Effective pattern-driven concurrency bug detection for operating systems

Shin Hong, Moonzoo Kim*

Computer Science Department, KAIST, South Korea

A R T I C L E   I N F O

Article history:
Received 6 September 2011
Received in revised form 12 July 2012
Accepted 27 August 2012
Available online 25 September 2012

Keywords:
Concurrency bug
Bug pattern
Static analysis
Linux

A B S T R A C T

As multi-core hardware has become more popular, concurrent programming is being more widely adopted in software. In particular, operating systems such as Linux utilize multi-threaded techniques heavily to enhance performance. However, current analysis techniques and tools for validating concurrent programs often fail to detect concurrency bugs in operating systems (OSes) due to the complex characteristics of OSes. To detect concurrency bugs in OSes in a practical manner, we have developed the CONcurrency Bug DETector (COBET) framework based on composite bug patterns augmented with semantic conditions. The effectiveness, efficiency, and applicability of COBET were demonstrated by detecting 10 new bugs in file systems, device drivers, and network modules of Linux 2.6.30.4 as confirmed by the Linux maintainers.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

As multi-core hardware becomes increasingly powerful and popular, operating systems (OSes) such as Linux utilize the cutting-edge multi-threaded techniques heavily to enhance performance. However, current analysis techniques and tools for concurrent programs have limitations when they are applied to operating systems due to the complex characteristics of OSes. In particular, the following three characteristics of OSes make concurrency bug detection on OSes difficult.

• Various synchronization mechanisms utilized
  Most concurrency bug detection techniques (Choi et al., 2002; Engler and Ashcraft, 2003; Naik et al., 2009; Raza and Vogel, 2008; Savage et al., 1997; Young et al., 2007) focus on lock usage, since a majority of user-level applications utilize simple mutexes/critical sections to enforce synchronization. However, OSes exploit various synchronization mechanisms (see Table 1) for performance enhancement.

• Customized synchronization primitives
  OS developers sometimes implement their own synchronization primitives. Thus, concurrency bug detection tools for standard synchronization mechanisms do not recognize these customized synchronization primitives and produce imprecise results (Xiong et al., 2010).

• High complexity of operating systems
  A dynamic analysis (i.e., testing) often fails to uncover hidden concurrency bugs due to the exponential number of possible interleaving scenarios between threads in OSes. In addition, replaying bugs is difficult, since it is hard to manipulate thread schedulers in OSes directly. A static analysis, on the other hand, has limited scalability to analyze OS code due to its high complexity and complicated data structures. Furthermore, the monolithic structure (i.e., tightly coupled large global data structure) of OSes severely hinder modular analyses.

For these reasons, in spite of much research on concurrent bug detection (see Section 6), such techniques have seldom been applied to OS development in practice.

To alleviate the above difficulties, we have developed the CONcurrency Bug DETector (COBET) framework, which utilizes composite bug patterns augmented with semantic conditions. Note that concurrency errors are caused by unintended interference between multiple threads. A salient contribution of COBET is that it utilizes multiple sub-patterns, each of which represents a buggy pattern in one thread, and checks semantic information that determines possible interferences between multiple threads in a precise and scalable manner (see Section 3). In addition, since engineers who use COBET can define various concurrency bug patterns in a flexible manner, COBET can detect concurrency bugs that are due to customized synchronization mechanisms or not targeted by lock-based concurrency bug detection tools.

One drawback of COBET is that a user has to identify and define bug patterns. To identify effective (i.e., detecting many bugs) and precise (i.e., raising few false alarm) bug pattern requires user’s domain knowledge on target code. In addition, it takes time to concretely define bug patterns for identified bugs in a machine
Table 1
Statistics on the synchronization statements in the Linux kernel 2.6.30.4.

<table>
<thead>
<tr>
<th>Atomic inst.</th>
<th>Cond. var.</th>
<th>Memory barrier</th>
<th>Mutex</th>
<th>rw sema-phore</th>
<th>rw spin lock</th>
<th>Sema-phore</th>
<th>Spin lock</th>
<th>Thread operation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Stmt</td>
<td>8926</td>
<td>949</td>
<td>1926</td>
<td>14,902</td>
<td>2471</td>
<td>4248</td>
<td>759</td>
<td>44,205</td>
<td>460</td>
</tr>
<tr>
<td>Ratio</td>
<td>11.3%</td>
<td>1.2%</td>
<td>2.4%</td>
<td>18.9%</td>
<td>3.1%</td>
<td>5.4%</td>
<td>1.0%</td>
<td>56.1%</td>
<td>0.6%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100.0%</td>
</tr>
</tbody>
</table>

processable form. Without such effort, it is easy to define imprecise bug patterns, which increases the burden to filter out false alarms manually and, thus, decreases practical usefulness of the COBET framework.1

However, once such bug patterns are well-defined, corresponding pattern detectors can be implemented to detect concurrency bugs in (1) subsequent releases of the target program, and/or (2) other modules in a similar domain. It has been frequently observed that although a given bug had been fixed previously, similar bugs often appeared in the subsequent releases or in the different modules of the target program (see Sections 5.1 and 5.3). Thus, initial efforts to define bug patterns could be sufficiently rewarded by detecting concurrency bugs in rapidly evolving large software systems such as Linux. Furthermore, to lessen the effort to define bug patterns and construct corresponding bug pattern detectors, the COBET framework provides a pattern description language (PDL) (see Section 3.2).

Currently, COBET provides four concurrency bug patterns that are identified based on a review of Linux kernel ChangeLog documents. The effectiveness of COBET was demonstrated by detecting 10 new bugs in files systems, network modules, and device drivers of Linux 2.6.30.4 (the latest Linux release at the moment of the experiments), which were confirmed by Linux maintainers.

The contributions of this research are as follows:

- We have derived interesting observations on the Linux concurrency bugs from a review of the Linux ChangeLog documents on Linux 2.6.x releases (Section 2).
- We have developed a pattern-based concurrency bug detection framework, which can define and match various bug patterns. To improve bug detection precision, our framework utilizes composite patterns with semantic conditions in a scalable manner (Section 3).
- Based on previous bug reports, we have defined four concurrency bug patterns with various synchronization mechanisms, which are effective to detect new bugs in Linux that are not targeted by lock-based analysis techniques. (Sections 4 and 5).

The remainder of this paper is organized as follows. Section 2 describes the characteristics of Linux to show the advantages of pattern-based bug detection approach on Linux. Section 3 overviews the COBET framework. Section 4 explains composite bug patterns with semantic conditions upon the COBET framework. Section 5 reports the evaluation of the COBET framework through the empirical results on Linux kernel. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2. Characteristics of Linux operating system

In this section, we describe the characteristics of concurrent programming practices used in Linux.

2.1. Synchronization mechanisms in Linux

Linux utilizes various synchronization mechanisms for enhanced performance. We gathered statistics on the nine standard synchronization mechanisms in the entire kernel code of Linux 2.6.30.4, which consists of around 11.6 million lines of C code. These nine synchronization mechanisms include atomic instructions, conditional variables, memory barriers, mutexes, read/write sema-phi-ores, read/write spin locks, semaphores, spin locks, and thread operations (e.g., thread creation and join). Those synchronization mechanisms are identified in target code by the name of the corresponding library function calls.

Table 1 shows the numbers of statements for the nine synchronization mechanisms. Locks, the most popular synchronization mechanism, can be implemented by using spin locks, mutexes, and binary semaphores. Thus, locks take 75-76% (+56.1% ± 18.9% ± 0-1.0%) of all synchronization statements in the Linux kernel code. Consequently, 24–25% of synchronization statements cannot be examined by lock-based bug detection techniques.

2.2. Survey of the Linux bug reports

We reviewed 324 ChangeLogs on Linux 2.6.0–2.6.30.3 to understand the nature of real concurrency bugs (as Lu et al., 2008 did on large application programs) and identified concurrency bug patterns accordingly. We concentrated on the bug reports related to Linux file systems for the following three reasons. First, file systems utilize heavy concurrency to handle multiple I/O transactions simultaneously. Thus, we expected that file systems had many concurrency issues. Second, there are relatively rich reference documents on the Linux file systems, so that it is easy to understand the bug reports and define bug patterns. Third, as Linux file system consists of multiple naive file systems such as ext4 and ext2 whose overall functionalities are similar, we expected that we could find a concurrency bug that occurred commonly in multiple naive file systems, which can be a good candidate for a bug pattern to define.

We collected the concurrency bug reports on the Linux file systems by searching related keywords (i.e., ‘lock’, ‘concurrency’, ‘data races’, ‘deadlock’, etc.) as well as manual inspection. Finally, we found 50 concurrency bug reports on the Linux file systems and 27 of them were selected for in-depth review (the remaining 23 bugs were discarded, since these bugs were caused by domain-specific requirement violations or could not be understood concretely). Through the review, we made the following observations:

Observation 1: Half of the concurrency bugs are involved with synchronization mechanisms other than locks. 12 of the 27 bugs were associated with synchronization mechanisms other than locks (i.e., atomic instructions, memory barriers, thread operations, etc.). In addition, locks were sometimes used in a non-standard manner (e.g., recursive locking and releasing on blocking). This observation indicates that we need customizable/flexible concurrency bug detection tools that can analyze various synchronization mechanisms, not only standard lock usages.

Observation 2: Code review was more effective to detect concurrency bugs than runtime testing was. Linux ChangeLogs reported that, among the 27 concurrency bugs, nine were detected by actual testing and 13 bugs detected by manual code review (the sources of the remaining five bugs were not clear). In general, code review does not reason with concrete input data and scheduling, but by

---

1 We have defined only four bug patterns (Section 4), since we had to learn domain knowledge on Linux kernel from scratch in limited research time. However, if Linux developers define bug patterns, they could build a database containing many effective and precise bug patterns in modest time. Since COBET is very fast to apply bug patterns to large program code (see Tables 3–5), a large number of bug patterns may not cause much overhead to detect concurrency bugs.
reading code statically. Note that pattern-based concurrency bug detection also has this characteristic of the code review.

**Observation 3:** Linux kernel code was updated frequently. For six years, 324 Linux releases (including major releases and minor releases) have been made. This means that a new Linux kernel has been released on average every week. In addition, on average, 3,831 patches have been applied to Linux 2.6.x releases per hour (Kroah-Hartman et al., 2009). Furthermore, the Linux kernel has been constantly growing up from 2.6.11 release (6.6 million lines of code in 17,090 files) to 2.6.30 release (11.6 million lines of code in 27,911 files) (Kroah-Hartman et al., 2009). Thus, we need a lightweight bug detection framework that can analyze a large program quickly and conveniently.

### 2.3. Complexities of Linux file systems

To estimate the complexities of the Linux file systems, we counted the number of different call sequences by traversing the inter-procedural control flow graphs (CFGs) starting from thread-starting functions such as system calls for the seven Linux file system codes, including btrfs, ext4, nfs, proc, reiserfs, sysfs, and udev. The number of call sequences can serve as a measure for the complexity of the file system, since each call sequence represents a unique execution scenario. Each file system is analyzed together with virtual file system (VFS) code.

Table 2 describes the statistics on call sequences of the seven Linux file systems. To analyze all of these file systems (145KL, i.e., 145 thousand lines of C codes), we had to analyze 20 billion different execution scenarios whose average call depth is around 38 (see the last column of Table 2). Furthermore, due to non-deterministic scheduling, the total number of concurrent execution scenarios is exponential in the number of sequential execution scenarios in Table 2. Thus, it is clear that, due to the huge number of execution scenarios, achieving high coverage by testing and/or model checking is infeasible. Therefore, lightweight analysis techniques should be developed for complex operating systems like Linux.

### 3. COBET framework

The observations in Section 2 suggest that a pattern-based concurrency bug detection framework can be a practical solution for Linux. Thus, we have developed the Concurrency Bug DETector (COBET) framework for concurrent C programs based on a pattern matching approach.

#### 3.1. Overview of the COBET framework

The overall structure of the COBET framework is depicted in Fig. 1. Since concurrency errors are caused by unintended interferences among multiple concurrent threads, a concurrency bug pattern should be specified as multiple sub-patterns each of which captures a specific code running on each thread. For this purpose, COBET provides a pattern description language (PDL) to describe the syntactic structure of a bug pattern (see Section 3.2). The COBET synthesizer generates a bug pattern detector from a user-specified bug pattern description in PDL. A synthesized bug pattern detector contains the following four components:

- syntactic pattern matcher,
- semantic condition checker,
- semantic analysis engine, and
- abstract syntax tree (AST) generator.

A syntactic pattern matcher in a generated bug pattern detector detects segments of a target program code that match sub-patterns in the given PDL description. Then, a semantic condition checker checks whether or not these code segments can run concurrently and interfere with each other through a semantic analysis engine. For this purpose, the semantic analysis engine performs path analysis, lock analysis, and alias analysis (see Section 3.3). For these analyses, a user should provide configurations of a target program, which include names of thread starting functions, specifications of lock/unlock operations (e.g., `spin_lock()` and `spin_unlock()`), and specification of memory allocation operations (e.g., `kmalloc()` and `kmalloc_cache_alloc()`). Different target domains may have different configurations. At the lowest layer, the AST generator parses and creates the AST of a target program using the EDG parser (EDG Group, 2011). The AST of a target program is used by the syntactic pattern matcher to detect code segments that match bug patterns syntactically.

The COBET framework consists of 4500 lines of C code in 96 functions. The COBET framework uses GCC 4.3.0 to preprocess a target code and EDG C/C++ Front-End 3.1 to parse the preprocessed code.

#### 3.2. Bug pattern detectors

The COBET synthesizer constructs bug pattern detector code from a user-given bug pattern specification. A pattern description language (PDL) is designed to help engineers define bug patterns in a correct and convenient manner. Fig. 2 shows the brief grammar of PDL. In PDL, a concurrency bug pattern is described as a set of sub-patterns (line 1 of Fig. 2), each of which specifies target code running on one thread. A sub-pattern contains one or more function descriptions (line 2). A function description consists of abstract statement descriptions (line 3). An abstract statement description (lines 4–10) is specified with a keyword (e.g., `if`, `loop`, `lock`, and `read`) which indicates a type of target code statement to match. A bug pattern detector based on PDL searches target code to find matched code statements while ignoring irrelevant code statements. In addition, PDL can describe a bug pattern by using `\((Stmt^*)\)` (exclusion), which specifies statements that should not appear in pattern matching.
instances. In PDL, $<name>$ is an untyped free variable that binds a corresponding code element in a target C statement.

First, a bug pattern detector matches a PDL description to a target code in a syntactic manner using the tree pattern matching algorithm (Dubiner et al., 1994); the syntactic pattern matcher in the pattern detector maps each abstract pattern statement to a C statement and each free variable to a C expression. For example, a pattern description if $\text{<cond>}$ { write $\text{<var>} ;$ } can match if $(x<0) \{ \text{x=f(x); y=x*x; }$ } and generates the following two pattern matching instances. For the first matching instance, $\text{<cond>}$ and $\text{<var>}$ are bound to $x<0$ and $x$ in a target code, respectively. For the second instance, $\text{<cond>}$ and $\text{<var>}$ are bound to $x<0$ and $y$, respectively. Free variables of PDL are used to describe subtle conditions in a bug pattern.

Second, to check semantic conditions on a pattern matching instance, pattern detector code invokes $\text{sem\_cond\_checking()}$ at every syntactic matching/binding step. As a default, $\text{sem\_cond\_checking()}$ checks feasibility of interference among code segments that match sub-patterns and run on multiple threads (e.g., checks whether or not the sets of held locks at different sub-patterns are disjoint (see Section 3.3)). In addition, a user can add sophisticated semantic condition checking routines to this function, since synthesized pattern detector code is human-readable. For this purpose, COBET provides library functions to check the semantic conditions in a bug pattern matching instance. Fig. 6 shows one example of $\text{sem\_cond\_checking()}$ for ‘misused test and test-and-set’ bug pattern (see Section 4.1).

### 3.3. Semantic analysis engine

The COBET semantic analysis engine checks whether or not multiple code segments that match specified sub-patterns can run concurrently and interfere with each other through path analysis, lock analysis, and alias analysis. The semantic analysis engine performs the path analysis first to explore inter-procedural execution paths reaching functions that match at least one syntactic sub-pattern. Then, lock analysis and alias analysis are performed on these execution paths.  

For example (see Fig. 3), suppose that a bug pattern $b$ consists of two sub-patterns, $b.1$ and $b.2$. Also, assume that, through syntactic pattern matching, COBET detects that the target program has functions $f_1()$ and $f_2()$, which contain statements $s_{b.1}$ and $s_{b.2}$ matching $b.1$ and $b.2$, respectively. Then, the COBET semantic analysis engine statically explores execution paths starting from thread-starting functions to the target functions $f_1()$ or $f_2()$ (path analysis). Suppose that the COBET analysis engine finds that there is a lock to prohibit concurrent executions of $f_1()$ or $f_2()$ in all possible paths (lock analysis). Then, COBET concludes that $b$ does not occur, since $s_{b.1}$ and $s_{b.2}$ cannot be interleaved and, thus, cannot interfere with each other. If there is no such lock to prevent interference between $s_{b.1}$ and $s_{b.2}$, then COBET checks whether or not $s_{b.1}$ and $s_{b.2}$ can access the same variable causing interference (alias analysis). If the alias analysis result indicates that $s_{b.1}$ and $s_{b.2}$ can access the same shared variable, COBET reports that $b$ is detected in the target program.

**Path analysis:** COBET’s path analysis generates an inter-procedural CFG from the AST of a target program. Then, the path analysis generates inter-procedural execution paths starting from thread starting functions to the functions that match specified sub-patterns by exploring the interprocedural CFG. Since exploration of all inter-procedural execution paths in the OS consumes a huge amount of time (see Section 2.3), COBET conducts syntactic pattern matching first to prune irrelevant execution paths.

If a target program has many function pointers, which are frequently used to link lower-layer modules to upper-layer modules in layered OS architecture, it is hard to generate an accurate

---

**Table 2**

Statistics on the call sequences of the seven Linux file systems.

<table>
<thead>
<tr>
<th></th>
<th>btrfs</th>
<th>ext4</th>
<th>nfs</th>
<th>proc</th>
<th>reiserfs</th>
<th>sysfs</th>
<th>udf</th>
<th>Total/avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>41KL</td>
<td>28KL</td>
<td>29KL</td>
<td>8KL</td>
<td>27KL</td>
<td>3KL</td>
<td>9KL</td>
<td>Total 145KL</td>
</tr>
<tr>
<td># of call sequences</td>
<td>2100M</td>
<td>1501M</td>
<td>3394M</td>
<td>12M</td>
<td>13413M</td>
<td>1M</td>
<td>51M</td>
<td>Total 20488M</td>
</tr>
<tr>
<td>Max. length of call seq.</td>
<td>88</td>
<td>54</td>
<td>57</td>
<td>33</td>
<td>55</td>
<td>26</td>
<td>43</td>
<td>Avg. 51</td>
</tr>
<tr>
<td>Avg. length of call seq.</td>
<td>60</td>
<td>43</td>
<td>39</td>
<td>25</td>
<td>38</td>
<td>23</td>
<td>35</td>
<td>Avg. 38</td>
</tr>
</tbody>
</table>
inter-procedural CFG, since we do not know which functions will be called via function pointers. COBET solves this problem using the following heuristics. COBET constructs a function pointer table consisting of pairs of a function pointer and candidate functions which can be invoked through the function pointer, by analyzing assignment statements on global variables of function pointer type. The path analysis refers to the table when it reaches a function call via a function pointer. If there exist multiple candidate functions for a function pointer, the path analysis generates multiple paths to all these candidate functions in a conservative manner.

Lock analysis: COBET’s lock analysis formulates locksets (i.e., a set of held locks) at a code location. The lockset information is used to check whether or not two code locations are guarded by the same lock. The lock analysis obtains a lockset of a code location by exploring inter-procedural execution paths while recording lock acquiring operations and lock releasing operations. The lock analysis recognizes lock/unlock operations in a target program based on the lock operation function names specified in an input configuration to COBET. The technique concludes that two lock operations with parameters acquire the same lock if their parameters may alias each other. COBET performs path-insensitive lock analysis due to the high computational cost of path-sensitive lock analysis. For a branching statement, the lock analysis explores both branches and takes the union of the two locksets obtained from both branches as a result of the branching statement. For a loop statement, COBET analyzes only one iteration.

COBET uses an inter-procedural lock analysis. In other words, the technique transfers the lockset of a call site in a caller function to the analysis on the callee function. However, to prevent propagation of incorrect lock analysis results from a callee function, COBET does not reflect the locksets of exit statements in the callee function to the caller function (Engler and Aschcraft, 2003).

To avoid redundant inter-procedural lock analysis, COBET semantic analysis engine uses a cache that records the lock analysis result for functions. When the analysis reaches a call site of $f()$ with a lockset $L$, COBET tries to find $(f(), L)$ in the cache. If the cache does not contain such an entry, the analysis continues to the callee function $f()$. Otherwise, the analysis does not go into $f()$. Fig. 4 illustrates the cache operations. The lock analysis on the first execution path records $(f1(), \emptyset), (f2(), \{L1\})$ in the cache. The lock analysis on the second execution path skips $f2()$, since the cache already contains $(f2(), \{L1\})$. This cache technique saves a large amount of lock analysis time in our experiments, achieving 2–20 times speedup compared to non-caching lock analysis.

**Figure 4. Caching in the lock analysis.**

**Alias analysis:** COBET’s alias analysis statically examines whether or not two expressions may access the same shared variable. COBET utilizes the extended type information on variables (i.e., type of a variable and the type of a structure containing the variable as a field) for the analysis and considers two heap variables of the same type to be aliased.

COBET assumes that the following variables are non-shared: (1) local variables, and (2) dynamically allocated variables through function calls such as `malloc()` and `kmem_cache_alloc()` that are not assigned to global variables yet (i.e., performing ‘uniqueness analysis’; Pratikakis et al., 2011). The technique traces pointer assignments and considers a variable that is assigned with a non-shared variable to be a non-shared variable, too. At a function call site, if actual parameters of the function are non-shared variables, the analysis utilizes this information during the analysis of the callee function.

**4. Composite bug patterns with semantic conditions**

After reviewing the bug reports on Linux file systems (see Section 2.2), we defined the following four bug patterns:

1. misused test and test-and-set,
2. unsynchronized communication at thread creation,
3. incorrect usage of atomic operations, and
4. waiting for an already terminated thread.

For each bug pattern, it takes approximately 3 h for one graduate student with the knowledge on the Linux file systems and the bug pattern to define a corresponding syntactic bug pattern in PDL and implement a bug pattern detector based on the generated template code upon the COBET framework. The following subsections explain these four patterns in detail.

**4.1. Misused test and test-and-set**

Test and test-and-set programming idioms are used to reduce the number of expensive lock operations required to protect a shared variable.

1: if(c) { // Test and test-and-set idiom
2:   lock l2;
3:   if(c) {
4:     update v;
5:   unlock l2;

Suppose that $c$ indicates whether or not a current thread can update a shared variable $v$. Before performing an expensive lock operation (line 2) to update $v$ safely (line 4), this idiom checks whether or not $c$ is satisfied (line 1). Thus, the lock operation is executed only when $c$ is true. If $c$ is true (line 1), the above code performs a lock operation (line 2) and checks $c$ again (line 3), since $c$ might be changed by other threads between lines 1 and 3.

Unfortunately, programmers often omit this second check (line 3), which results in a race condition. For example, Linux ChangeLog 2.6.11.1 reported a data race bug in `ext3_discard_reservation()` in the ext3 file system, which was caused by misused test and test-and-set idiom. We suspect that similar bugs existed in the subsequent releases or other modules of Linux as this bug pattern could be easily introduced during manual code optimization.

We define the ‘misused test and test-and-set’ bug pattern\(^3\) as two sub-patterns, pattern 1 and pattern 2, in Fig. 5. This bug pattern is different from ‘double-checked locking’ (Hovemeyer and Pugh, 2004), which consider test and test-and-set idioms as bugs due to the Java memory model. Our bug pattern checks whether or not test and test-and-set idioms are correctly used in general.

---

\(^3\) This pattern is different from ‘double-checked locking’ (Hovemeyer and Pugh, 2004), which consider test and test-and-set idioms as bugs due to the Java memory model. Our bug pattern checks whether or not test and test-and-set idioms are correctly used in general.
pattern has the following semantic conditions to check in all pattern matching instances:

1. A lockset at target code that matches $3a$ must be disjoint with the lockset at target code that matches $3b$. Otherwise, target code that matches pattern 1 and pattern 2 do not interfere with each other.
2. $\$w$ should be a shared variable.
3. $\$cond$ should contain a variable that is equal to or alias to $\$w$.

\texttt{sem\_cond\_checking(bug\_instance \textbf{bi})} in Fig. 6 checks these semantic conditions. Through the pattern matching, each field of \texttt{bug\_instance} contains a target code element that matches a corresponding PDL element of a bug pattern. For each binding of \texttt{bug\_instance} to a target code entity, \texttt{sem\_cond\_checking(bug\_instance \textbf{bi})} is invoked to check if the current pattern matching instance \textbf{bi} satisfies the semantic conditions. Line 2 in Fig. 6 checks whether or not target code that matches pattern 1 and pattern 2 is protected by a common lock. \texttt{\textbf{bi}.\_3a} and \texttt{\textbf{bi}.\_3b} represent target code that matches the abstract statements 3a and 3b in the PDL description (see Fig. 5). \texttt{\textbf{bi}.\_lock\_disjoint} (\texttt{\textbf{bi}.\_3a, \textbf{bi}.\_3b}) obtains the lock analysis results for the code statements that match abstract statements 3a and 3b in the PDL description. Lines 4 and 6 utilize alias analysis to check whether or not matched code statements may access the same shared variable and cause a data race. \texttt{\textbf{bi}.\_w} and \texttt{\textbf{bi}.\_cond} represent the code elements that match $\$w$ and $\$cond$ in the PDL description, respectively.

We now illustrate how COBET detects this bug pattern in the example shown in Fig. 7. In this example, suppose that COBET detects two syntactically matching instances:

- **Matching instance 1**: \texttt{\textbf{proc\_get\_sb}()} (lines 1c–6c) and \texttt{\textbf{proc\_get\_sb}()} (lines 1d–4d) matched pattern 1 and pattern 2, respectively.
- **Matching instance 2**: \texttt{\textbf{proc\_get\_sb}()} (lines 1c–6c) and \texttt{\textbf{proc\_alloc\_inode}()} (lines 1e–5e) matched pattern 1 and pattern 2, respectively.

For matching instance 1, to check semantic condition 1 (the lockset at target code that matches 3a must be disjoint to the lockset at a target code that matches 3b), COBET checks whether or not there exists a lock to synchronize the target codes that match pattern 1 and pattern 2. COBET finds that the lockset at line 3c and the

\begin{verbatim}
1a: pattern 1 { 1b: pattern 2 {
2a: fun $f1 { 2b: fun $f2 {
3a: if $cond {
3b: write $w;
4a: lock $l;
5a: \(\text{if } $cond \{ \} \)
6a: unlock $l;
7a: })}

\textbf{Fig. 5.} Misused test and test-and-set bug pattern.
\end{verbatim}

\begin{verbatim}
\textbf{Fig. 6.} Semantic condition checking function.
\end{verbatim}

\begin{verbatim}
1: if (is_shared_var(bi._w) == FALSE)
2: return FALSE;
3: if (is_lock_disjoint(bi._3a, bi._3b) == FALSE)
4: return FALSE;
5: if (may_alias(bi._cond, bi._w) == FALSE)
6: return FALSE;
7: return TRUE;

\textbf{Fig. 7.} \texttt{proc\_get\_sb()} (in fs/proc/inode.c) and \texttt{proc\_alloc\_inode()} (in fs/proc inode.c) of Linux kernel 2.6.30.4.
\end{verbatim}

lockset on line 3d always contained \texttt{lock\_kernel}. This indicates that line 3c and line 3d cannot run concurrently, thus cannot interfere each other. Thus, COBET ignores this matching instance.

For matching instance 2, COBET finds that there is an execution path that has no lock to synchronize the target code that match pattern 1 and pattern 2. However, by checking semantic condition 2 ($\$w$ should be a shared variable), the alias analysis finds that ei->pid at line 5e is not a shared variable, since ei was allocated (line 3e) and had not yet become shared. Therefore, COBET concludes that the matching instance 2 should be rejected as well.

As another example, the following matching instance was found in the network module (see Section 5.3) as shown in Fig. 8.

- **Matching instance 3**: \texttt{htable\_put()} (lines 1f–5f) and \texttt{htable\_find\_get()} (lines 1g–3g) matched pattern 1 and pattern 2, respectively.

Matching instance 3 was not filtered out by the semantic analyses, so we reported this result as a suspected bug to a corresponding Linux maintainer. The Linux maintainer in charge of netfilter confirmed this bug report and fixed \texttt{htable\_put()} in Linux 2.6.34 (Linux, 2010).

4.2. Unsynchronized communication at thread creation

The Linux kernel creates a new thread by using \texttt{kthread\_run()}. For instance, \texttt{kthread\_run(func, arg, &daemon}) creates a new kernel thread whose name is "daemon" and then executes

\begin{verbatim}
1: void htable_put(xt_hashlimit_htable *hinfo)
2: atomic_dec_and_test(&hinfo->use) {\}
3: ei = proc_I(sb->s_root->d_inode);
4: if(!ei->pid) {\}
5: rcu_read_lock();
6: ei->pid = get_pid(...);

\textbf{Fig. 8.} \texttt{htable\_put()} and \texttt{htable\_find\_get()} in net/netfilter/xt\_hashlimit.c of Linux kernel 2.6.30.4.
\end{verbatim}
Fig. 9. Unsynchronized communication at thread creation bug pattern.

```
1h: pattern 1 { 11: pattern 2 {
2h: fun $f1 { 21: fun $f2 {
3h: call "kthread_run" $a1;
31: read $a3;
4h: write $a2;
41: }
5h: }
}
```

Fig. 10. btrfs_start_workers() and worker_loop() at fa/btrfs/async-thread.c of Linux 2.6.30.4.

```
func(arg) on the thread, arg is a single variable used as a function parameter. Through this variable, a parent thread transfers data used for the initialization of a new thread. In many cases, a parent thread passes a shared memory address through which the parent thread communicates with the child thread. For this type of communication, a parent thread and the child thread should be synchronized. However, programmers often omit synchronization so that concurrent execution of a parent thread and its child thread can exhibit data race errors. Linux ChangeLog 2.6.24 reported such a bug in GFS2 file system.

We defined this ‘unsynchronized communication at thread creation’ bug pattern as the two sub-patterns pattern 1 and pattern 2 shown in Fig. 9. pattern 1 indicates a parent thread that calls kthread_run() and then assigns some value to the shared memory (line 4h). pattern 2 describes a function executed by a child thread. $f2 reads data passed from its parent thread through a pointer of the function parameter (line 3i). Since PDL does not specify expression-level conditions, a user needs to add additional condition checking code to the synthesized bug detector code (i.e., the first element of $a1 should be same to $f2). This bug pattern has the following semantic conditions:

1. The lockset at 4h must be disjoint with the lockset at 3i.
2. $a2 should be a shared variable.
3. $a3 should contain a variable which is equal to or alias to $a2.

The ‘unsynchronized communication at thread creation’ bug detector found a new bug in btrfs, as shown in Fig. 10. The child thread (worker_loop()) can access worker and read an invalid value (line 4k), since the parent thread (btrfs_start_workers()) may not have assigned a proper value to worker (line 4j) yet. We reported this bug to the kernel developers and they made the patch immediately (see Linux ChangeLog 2.6.31).

```
1h: pattern 1 {
2h: fun $f1{
3h: call $atomic1 $a1;
31: }
4h: if ($a2) {
41: }
5h: }
```

4.3. Incorrect usage of atomic operations

CPU architectures often provide atomic instructions for synchronization operations, which guarantee the atomic execution of read and consequent update on operands. ‘test-and-set’ and ‘compare-and-swap’ are examples of these instructions. Linux kernel provides a special variable type atomic_t and library functions for atomic operations (e.g., atomic_read() and atomic_set()) to utilize these atomic instructions. When a programmer develops synchronization code, he/she should use a proper library function to handle two subsequent read and update atomically. However, a programmer often mistakenly uses two separate atomic operations instead of one combined atomic operation, and this can lead to data races.

We characterize this incorrect atomic operation usage as a bug pattern shown in Fig. 11. pattern 1 represents two consecutive atomic operations whose executions are intended to be atomic. pattern 2 expresses another thread which can be scheduled between these two atomic operations in pattern 1 and interfere their executions.

The semantic condition checking examines the following conditions:

1. The set of held locks at 31 must be exclusive to the set of held locks at 3m. Otherwise, pattern 1 and pattern 2 do not interfere with each other.
2. $a1 and $a3 should be shared variables and may be alias to each other.
3. $a2 should contain a variable which is equal to or alias to $a1.
4. $atomic1 is an atomic operation that updates $a1. Similarly, $atomic2 is an atomic operation that updates $a3.
5. $a2 should contain a function call to an atomic operation which reads $a1 (e.g., atomic_read()).

In addition, we modified the generated bug pattern detector code to check whether the type of associated variables is atomic_t or not. Also, we provide a list of library function names for atomic operations. The bug pattern detector utilizes these information to check the fourth and the fifth semantic conditions.

We found new bugs in the Linux kernel using this bug pattern detector. One example is shown in Fig. 12. This code updates &cp->in_pkts through atomic_inc() at line 5 and then examines its value through atomic_read() at line 10 separately, although these two operations should be executed together atomically. We
reported this bug to corresponding Linux maintainers and the maintainers immediately patched the code that these two separate atomic operations were replaced by one combined atomic operation atomic_add_return().

4.4. Waiting already finished thread

Linux kernel can create and execute a child thread by calling kthread_run() and terminate the child thread by calling kthread_stop(). kthread_stop() sends a special message to a child thread and waits until the child thread is terminated. A child thread should regularly call kthread_should_stop() that returns true if the message is received, and terminate accordingly. Otherwise, the parent thread waits indefinitely at kthread_stop().

A child thread contains a loop whose condition checks kthread_should_stop(); the child thread operates until kthread_should_stop() returns true. The child thread should not terminate even when the task is completed or the task may not proceed due to errors. Otherwise, the parent thread may invoke kthread_stop() after the child thread terminates and it waits indefinitely at kthread_stop(). Therefore, if a child thread terminates earlier than the parent thread calls kthread_stop(), deadlock will occur. Linux ChangeLog 2.6.28 reported this bug, where the loop was escaped by break statement for an error condition.

We specified this bug into a PDL pattern as shown in Fig. 13. pattern 1 represents a parent thread which creates a child thread and invokes kthread_run() for the child thread. pattern 2 specifies the function for a child thread which has a loop with kthread_should_stop(). Line 5p escapes the loop and terminates the child thread.

We detected related bugs in Linux 2.6.30.4 btrfs file system. One example is shown in Fig. 14. The child thread terminates when the error handling branch is taken (lines 5q–6q). The btrfs developers confirmed these bugs. Furthermore, the Linux kernel developers modified the semantics of kthread_stop() to prevent indefinite waiting if the child thread is already finished since Linux 2.6.32.

5. Empirical results

To investigate the effectiveness, efficiency, and applicability of the COBET framework, we performed the following three empirical evaluations on Linux 2.6.30.4, the latest version at the time of this empirical study.

- To determine whether pattern-driven bug detectors based on the old bug reports can detect new concurrency bugs in subsequent releases, we applied the four bug pattern detectors (based on the bug reports on the file systems in Linux 2.6.0–2.6.30.3) to the file systems in Linux 2.6.30.4. We reported our bug detection results to Linux maintainers and validated the bug detection results by their feedback (Section 5.1).
- We evaluated the effectiveness and efficiency of the three semantic analyses (path analysis, lock analysis, and alias analysis) of the COBET semantic analysis engine. We measured the improvement in bug detection precision and the additional time cost associated with each semantic analysis (Section 5.2).
- To investigate the applicability of the COBET framework, we applied the four bug pattern detectors not only to file systems, but also to other modules. We applied the four bug pattern detectors to the device drivers and the network modules, and then evaluated the bug detection capability (Section 5.3).

In addition, we applied Coverage Prevent (Coverity, 2011) to the same Linux targets to evaluate the advantages of COBET over conventional concurrency bug detectors (Section 5.5). Prevent is a static bug detector and demonstrated its effectiveness successfully through many real-world software projects (Bessey et al., 2010) including Linux kernels. We selected Prevent as a representative static concurrency bug detection tool to compare with COBET, since most other static concurrency bug detection tools are not available to analyze Linux kernel.

All empirical studies in this section were performed on 64-bit Fedora Linux 9 equipped with a 3.6 GHz Core2Duo processor and 16 GB memory.

5.1. Bug detection result on file systems

We applied the four bug pattern detectors (Section 4) to the seven Linux file systems (btrfs, ext4, nfs, proc, reiserfs, sysfs and udf). Since Linux file systems are tightly coupled with virtual file system layer (VFS), we analyzed each file system together with VFS. We specified all system call handling functions in VFS as the thread starting points.

Table 3 describes the number of detected bugs for each bug pattern and for each file system. The first row shows the sizes of the file systems before pre-processing (for example, btrfs is 41,000 lines long (the second column)). The first number in a cell of Table 3 indicates the number of new bugs detected by COBET. The second number indicates the number of real bugs among the detected bugs that were confirmed by the Linux maintainers. For example, COBET detected two ‘unsynchronized communication at thread creation’ bugs in btrfs file system (third row, second column). We reported these two suspected bugs to a btrfs maintainer, who confirmed that one was a real bug, but the other was a false alarm. COBET took only 9 s to apply these four bug pattern detectors to the seven file systems (see the last row and the last column of Table 4).

Another observation is that relatively new file systems such as btrfs have several concurrency bugs. For example, btrfs was introduced in the Linux 2.6.29 release in March 2009. It has three ‘waiting already terminated thread’ bugs and one ‘unsynchronized communication at thread creation’ bug, which were confirmed by the Linux maintainers. If we can generalize this result, COBET can detect bugs in recently revised modules more effectively. Considering that Linux evolves rapidly (see Section 2.2), a light-weight bug detection tool such as COBET can be a practical aid to detect concurrency bugs in the OS.

