should specify which statement is recognized as acquiring (or releasing) operation of which lock. In the lock analysis technique, we investigate reasonable assumption to alleviate lock alias problem.

The detector reports a pair of matches as a bug if there exists two inter-procedural paths starting from thread starting routines to the matching functions and the locksets at the points of matches started are exclusive. The detector reports two paths and the lockset information to users for supporting user understanding.

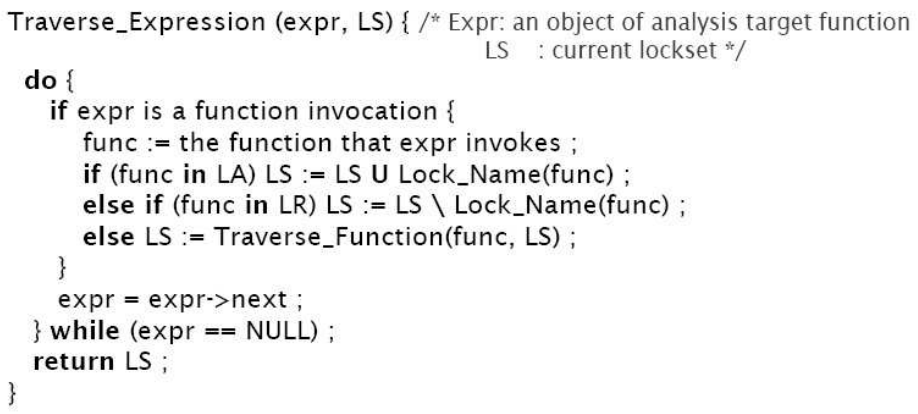
The lockset analysis has two sources of inaccurate results. One is the inaccurate lock alias analysis, and the other is inaccurate lockset analysis. In our approach, we do not distinguish two dynamically allocated locks of a type. In general, computing accurate lockset for a program location is difficult. In our approach, we take a conservative analysis. Inaccurate intra-procedural

```
LockAnalysis(TS, LA, LR, FS) { /* TS: thread spawning functions LA: lock acquiring functions,
                                 LR: lock releasing functions, FS: function summaries */
 foreach func in TS {
   LS := { } ;
   Analyze_A_Function(func, LS) }
}


Traverse_Function (Func, LS) { /* Func: an object of analysis target function
                                  LS    : current lockset */
  if (visit(Func,LS) == true)
    return cache(Func, LS) ;
  if (Func in FS)
    return LS U FSInc(Func) \ FSDec(Func) ;

  stmts := the statement block of Func ;
  Traverse_Stmts(stmts, LS) ;
}
```

Figure 5.1: The lock analysis algorithm for functions

```
Travere_Stmts(Stmts, LS) {  /* Stmts: a statement block object
                               LS     : current lockset */
  stmt := Stmts.stmt ; /* the entrace statement of Stmts */
  do {
    if expr is a branching {
      LS' := { }
      foreach branch in expr's branches {
        LS' := LS' U Analyze_A_Stmts (branch, LS) ; }
      LS = LS' ; }
    else {
      foreach expr in stmt.expressions {
        LS := Analyze_An_Expression(expr, LS) ; }
      stmt = stmt->next ; }
  } while (stmt != NULL) ;
  return LS ;
}
```

Figure 5.2: The lock analysis algorithm for statements



```
Traverse_Expression (expr, LS) { /* Expr: an object of analysis target function
                                    LS   : current lockset */
  do {
    if expr is a function invocation {
       func := the function that expr invokes ;
       if (func in LA) LS := LS U Lock_Name(func) ;
       else if (func in LR) LS := LS \ Lock_Name(func) ;
       else LS := Traverse_Function(func, LS) ;
    }
    expr = expr->next ;
  } while (expr == NULL) ;
  return LS ;
}
```

Figure 5.3: The lock analysis algorithm for expressions

analysis may result inaccurate inter-procedural analysis. RacerX do not allow up-propagation of lockset to avoid this propagation. In our approach, users can additionally specify the effects to lockset as a function summary if necessary. Unless, the detector also ignores up-propagation effects.

5.3 Points-to Analysis

In this research, we apply a simple point-to analysis for reducing false positives by non-shared variables. The enhanced syntactic bug pattern matching find pairs of code fragments expected to result concurrency errors in concurrent executions. Among the code fragments in a detected bug pattern match, there must exist at least one variable (or memory location) shared. In the syntactic bug pattern matching, we assume that all pointers of the same type may point to the same memory location. This assumption is the most conservative may alias analysis criteria. For Linux file system programs, it is difficult to get accurate points-to analysis result for the high complexity of data structures and the limitation of partial code analysis.

