ABSTRACT
With the development of multicore hardware, concurrent, parallel and multicore software are becoming increasingly popular. Software companies are spending a huge amount of time and resources to find and debug the bugs. Among all types of software bugs, concurrency bugs are also important and troublesome. This type of bugs is increasingly becoming an issue particularly due to the growing prevalence of multicore hardware. In this position paper, we propose a model for monitoring and debugging Starvation bugs as a type of concurrency bugs in multicore software. The model is composed into three phases: monitoring, detecting and debugging. The monitoring phase can support detecting phase by storing collected data from the system execution. The detecting phase can support debugging phase by comparing the stored data with Starvation bug’s properties, and the debugging phase can help in reproducing and removing the Starvation bug from multicore software. Our intention is that our model is the basis for developing tool(s) to enable solving Starvation bugs in software for multicore platforms.

CCS Concepts
Software and its engineering → Software system models; Formal software verification: Abstraction, modeling and modularity; Runtime environments;

Keywords
Starvation bug; multicore software; monitoring; debugging; concurrency bugs; parallel application; concurrent program.

1. INTRODUCTION
Multicore hardware and multicore software are two main categories in multicore technology. Multicore hardware is a processor that contains multiple cores in a chip and multicore software is typically a parallel application executing on multicore hardware. Multicore software is a field that is emerging from the necessity of obtaining a good performance on multicore processors. Using the potential advantages of multicore hardware is desired in multicore software field. However obtaining this aim brings some challenges such as designing concurrent and parallel software on multicore processors as well as testing and debugging on these systems. Parallel execution of multicore software makes them complicated, error prone and thus expensive.

Our previous investigation [2] indicates that researchers and developers are still faced with problems on concurrency software. Concurrency bugs are one of the most difficult types of bugs to detect, fix and reproduce [7], thus researchers have still challenge in proposing and improving solutions. Moreover, another challenge is the lack of reliable methods for monitoring, debugging and testing concurrent and multicore software [2]. In parallel and concurrent applications, Starvation is a condition in which a thread is indefinitely delayed because other threads are always given preference [9]. Starvation typically occurs when high priority threads are monopolising the CPU resources. During starvation, at least one of the involved threads remains in the ready queue.

Based on our investigation in [2], we found that debugging Starvation bugs compared to debugging other types of concurrency bugs have attracted less attention and there is a gap among the researches in the field. For instance, the number of published papers during 2005 to 2014 on Starvation bugs was 63 times smaller than the number of published papers with a focus on debugging Data race bugs and 15 times smaller than the Deadlock bugs. Although, our investigation in an open source software shows that some other type of concurrency bugs (e.g., Data race and Deadlock) are more severe than Starvation bugs [3] but we believe Starvation bugs still deserve to get more attention by proposing novel solution or significant extension to an existing technique specially with the focus on new demands in multicore software. This leads us to propose a novel model for debugging Starvation bugs (as one type of concurrency bugs) on multicore software in order to apply the model in designing and developing a tool(s) with the aim of fixing the Starvation bugs in multicore software as our future work. Due to the complexity of multicore software, it may be harder to detect potential concurrency bugs.
bugs in early stages of software life-cycle and such bugs can arise during system execution; thus software monitoring may address and alleviate this challenge by collecting, processing and measuring significant data at actual execution time.

In this article we particularly make the following main contributions: (1) we introduce Starvation bugs in multicore software and we explain two different Starvation bug examples. (2) We propose an integration model for fixing Starvation bugs. The proposed model is composed of three parts: Monitoring, Detecting and Debugging.

The remainder of this paper is organized as follows. We survey related work in Section 2. Some terminologies, the assumed hardware architecture and Starvation bug examples are presented in Section 3. We describe our proposed model in Section 4. Finally we conclude the study and highlight future directions of this work in Section 5.

2. RELATED WORK

Burky and Kalla present a patent and propose a method for detecting and handling a starvation of a thread in a multi-threading processor environment [5]. They believe that some of the available methods are too slow. In their proposed method first they define a counter for each thread while the counters are loaded with a pre-selected value. If a group of instructions from one thread not form the other threads has been completed then the value of associated counter for that thread will be updated by decrementing the counter stored value. If the group of instructions has not been completed for either thread then the counters will not change and they will be remained (or reload) with the previous counters stored value. The value of counters will check whether they reach to predetermined value or not. If the values are not equal to predetermined value then the previous steps will repeat otherwise a thread starvation condition may be detected. The comparison between predefined value and each thread counter defines which thread may be starved. Their method may detect the starvation bugs which causes the performance of the system and not the starvation bugs which cause nondeterministic output(s) while our proposed method can monitor, detect and debug the starvation bugs which cause nondeterministic output.

Moreover, Marwa et al. [8] believe that some of the concurrency bugs, Starvation and Deadlock bugs, should be identified in early stage of software life-cycle such as designing process. They use a search technique to propose a method for detecting Deadlock and Starvation bugs. Their proposed method requires a Unified Model Language (UML, sequence diagram) and Modeling and Analysis of Real-Time and Embedded Systems (MARTe) model of the system under test. Their method relies on a number of Genetic Algorithms (GAs), which directed at finding a particular concurrency bug to search for threads’ conditions by using the information available in UML/MARTe model. Their results show that a great deal of improvement could be obtained in runtime efficiency by using more powerful hardware and distributed GAs. Since their method is incorporated with design inspections therefore the design inspections might effect on detecting the concurrency bugs by this method.