```c
1q: static int cleaner_kthread(void *arg) {
2q: btrfs_root *root = arg;
3q: do {
4q: smp_mb();
5q: if (root->fs_info->closing)
6q: break;
7q: ... }
8q: while(!kthread_should_stop());
9q: return 0;
}
```

Fig. 14. Waiting for an already finished thread bug detected at fs/btrfs/async-thread.c of Linux 2.6.30.4.
5.2. Evaluation of semantic analysis techniques

To investigate the effectiveness and efficiency of the COBET semantic analyses, we measured the false alarm reduction rate through the semantic analyses and additional time cost for the analyses. For this purpose, we performed four series of studies for each bug pattern detector with different combinations of semantic analyses (see Section 3.3).

1. The first series of studies detected one main sub-pattern without semantic analysis (see the second column of Table 4). This series of studies was similar to the studies with conventional pattern-based bug detection tools (e.g., Metal [Hållem et al., 2002] and FindBugs [Hovemeyer and Pugh, 2004b]).

2. The second series of experiments detected multiple sub-patterns of a bug pattern, but still without semantic analyses.

3. The third series extended the second series by performing path analysis and lock analysis as well. Note that the lock analysis depends on the path analysis and cannot be performed separately.

4. The fourth series extended the third by performing alias analysis as well. This series utilizes all semantic analyses by the semantic analysis engine.

Table 4 describes the numbers of bugs detected and corresponding analysis time on the seven file systems in total. This table shows that the false alarms are reduced as additional analysis techniques are employed. For example, ‘misused test and test-and-set’ bugs (second row of Table 4) are reduced from 51 to 36, 32, and 28 as multiple pattern matching, path/lock analyses, and path/lock/alias analyses are applied respectively; finally 45% (=51 – 28)/51 of the ‘misused test and test-and-set’ bugs were filtered out through these techniques.

The time costs for these analysis techniques were not burdensome. For example, the four bug pattern detectors spent 8.35 s in total to analyze the seven file systems with vfs with multiple sub-pattern matching and all semantic analyses (last row and last column of Table 4) while they required 3.78 s with syntactic analysis for a single sub-pattern only. The maximum memory consumption was less than 50 MB in all experiments.

5.3. Bug detection results on device drivers and network modules

To investigate the general applicability of COBET, we applied the four pattern detectors for Linux file systems to other Linux modules. We targeted the seven modules in total including three Linux device drivers (bluetooth, ieee1494, and mtd) and four network modules (atm, ax25, netfilter, and rds(ib)). These target programs were implemented as loadable kernel module objects. Thus, the thread starting points of these modules are the function pointers registered at the module initializations.

Table 5 shows the bug detection results. The first number in each cell indicates the number of new bugs detected by COBET. The second number indicates the number of bugs that were confirmed as “real” by Linux maintainers. The three bug pattern detectors (‘misused test and test-and-set’, ‘unsynchronized communication at thread creation’, and ‘incorrect usage of atomic operations’) detected 13 bugs while ‘waiting for an already terminated thread’ detected no bug. Six bugs among these 13 bugs were confirmed as real ones by Linux maintainers.

Although the scope of this empirical study is limited, these results suggest that the bug patterns defined in one domain can be applied effectively to other domains and can help OS developers in practice. The following quotation is part of a response from a Linux maintainer to our ‘misused test and test-and-set’ bug report on netfilter.

Nice catch, this does indeed look like a bug. The entire locking concept seems a bit strange, we neither need an atomic_t for the reference count nor two locks to protect the list . . .

This bug report was immediately followed by the corresponding kernel patch. We received similar positive responses from other Linux maintainers regarding our bug reports and it indicates that the COBET approach can help kernel developers to detect subtle concurrency bugs in a practical manner.

5.4. Analysis on false alarms

To investigate the practical effectiveness of COBET, after we removed obvious false alarms through code review, we reported
the alarms raised by the COBET bug detectors to the corresponding Linux maintainers. We received 16 feedbacks for our bug reports on the file systems, the device drivers, and the network modules. Among these 16 feedbacks, 10 feedbacks confirmed that the corresponding alarms were real bugs. From the six negative feedbacks, we could identify the following sources of false positives:

- **Imprecise semantic analysis:**
  Although concurrent accesses to a shared variable can be carefully coordinated to prevent data race (i.e., by a lock-free algorithm), the COBET bug detectors may still report alarms on the concurrent access code. This is because the semantic analysis of COBET is not precise enough to recognize a lock-free algorithm and suppress false positives.

- **Limited alias analysis:**
  Since COBET performs lightweight alias analysis, it may fail to reason alias constraints in a complex data structure precisely and raise a false alarm. A bug pattern detector utilizes the alias analysis result to check if matched sub-patterns can interact with each other through a shared variable. Thus, imprecise alias analysis result may cause a bug pattern detector to conclude that two sub-patterns can conflict each other, although they cannot.

- **Path insensitive analysis:**
  COBET may report a false bug pattern matching for an infeasible execution path due to path insensitive analysis. For example, a bug pattern may match two statements guarded by two mutually exclusive branch conditions respectively. In this case, a matching for the bug pattern with these two statements should be ignored, since the two statements cannot execute in a sequence.

- **Unrealistic operation:**
  Although it was confirmed that some alarms could cause a problem in theory, Linux maintainers considered possibility for those alarms to make serious problem (i.e., crash Linux kernel) is very low in practice and ignored those alarms.

As we have described above, main causes of false alarms are due to lack of detailed understanding of semantics of target code. Thus, leveraging domain knowledge of developers will be important for the COBET approach to reduce false alarms and improve practical effectiveness of concurrency bug detection.

### 5.5. Comparison with Coverity Prevent

We applied Prevent to compare COBET with a conventional concurrency bug detection technique. Prevent has the following five pre-defined concurrency checkers ([Coverity, 2011]):

- **ATOMICITY checker** reports a bug if a shared variable that was updated while holding a lock is read after releasing the lock.
- **LOCK checker** checks pairing of a lock operation and an unlock operation within a function.
- **MISSING_LOCK checker** reports a data race bug for a shared variable that is not consistently guarded by a lock.
- **ORDER_REVERSAL checker** reports a deadlock bug when lock A is acquired while holding lock B and lock B is acquired while holding lock A in a program, since this program may result in deadlock due to cyclic dependencies between lock A and lock B.
- **SLEEP checker** reports a thread can invoke sleeping operations while holding a lock. A thread should release every held lock before sleeping; otherwise other threads can be blocked by acquiring a lock held by the sleeping thread.

We applied these five bug checkers to the file systems, network modules, and device drivers of Linux 2.6.30.4. For fair comparison with COBET, we configured Prevent to (1) enable function call via function pointers, (2) set bug sensitivity high so to report as many bugs as possible, and (3) utilize the specification of the lock operations in a target domain.

Tables 6 and 7 show the number of bugs reported by Prevent and the execution time for each target module. When we checked the file systems, we examine each naive file system together with VFS as we did with COBET. For example, ATOMICITY checker reported 45 bugs in btrfs and it took 47.72 s to apply all five bug checkers to btrfs (see Table 6). However, we did not validate these bug reports (i.e., to check whether a reported bug report is real or false), since communicating with developers to validate these bug reports would require large efforts and time.

First, we checked how many real bugs detected by COBET were also detected by Prevent. We found that Prevent did not detect any of the 10 real bugs detected by COBET. This result is not surprising, since Prevent handles only standard lock and recursive lock operations as synchronization operations and cannot analyze various synchronization operations correctly ([Coverity, 2011]). Therefore, this result confirms that COBET can detect concurrency bugs that cannot be detected by other conventional bug detection tool and serve as a complementary concurrency bug detection tool to support kernel developers.

Second, the analysis of Prevent was much slower than that of COBET. For example, Prevent spent 252.86 s to analyze the seven Linux file systems while COBET spent 8.35 s. Since bug checkers of Prevent and those of COBET are different and the bug detection algorithm of Prevent is not publicly available, it is difficult to identify the reasons for this performance difference precisely. Our conjecture for the performance difference is due to the fact that COBET performs path-insensitive analysis whereas Prevent does path-sensitive analysis ([Hallem et al., 2002]). In addition, COBET performs the syntactic pattern matching prior to semantic analysis and avoids analysis of irrelevant code consequence (see Section 3.3), which enables COBET to analyze a target code faster than Prevent.

### 6. Related work

Pattern based techniques ([Engler et al., 2000; Hallem et al., 2002; Hovemeyer and Pugh, 2004b; Otto and Moschyna, 2008]) can analyze large programs quickly, since these techniques perform pattern matching on a target program without sophisticated analyses. Engler et al. ([Hallem et al., 2002]) used a high-level

---

**Table 5**

<table>
<thead>
<tr>
<th>Device drivers</th>
<th>Network modules</th>
<th>Total (100KL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bluetooth (11KL)</td>
<td>imei394 (25KL)</td>
<td>std (15KL)</td>
</tr>
<tr>
<td>Misused test and test-and-set</td>
<td>0/0</td>
<td>1/0</td>
</tr>
<tr>
<td>Unsynchronized at thread creation</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>Incorrect usage of atomic operations</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>Waiting already terminated thread</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>Total</td>
<td>0/0</td>
<td>1/0</td>
</tr>
<tr>
<td>Time (s)</td>
<td>5.90</td>
<td>7.29</td>
</tr>
</tbody>
</table>
state-machine language Metal to specify system rules (i.e., programming idioms) over linear execution paths. They applied system rules such as a ‘holding lock’ rule (i.e., the acquired locks should be released before a function exits) to several operating systems and found bugs (Engler et al., 2000). However, they target sequential errors related to synchronization operations while COBET targets complex concurrency errors caused by thread interactions. Hovemeyer et al. (Hovemeyer and Pugh, 2004b) defined frequently observed Java concurrency bug patterns and analyzed the bytecode of the target Java program through code pattern matching. They found several concurrency bugs in Sun JDK 1.5 and an open source J2SE library. The false alarm ratio of simple bug patterns such as ‘double check’ that target sequential errors related to lock operations was less than 20%. However, the false alarm ratio of complex concurrency bug patterns (e.g., ‘wait not in loop’) was high, since (Hovemeyer and Pugh, 2004b) does not check thread interactions or semantic conditions. COBET targets complex concurrency bugs by utilizing multiple sub-patterns and semantic conditions. In addition, COBET helps engineers build bug detectors in a semi-automatic manner using PDL.

Otto et al. (Otto and Moschny, 2008) propose a bug pattern matching technique with semantic analyses on locks for finding concurrency bugs in Java programs. The idea of utilizing semantic information for better accuracy is similar to COBET. However, they do not provide a pattern description language, or support multiple sub-patterns.

Lock based techniques concentrate on lock usages. Lock based techniques effectively detect deadlocks (Engler and Ashcraft, 2003; Naik et al., 2009; Molnar and van de Ven, 2012; Agarwal et al., 2005) and low-level data races (Choi et al., 2002; Engler and Ashcraft, 2003; Erickson et al., 2010; Savage et al., 1997; Young et al., 2007) which occur only when no lock synchronizes multiple threads which read and update one shared variable. However, these techniques share the limitation when they are applied to OS codes which utilize various synchronization mechanisms other than lock.

Concurrency bug detection techniques that analyze stateful behavior of a target program detect violations of user-specified properties (i.e., assertions, invariants, or temporal logic formulae) by analyzing executions state by state, either by model checking (Musuvathi and Qadeer, 2007; Post et al., 2009; Qadeer and Wu, 2004) or by systematic testing (Farchi et al., 2003; Joshi et al., 2009). Nonetheless, the scalability of these approaches is still limited due to the state explosion problem. Thus, these approaches are still not capable of analyzing OS kernels in practice.

7. Conclusion
We have developed a pattern-based CONcurrency Bug dETector (COBET) framework for operating systems. To target complex concurrency bugs, COBET utilizes composite bug patterns and associates semantic information with code structures in bug pattern matching. While most concurrency bug detection techniques concentrate on lock usages, COBET targets various concurrency bug patterns specified by a user, so as to detect complex bugs. The effectiveness, efficiency, and applicability of COBET were illustrated by detecting ten new bugs in the file systems, device drivers, and network modules of Linux 2.6.30.4 with a modest cost. Although the bug detection of COBET is neither sound nor complete, the empirical results indicate that the COBET approach can detect concurrency bugs in large and complex programs practically.

Acknowledgement
This work was supported by Basic Science Research Program through the NRF funded by the MEST (2009-0064639) and the Excellent Research Center (ERC) of Excellence Program of Korea MEST/NRF of Korea (Grant 2012-0000473).

References

Table 6
Bug detection result by Prevent on the Linux file systems.

<table>
<thead>
<tr>
<th></th>
<th>btrfs (41KL)</th>
<th>ext4 (28KL)</th>
<th>nfts (29KL)</th>
<th>proc (8KL)</th>
<th>reiserfs (27KL)</th>
<th>sysfs (3KL)</th>
<th>udev (9KL)</th>
<th>vfs (48KL)</th>
<th>Total (103KL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOMICITY</td>
<td>45</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>62</td>
</tr>
<tr>
<td>LOCK</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>MISSING_LOCK</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>ORDER_REVERSAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLEEP</td>
<td>55</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>86</td>
</tr>
<tr>
<td>Time (s)</td>
<td>47.72</td>
<td>51.37</td>
<td>75.80</td>
<td>14.56</td>
<td>11.53</td>
<td>14.58</td>
<td>27.93</td>
<td>9.37</td>
<td>252.86</td>
</tr>
</tbody>
</table>

Table 7
Bug detection result by Prevent on Linux device drivers and network modules.

<table>
<thead>
<tr>
<th></th>
<th>bluetooth (11KL)</th>
<th>ieee802.11 (25KL)</th>
<th>std (15KL)</th>
<th>Network modules</th>
<th>Total (100KL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOMICITY</td>
<td>0</td>
<td>4</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOCK</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>MISSING_LOCK</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>21</td>
</tr>
<tr>
<td>ORDER_REVERSAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLEEP</td>
<td>0</td>
<td>7</td>
<td>14</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>0</td>
<td>7</td>
<td>14</td>
<td>13.23</td>
<td>9.18</td>
</tr>
<tr>
<td>Time (s)</td>
<td>9.12</td>
<td>29.78</td>
<td>17.27</td>
<td>99.33</td>
<td></td>
</tr>
</tbody>
</table>


Shin Hong received the MS degree in computer science from KAIST in 2010, where he is currently working toward the PhD degree in computer science. His research interest includes concurrent program testing, automated testing, and embedded software.

Moonzoo Kim is an associate professor of computer science department at KAIST, South Korea. His research interests are in automated software testing, systematic analysis of concurrent programs, and formal analysis of embedded software. He has focused on practical application of automated analysis techniques to improve the quality of industry software through close collaboration with companies such as Samsung Electronics.