By applying simple points-to analysis, we can eliminate false positives

```
static int proc_get_sb
(struct file_system_type *fs_type, …) {
                         …
  ei = PROC_I(sb->s_root->d_inode);
  if (!ei->pid) {
    rcu_read_lock();
    ei->pid = get_pid(find_pid_ns(1,ns));
    rcu_read_unlock();
}
                         …
```

```
static struct inode * proc_alloc_inode
(struct super_block * sb) {
                         …
  ei = kmem_cache_alloc(proc_inode_cachep,
                        GFP_KERNEL);
  if (!ei) return NULL ;
  ei->pid = NULL ;
                         …
```

Figure 5.4: False positive caused by dynamic memory allocation

where a bug involving variable in a code fragment definitely cannot be accessed by the other code fragments. One trivial non shared variable is a dynamically allocated variable whose address is not assigned to any shared variable. We apply a points-to analysis technique to code fragment of a pattern match.

In Linux kernel, dynamic memory address spaces are allocated by calling several library functions such as `kmalloc()`, `zmalloc()`, etc. Using simple points-to analysis, it is possible to check whether a dynamically allocated variable is not shared or may be shared. The statements which access a dynamically allocated non shared variable will not involve any concurrency errors. Many Linux functions allocate a dynamic memory space and access it without any locking until it is registered into a shared data structure. This is correct program, but the initialization access would be detected as a data race candidate since its absence of locking. From Linux kernel 2.6.26, we found a code pattern match of the misused test-test-and-set bug pattern. It seems that `proc_alloc_inode()` function accesses a `proc_inode` object's `pid` field without any locking so that it might be harmful. However, it is not harmful since `ei` in the interesting access is not shared variable. A similar idea is employed to static data race detection tool RELAY for filtering out false positives by initializations.

We assume that the names of functions used for allocating dynamic memory address space are known at the beginning of analysis. The algorithm notices a pointer variable which is assigned as the return value of the memory allocation function calls. We recognize that a newly allocated memory address is non-shared until it might become shared. The following operations are regard to make the related memory address as shared. The address of the memory space is assigned to a shared variable. We regard global variables, heap variables, and their members as shared variables. We apply the points-to analysis inter-procedural manner so that we do not consider the effect of function invocations. The syntactic pattern matching indicates the interesting accesses in a code fragment. This technique is to assure that the variable in the interesting accesses is not non-shared variable. This technique may result false positive since the lacks of context-sensitivity. However, this technique can effectively eliminate obvious false positives from the static bug pattern matching. This technique inputs a target function, an interesting statement, and a type of interesting variable. For a given function, the technique tracks non-shared variables in path-insensitive manner. The algorithm is described in Figure 5.5.

5.4 Experiment

We apply the improved pattern matching techniques to the ten bug pattern detectors. The improvement was done in two steps. For first step, the ten bug patterns are conceptually extended to include multithreaded features and constraints on semantic information. For second step, We newly generate the ten bug pattern detectors considering semantic information. In order to synthesize the improved bug pattern detector, we build a set of templates upon EDG parser. This template supports inter-procedural control-flow graph traversal while it maintains semantic information.

We apply the improved bug pattern detectors to the previously targed 9 Linux file systems. Firgure 5.4 show the number of suspected bug reported

```
Points_to_stmts (Statement start_stmt, Set non_shared,
                   Statement target_stmt, Variable target_var){

  stmt := target_func.entry ;
  non_shared := {} ;

  do {
    if stmt is an assignment operation{
      if stmt.right is dynamic memory allocation {
        non_shared := non_shared U stmt.left ; }
      if stmt.right is in non_shared AND stmt.left is local  {
        non_shared := non_shared U stmt.left ; }
      if stmt.right is in non_shared AND stmt.left is shared {
        non_shared := non_shared \ stmt.left ; }}
    if stmt is a branching {
      foreach stmti in stmt.branches {
        non_shared := Points_to_stmt(stmt.branches, non_shared, target_stmt, target_var);}}
    if stmt is a loop {
      non_shared := Points_to_stmt(stmt.body, non_shared, target_stmt, target_var) ; }
    stmt = stmt->next ;
  } while (stmt != target_stmt) ;

  if target_var is in non-shared { return false ; }
  else { return true ; }
}
```

Figure 5.5: The points-to analysis algorithm

by the improved bug pattern detectors. The number is a paraenthesis is the corresponding result by the previous bug pattern detector. It shows that the semantic information reduces the number of bugs reports from 195 to 80 (41%). This result indicate that the improved pattern matching techniques using semantic information effectively reduce the false alarm ratio in concurrency bug detection.

We had experiments to show the effectiveness of each semantic information in false alarm reducing. For misused test and test-and-set pattern dection, we selectively applied three different types of semantic information. The result on Figure 5.5 show that the multiple pattern matching reduces almost halve of false alarms. The lock analysis upon the multiple pattern matching contributes to reduce additional 20% of false alarms.

Pattern	Ext2	Ext3	Ext4	NFS	Reiser FS	SysFS	Proc FS	UDF	BtrFS	Total
Use Atomic Instructions in Non-Atomic Ways	0	0	0	1	1	0	0	0	1	3
No Memory Barrier After Object Initialization	0	0	0	0	0	0	0	0	0	0
Waiting Already Finished Thread	0	0	0	0	0	0	0	0	6	6
Unsynchronized Data Passing to Child Thread	0	0	0	0	0	0	0	0	1	1
Misused Test and Test-and-Set	6	11	11	6	10	7	4	7	9	65
Busy-waiting on Atomic Variable without Memory Barrier	0	0	0	0	0	0	0	0	0	0
Unlock Before I/O Operations	0	0	0	0	0	0	0	0	0	0
Unintended Big Kernel Lock Releasing	0	0	0	1	3	0	0	1	1	5
Releasing and Re-taking Outer Locks	0	0	0	0	0	0	0	0	0	0
Waiting with Lock Held	0	0	0	0	0	0	0	0	0	0

Table 5.1: Suspected bug reported by the improved bug detection techniques

	Ext2	Ext3	Ext4	NFS	Reiser FS	SysFS	Proc FS	UDF	BtrFS	Total
Syntactic	13	19	18	15	18	18	12	15	17	145
+ Multiple	9	14	14	9	13	11	7	10	14	101
+ Multiple + Lockset	9	11	11	9	13	11	7	10	13	94
+ Multiple + Points-to	6	11	11	6	10	8	4	7	11	74
+ Multiple + Lockset + Points-to	6	8	8	6	10	7	4	7	9	65

Table 5.2: Effect of different combinations of semantic information consideration to misused test and test-and-set bug pattern matching

Chapter 6

Conclusion

In this research, we propose the pattern-driven concurrency bug detection techniques primilary targeting for Linux kernel. The bug pattern dectection technique aims to supplement the lock-based concurrency bug detection techniques by covering the bugs with various synchronization primitives. We develop the bug detection techniques in two steps: First step is defining the bug patterns to detect concurrency bugs, and second step is improving pattern matching techniques.

We study the nature of real world concurrency bugs by surveying previously reported bugs from Linux File System. The study includes the analysis to symptom, fault, and resolution of each bug. As a result of the study, we construct the classification with five attributes to diagnose concurrency bugs.

Based on previously reported bugs, we define ten concurrency bug patterns. These bug patterns are mainly proposed to specify concurrency bugs which are not appropriate to be detected by the conventional lock-based bug detection techniques. And then we compose the bug pattern detectors for the ten concurrency bug patterns. We show that the result of applying these bug pattern detectors to Linux kernel 2.6.30.4.

We improve the bug pattern detection by supplementing semantic information specification in bug pattern definition. The improved bug pattern matching techniques significanlly reduce the false positive ratio with respect to syntactic pattern matching. Based on C parser, we implement the improved bug pattern detectors. The analysis result indicates that the correlating semantic information to pattern matching enhance the correctness of analysis result.

요 약 문

시맨틱 정보를 이용한 패턴 매칭을 통한 동시성 결함 검출

멀티코어 프로세서가 널리 보급됨에 따라 오늘날 많은 소프트웨어 시스템은 동시성 프로그램으로 작성된다. 하지만, 기존의 테스팅 기법과 모델 체킹 기법으로는 운영체제와 같은 상용 프로그램 수준 크기의 동시성 프로그램의 동작정확성 검증이 어려운 실정이다. 이 연구에서는 리눅스 운영체제 커널 소스 코드를 주대상으로 개발한 결함 패턴 매칭 기법을 통한 동시성결함 검출 기법을 소개한다. 데이터 레이스, 교착상태와 같은 동시성결함을 이해하기 위하여, 리눅스 파일 시스템에서 실제 발견되었던 결함들을 조사하였으며, 각 결함을 증상, 원인, 해결방법, 관련 동기화 기법, 동기화 범위에 따라 분류하는 분류체계를 개발하였다. 이를 바탕으로 10개의 동시성결함 패턴을 만들었으며, 패턴 매칭을 통해 결함을 검출하는 프로그램을 작성하여 리눅스 파일 시스템 코드에 적용하였다. 마지막으로, 패턴매칭에 시맨틱 정보를 추가로 고려함으로써 동시성결함 검출의 정확도를 향상 시켰으며, 개발된 기술을 리눅스 파일 시스템 코드에 적용하여 유용성을 확인하였다.

References

[1] E.G.Coffman, M.J.Elphick, and A.Shoshani. System deadlocks. *ACM Computing Surveys*, 3:67–78, 1971.

[2] IEEE. Ieee standard glossary for software engineering technology. *IEEE*, 1900.

[3] Alex Ho, Steven Smith, and Steven Hand. On deadlock, livelock, and forward progress. *Technical Report*, UCAM-CL-TR-633, 2005.

[4] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM.

[5] David Hovemeyer and William Pugh. Finding concurrency bugs in java. In *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[6] Andy Chou, Junefeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, 2001. ACM.

[7] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, Berkeley, CA, USA, 2000. USENIX Association.

[8] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. *Parallel and Distributed Processing Symposium, International*, 0:286b, 2003.

[9] Eitan Farchi and Bradley R. Harrington. Assisting the code review process using simple pattern recognition. In *Haifa Verification Conference*, pages 103–115, 2005.

[10] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Not.*, 43(3):329–339, 2008.

[11] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *ACM SIGPLAN Notices, vol. 35, issues 5 (May 2000)*, pages 219–232, 2000.

[12] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI03)*, 2003.

[13] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2002.

[14] Dan Grossman. Type-safe multithreading in cyclone. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI03)*, 2003.

[15] David Gay, Matt Welsh, Philip Levis, Eric Brewer, Robert von Behren, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, 2003.

[16] Dawson Engelr and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles*, 2003.

[17] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *In Computer-Aided Verification (CAV), LNCS 3576*, pages 82–97. Springer, 2005.

[18] Shaz Qadeer and Dinghao Wu. Kiss: Keep it simple and sequential. In *PLDI*, 2004.

[19] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *In: CAV*, pages 262–274. Springer, 2003.

[20] Zdenek Letko, Tomas Vojnar, and Bohuslav Krena. Atomrace: Data race and atomicity violation detector and healer. In *6th workshop on Parallel and distributed systems (PADTAD)*, 2008.

[21] Patrice Godefroid Bell. Model checking for programming languages using verisoft. In *In Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186. ACM Press, 1997.

[22] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *SPIN*, pages 288–305, 2008.

[23] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.

[24] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Computer Aided Verification*, 2009.

[25] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[26] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transaction on Computer Systems*, 15(4):391–411, 1997.

[27] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *ACM SIGPLAN Conference on Program Language Design and Implementation*, pages 196–205, 1994.

[28] Markus Dahm. Byte code engineering with the bcel api. *Technical Report*, B-17-98, 2005.

[29] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. *SIGPLAN Not.*, 37(5):258–269, 2002.

[30] S.D.Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, 2006.

[31] Eli Pozniansky and Assaf Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs. *Concurrency Computation: Practice and Experience*, 19(3):327–340, 2007.

[32] Cormac Flanagan and Martin Abadi. Types for safe locking. In *European Symposium on Programming*, 1999.

[33] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Detecting potential deadlocks with static analysis and run-time monitoring. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, 2005.

[34] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multithreaded programs. In *Haifa Verification Conference*, volume 3875 of *LNCS*, pages 208–223. Springer, 2005.

[35] Saddek Bensalem, Jean-Claude Fernandez, Klaus Havelund, and Laurent Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *Parallel and Distributed Systems: Testing and Debugging*, 2006.

[36] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Symposium on Principle of Programming Languages*, 2004.

[37] Liqiang Wang and Scott D. Stoller. Runtime analysis for atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32(2), 2006.

[38] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *The First International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS)*, 2003.

[39] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[40] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[41] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, and Weihang Jiang. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *ACM Symposium on Operating Systems Principles*, 2007.

[42] Cyrille Artho, Klaus Havelund, and Armin Biere. Using block-local atomicity to detect stale-value concurrency errors. In *Second International Symposium on Automated Technology for Verification and Analysis*, 2004.

[43] Eric Bodden and Klaus Havelund. Racer: Effective race detection using aspectj. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[44] Aoun Raza and Gunther Vogel. Rcanalyzer: A flexible framework for the detection of data races in parallel programs. In *Ada-Europe*, 2008.

[45] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

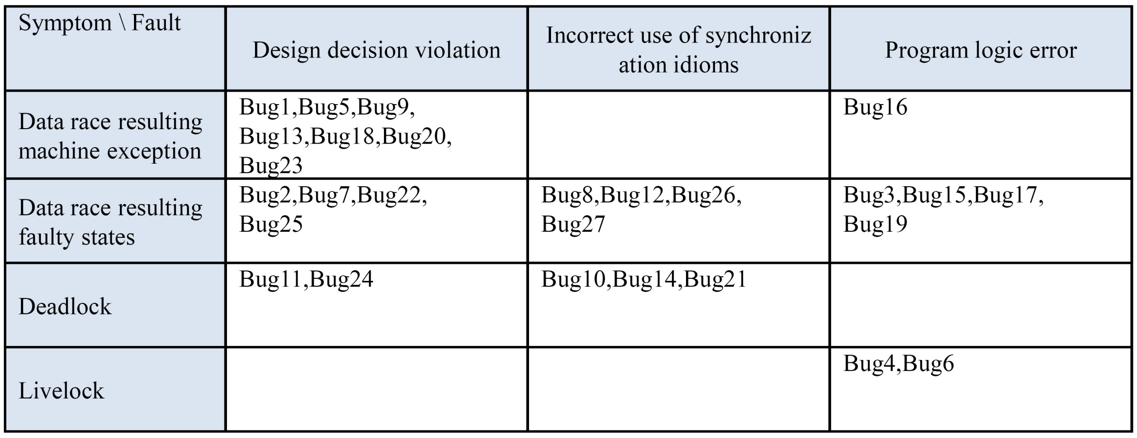
[46] Edison Design Group. The c++ front end. `http://www.edg.com/index.php`.

Appendix

Classification of Concurrency Bugs

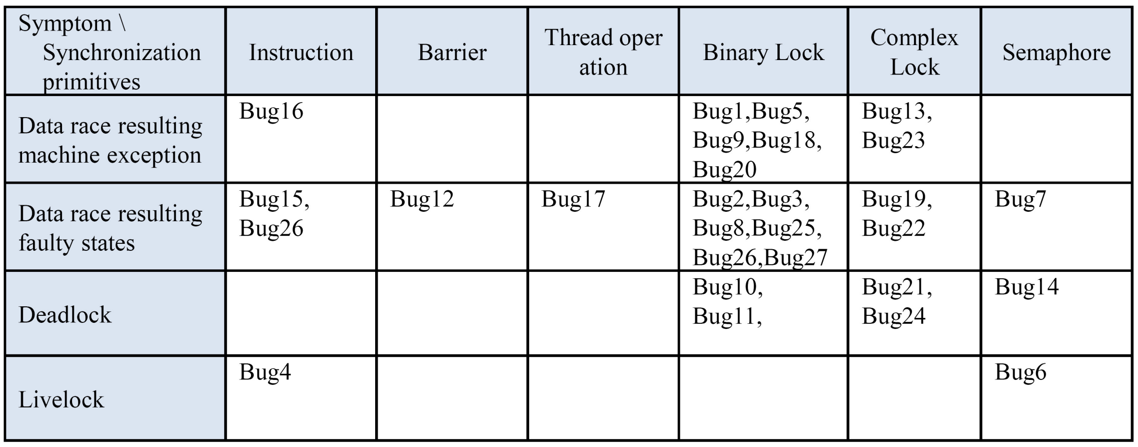
Id.	Version/ Module	Symptom	Fault	Resolution	Sync. primitive	Sync. granularity	Description
1	2.6.2 Proc	Data race resulting machine exception	Design decision violations	Reorder sync. op. and data op.	Binary Lock	Kernel-level	A dentry object access is not guarded by vfsmount_lock.
2	2.6.3 VFS	Data race resulting faulty states	Design decision violations	Insert sync. operations	Binary Lock	Kernel-level	The traversal to directory structure is not protected by read-copy-update lock.
3	2.6.6 VFS	Data race resulting faulty states	Program logic error	Change sync. op- erations	Binary Lock	Kernel-level	Consequent atomic instructions may result data races.
4	2.6.6 VFS	Livelock	Program logic error	Insert data operations	Instruction	Kernel-level	A loop may manipulate an object repeatedly without progressing in worst case.
5	2.6.7 VFS	Data race resulting machine exceptions	Design decision violations	Reorder sync. op. and data op.	Binary Lock	Kernel-level	A global array ioctl_hash_table is not guarded by Big Kernel Lock.
6	2.6.8 VFS	Livelock	Program logic error	Insert data operations	Semaphore	Kernel-level	Atomicity might be broken between two consequent code blocks guarded by the same lock.
7	2.6.10 SysFS	Data race resulting faulty states	Design decision violations	Insert sync. operations	Semaphore	File-level	Before accessing a sysfs_buffer object, its semaphore should be downed.
8	2.6.11.11 Ext3	Data race resulting faulty states	Incorrect use of sync. idioms	Insert control operations	Binary Lock	File System-level	Test-Test-and-Set pattern is mistakenly applied without second test.
9	2.6.11.17 JBD	Data race resulting machine exceptions	Design decision violations	Reorder sync. op. and data op.	Binary Lock	File System-level	An access operation to journal's head field is not guarded by j_list_lock.
10	2.6.15.5 Ext2	Deadlock	Incorrect use of sync. idioms	Insert sync. operations	Binary Lock	Kernel-level	There is an execution path which acquires a lock while it already held the lock.
11	2.6.18 Inotify	Deadlock	Design decision violations	Change data value	Binary Lock	Kernel-level	A lock ordering is conflicted among 5 binary locks.
12	2.6.20.2 VFS	Data race resulting faulty states	Incorrect use of sync. idiom	Insert sync. operations	Barrier	Kernel-level	Bit operations might be unintentionally re-ordered due to the lack of memory barriers
13	2.6.22 Proc	Data race resulting machine exceptions	Design decision violations	Insert sync. operations	Complex Lock	File-level	Necessary get()/put() function invocations are missed.

14	2.6.22 JFFS2	Deadlock	Incorrect use of sync. idioms	Replace data operations	Semaphore	File System-level	There is an execution path which acquires a semaphore while it already held the semaphore.
15	2.6.23.7 NFS	Data race resulting faulty states	Program logic error	Remove data operations	Instruction	File-level	The meaning of the return value of PageUptodate(page) has been changed so that the function might result data race.
16	2.6.24 Proc	Data race resulting machine exceptions	Program logic error	Replace sync. operations	Instruction	File-level	dentry objects' count fields are not correctly manipulated by appropriate atomic instructions.
17	2.6.24 GFS2	Data race resulting faulty states	Program logic error	Replace data values	Thread operation	File System-level	Accesses to a shared variable between parent and child thread are not correctly synchronized.
18	2.6.24 VFS	Data race resulting machine exceptions	Design decision violations	Remove data operations	Binary Lock	Inode-level	A dentry's inode object is not guarded by its i_mutex.
19	2.6.24 VFS	Data race resulting faulty states	Program logic error	Reorder data operations	Complex Lock	Kernel-level	Atomicity of code block guarded by Big Kernel Lock might be broken by sleeping.
20	2.6.27 NFSD	Data race resulting machine exceptions	Design decision violations	Insert sync. operations	Binary Lock	File-level	An access to nfsd_serv objects is not guarded by a lock.
21	2.6.27 VFAT	Deadlock	Incorrect use of sync. idioms	Remove sync. operations	Complex Lock	Inode-level	There is an execution path which acquires a lock while it already held the lock.
22	2.6.27.19 Ext4	Data race resulting faulty states	Design decision violations	Insert sync. operations	Binary Lock	File System-level	An ext4_group_descriptor object's bg_flags field is modified without holding appropriate lock.
23	2.6.27.20 JBD2	Data race resulting machine exceptions	Design decision violations	Insert sync. operations	Binary Lock	Inode-level	An access to jbd2_inode objects' i_transaction fields is not guarded by a lock.
24	2.6.27.22 VFS	Deadlock	Design decision violations	Change sync. operations	Complex Lock	Inode-level	Lock ordering confliction is possible for the two locks acquired by double locking operations.
25	2.6.27.25 Ext4	Data race resulting faulty states	Design decision violations	Insert sync. operations	Binary Lock	Inode-level	i_cached_extet is shared by many threads without any sync.
26	2.6.28 FAT	Data race resulting faulty states	Incorrect use of sync. idioms	Insert sync. operations	Instruction, Binary Lock	Inode-level	The program does not consider 32-bit machine environment where a variable access is decomposed into two.
27	2.6.28 FAT	Data race resulting faulty states	Incorrect use of sync. idioms	Insert sync. operations	Binary Lock	Inode-level	64 bit variable may not be updated atomically without locking in 32-bit machine environments.



Symptom \ Fault	Design decision violation	Incorrect use of synchroniz ation idioms	Program logic error
Data race resulting machine exception	Bug1,Bug5,Bug9, Bug13,Bug18,Bug20, Bug23		Bug16
Data race resulting faulty states	Bug2,Bug7,Bug22, Bug25	Bug8,Bug12,Bug26, Bug27	Bug3,Bug15,Bug17, Bug19
Deadlock	Bug11,Bug24	Bug10,Bug14,Bug21	
Livelock			Bug4,Bug6

Figure 6.1: Classification with respect to symptom and fault



Symptom \ Synchronization primitives	Instruction	Barrier	Thread oper ation	Binary Lock	Complex Lock	Semaphore
Data race resulting machine exception	Bug16			Bug1,Bug5, Bug9,Bug18, Bug20	Bug13, Bug23	
Data race resulting faulty states	Bug15, Bug26	Bug12	Bug17	Bug2,Bug3, Bug8,Bug25, Bug26,Bug27	Bug19, Bug22	Bug7
Deadlock				Bug10, Bug11,	Bug21, Bug24	Bug14
Livelock	Bug4					Bug6

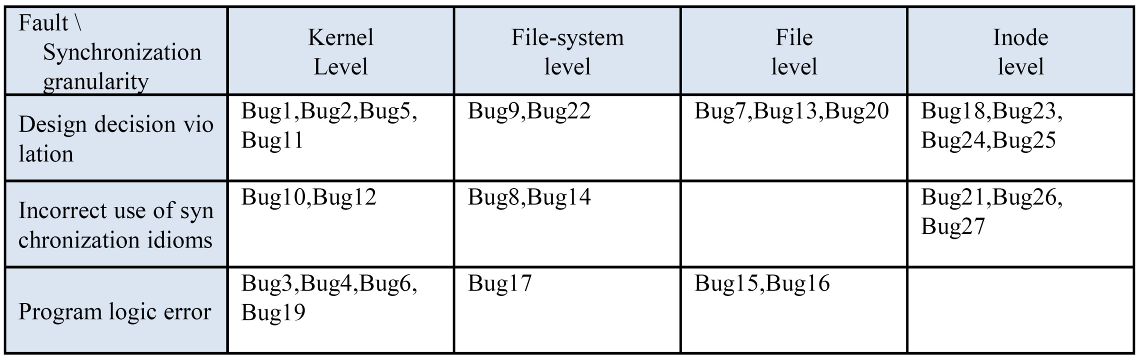
Figure 6.2: Classification with respect to symptom and synchronization primitive

Symptom \ Synchronization granularity	Kernel Level	File-system level	File level	Inode level
Data race resulting machine exception	Bug1,Bug5	Bug9	Bug13,Bug16, Bug20	Bug18,Bug23
Data race resulting faulty states	Bug2,Bug3,Bug12, Bug19	Bug8,Bug17,Bug22	Bug7,Bug15	Bug25,Bug26, Bug27
Deadlock	Bug10,Bug11	Bug14		Bug21,Bug24
Livelock	Bug4,Bug6			

Figure 6.3: Classification with respect to symptom and granularity

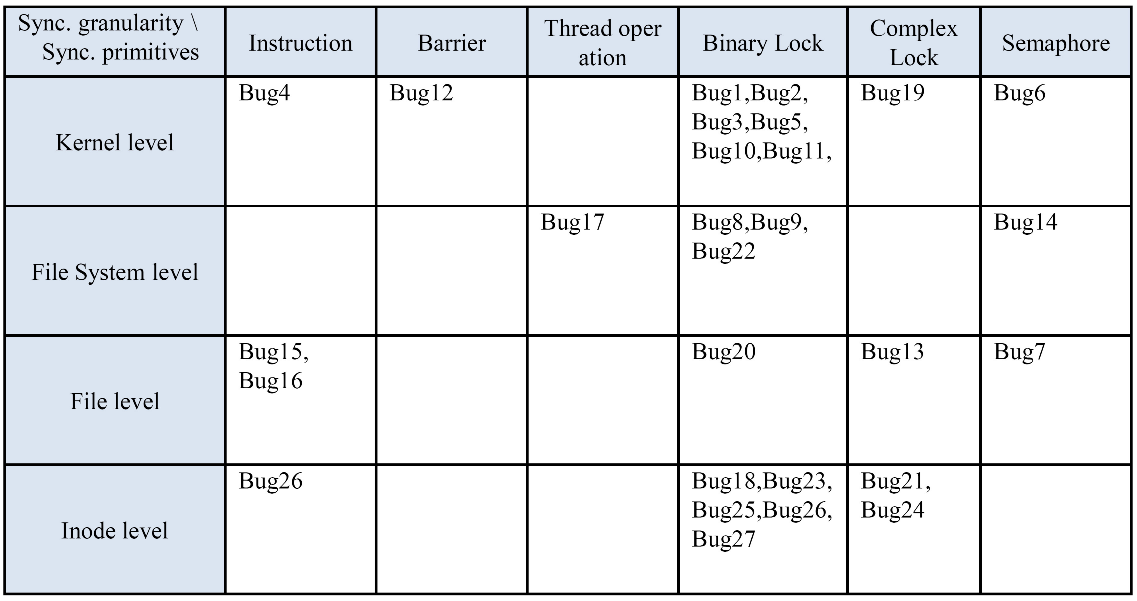
Fault \ Synchronization primitives	Instruction	Barrier	Thread oper ation	Binary Lock	Complex Lock	Semaphore
Design decision vio lation				Bug1,Bug2, Bug5,Bug9, Bug11,Bug18, Bug20,Bug22, Bug23,Bug25	Bug13, Bug24	Bug7
Incorrect use of syn chronization idioms	Bug26	Bug12		Bug8,Bug10, Bug26,Bug27	Bug21	Bug14
Program logic error	Bug4,Bug15 ,Bug16		Bug17	Bug3	Bug19	Bug6

Figure 6.4: Classification with respect to fault and synchronization primitive



Fault \ Synchronization granularity	Kernel Level	File-system level	File level	Inode level
Design decision vio lation	Bug1,Bug2,Bug5, Bug11	Bug9,Bug22	Bug7,Bug13,Bug20	Bug18,Bug23, Bug24,Bug25
Incorrect use of syn chronization idioms	Bug10,Bug12	Bug8,Bug14		Bug21,Bug26, Bug27
Program logic error	Bug3,Bug4,Bug6, Bug19	Bug17	Bug15,Bug16	

Figure 6.5: Classification with respect to fault and granularity



Sync. granularity \ Sync. primitives	Instruction	Barrier	Thread oper ation	Binary Lock	Complex Lock	Semaphore
Kernel level	Bug4	Bug12		Bug1,Bug2, Bug3,Bug5, Bug10,Bug11,	Bug19	Bug6
File System level			Bug17	Bug8,Bug9, Bug22		Bug14
File level	Bug15, Bug16			Bug20	Bug13	Bug7
Inode level	Bug26			Bug18,Bug23, Bug25,Bug26, Bug27	Bug21, Bug24	

Figure 6.6: Classification with respect to synchronization granularity and synchronization primitive

감 사 의 글

내 힘이요 반석이요 상급이 되시는 하나님께 감사를 드립니다. 하나님께서 함께 하신 시간들, 그 속에서 자라나게 하신 믿음, 그리고 석사 과정 동안 만나게 하신 여러분과 베푸신 모든 은혜를 기억하고 기억하며 살겠습니다.

연구실로 불러주시고 가르쳐주시고 도와주신 김문주 교수님께 감사 드립니다. 성실하고 정직하게 공부하시는 모습으로 참 연구자의 모습에서 많은 것을 배울 수 있었습니다. 교수님께서 기다려주시고 믿어주시고 격려해주신 것 정말 깊이 감사 드립니다. 동기 김윤호 박사과정에게도 고맙다는 말을 전합니다. 항상 도움과 웃음이 큰 위로였습니다. Liu Yuyang 박사과정에게도 감사를 드립니다. 함께 생활하며 보여준 패기와 열심이 제게 큰 귀감이 되었습니다. 연구실에서 만난 최창범, 홍창기, 이준희 학생에게 감사 드립니다. 교수님과 여러 학생 덕분에 행복한 대학원 생활이었습니다.

좋은 연구 주제 주시고 지원해주신 삼성전자 김호태 책임연구원님, ETRI의 김태호 박사님께 감사를 드립니다. 함께 연구하였던 Prof. Bow-Yah Wang께도 감사를 드립니다. 수많은 질문에 성실하게 응답해준 Linux Maintainer들께 감사를 드립니다. KAIST와 전산과 여러분께 깊은 감사를 드립니다.

사랑하는 아버지, 어머니, 할아버지, 할머니, 모든 가족에게 감사를 드립니다. 멀리 있고 바쁘다는 핑계로 아들 노릇 못해도 용서하시고 항상 응원해주시고 도와주셔서 감사합니다. 지난 3년간 함께 나누지 못한 많은 것들이 아쉽습니다. 앞으로는 더 많은 것들 함께 하며 은혜에 보답하겠습니다.

항상 믿음을 단단히 붙잡을 수 있도록 기도해주시고 격려해주신 이중기 장로님, 김명옥 권사님, 그리고 이은정 자매님께 깊은 감사를 드립니다.

주님 안에서 한 가족이 된 대덕 한빛 교회에 감사 드립니다. 아들처럼 아껴주시고 섬겨주신 김세진 목사님, 윤영식 전도사님, 민병우 전도사님, 김진현 전도사님과 형제와 자매 된 주의빛목장, 초등부 선생님들, 새벽이슬 여러분께 감사 드립니다.

Curriculum Vitae

Name	:	Hong, Shin
Date of Birth	:	June 18, 1985
Address	:	Department of Computer Science, 335 Gwahak-ro, Yuseong-gu, Daejeon 305-701 Republic of Korea
E-mail	:	hongshin@gmail.com

Educations

2007. 3. – 2010. 2.	M.S. Computer Science, KAIST
2003. 3. – 2007. 2.	B.S. Computer Science, KAIST
2001, 1. – 2003. 2.	Pusan Science High School (Currently, Korea Science Academy)

Publications

International Conferences

1. M.Kim, **S.Hong**, C.Hong, and T.Kim. Model-based Kernel Testing for Concurrency Bugs through Counter Example Replay. Model-based Testing (ENTCS vol 253, Issue 2), York, UK, Mar 2009.

Domestic Journals

1. Moonzoo Kim, **Shin Hong**. 모델기반의 커널 테스팅 프레임워크. Journal of KIISE:Software and Applications, 36(7), July 2009.

Domestic Conferences

1. Moonzoo Kim, Changki Hong, and **Shin Hong**. 검증반례재연을 통한 모델 기반 커널 테스팅. Korea Conference on Software Engineering (KCSE), February 2009.

Thesis

1. Concurrency Bug Detection through Improved Pattern Matching Using Semantic Information. Master Thesis, Department of Computer Science, KAIST, 2010.