In general, the current fault detectors cannot guarantee that no faults exist while one may feel confident that such a case is unlikely. The user may later uncover potential problems that can arise due to actual execution times of threads. Our proposed model considers this issue and can uncover potential problems that can arise due to actual execution times of threads by collecting, processing and measuring significant data at actual execution times.

3. PRELIMINARIES

In parallel and concurrent applications, Starvation is a condition in which a thread indefinitely delayed because other threads are always given preference [9]. Starvation bug typically occurs when high priority threads are monopolising the CPU resources. During Starvation, at least one of the involved threads remains in the Ready queue [2]. In general, four conditions should be fulfilled when a Starvation bug occurs [1]:

1. At least one of the threads is executing on one of the processor core.
2. At least one of the threads is in the Ready state.\(^1\)
3. The number of involved threads is larger than the number of free cores.
4. At least one thread is in the Ready queue for an unacceptably long time.

Starvation bugs on multicore software might happen because of CPU core availability, when the number of CPU core is not sufficient to execute the threads then an unexpected output (nondeterministic result) might be produce within an expected time-frame.

It should be noted that the terminology concerning software problems is not entirely consistent. In software testing, debugging and troubleshooting, different terms like fault, error, bug, failure, problems, anomalies, troubles, and defect exist and are sometimes used interchangeably. In this research we use the term bug while this may not be entirely in line with the above terminology, it is consistent with the terminology used in related work on concurrency bugs specially Starvation bug.

In this research, our focus is on Symmetric Multiprocessing (SMP) architecture (and not on Asymmetric Multiprocessing (AMP)). On SMP architecture the memory and I/O devices are shared equally among all of the processors [4]. It is more uniform and we believe that concurrency problems appear in a more similar way among SMPs than AMPS, which implies that articles relaying to concurrency in SMPs are straightforward to classify. In this SMP model the system have a single-chip multicore processor with “k” identical cores and two levels of cache\(^2\). Each core has its private level one cache, while the last level cache (LLC) is shared among all cores. We furthermore assume a single operating system managing resources and execution on all cores.

3.1 Starvation Examples on Multicore Software

We provide two starvation examples as a part of a multicore software and explain their execution scenario in this section. Figure 1(a) shows an example of a Starvation bug which is data race free and causes a nondeterministic output. In order to consider synchronization issues and avoid data race

\(^1\)Ready is a state when the thread is prepared (ready) to execute when given the opportunity.

\(^2\)Cache is “an area of memory that holds recent used data and instruction” [6].
Thread A

4: lock(customerName)  
5: lock(balance)  
6: customerName = read(customerName)  
7: Balance = 1000  
8: unlock(balance)  
9: unlock(customerName)  

Thread B

13: lock(customerName)  
14: lock(balance)  
15: customerName = read(customerName)  
16: Balance = 500  
17: unlock(balance)  
18: unlock(customerName)  

Thread C

26: file.write (customerName, balance)

(a) An example of Starvation bug causing unexpected (nondeterministic) output

Thread X

X2: Do  
X3: If (time.now = = timeSet) then
X4: sendSignal(alarmA, 1)
X5: else (time.now > = timeSet +10) then
X6: sendSignal(alarmA, 0)
X7: while (true)

Thread Y

Y2: Do  
Y3: If (time.now = = timeSet) then
Y4: sendSignal(alarmB, 1)
Y5: else (time.now > = timeSet +10) then
Y6: sendSignal(alarmB, 0)
Y7: while (true)

Thread Z

Z2: Do  
Z3: If (time.now = = timeSet) then
Z4: sendSignal(alarmC, 1)
Z5: else (time.now > = timeSet +10) then
Z6: sendSignal(alarmC, 0)
Z7: while (true)

(b) An example of Starvation bug causing performance problem

Figure 1: Starvation bug examples
Detecting phase is proposed to find the starvation bugs either on online mode (at software execution time) or on offline mode. It is decomposed into six steps: 1) comparing the properties of bug to Starvation bugs’ properties, 2) checking if a starvation bug has happened, 3) saving starvation bug log file, 4) checking if the user wants to fix the starvation bug at that time, 5) checking if the user wants to fix the bug through debugging steps and 6) applying appropriate mitigation method. As explained in Section 3 four properties should be fulfilled when a Starvation bug occurs. In this phase first, the saved data will be compared to these properties in order to find Starvation bugs [1]. If the saved data and starvation bug properties were matched then a starvation bug will be reported and a log record will be stored in a file. In this step the user becomes involved and plays an important role in order to decide whether to continue with Monitoring or Debugging phase. If the user wants to let the system continue its execution despite the occurrence of the bug (e.g., it is (still) deemed acceptable in terms of performance, etc.), then the Monitoring phase will not stop and execution of multicore software will continue (“Multicore software execution” step). On the other hand, if the detected bug requires to be fixed, the user may decide that mitigating the consequences of the bug would be possible without applying the debugging process. In this case the multicore software execution will not stop and instead an appropriate mitigation process will apply (such as software runtime reconfiguration, dynamically changing the priorities of tasks and threads, changing scheduling policy at runtime, etc.); otherwise the Monitoring phase will stop and Debugging phase can start.

Finally, