The Journal of Systems and Software 86 (2013) 377–388

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Effective pattern-driven concurrency bug detection for operating systems

Shin Hong, Moonzoo Kim*

Computer Science Department, KAIST, South Korea

ARTICLE INFO

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

ABSTRACT

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.

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; Voung 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

* Corresponding author.
E-mail addresses: hongshin@kaist.ac.kr (S. Hong), moonzoo@cs.kaist.ac.kr (M. Kim).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.08.063

Table 1
Statistics on the synchronization statements in the Linux kernel 2.6.30.4.

	Atomic inst.	Cond. var.	Memory barrier	Mutex	rw sema-phore	rw spin lock	Sema-phore	Spin lock	Thread operation	Total
# of Stmt	8926	949	1926	14,902	2471	4248	759	44,205	460	78,846
Ratio	11.3%	1.2%	2.4%	18.9%	3.1%	5.4%	1.0%	56.1%	0.6%	100.0%

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 file 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-phores, 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 `nfs` and `ext4` 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.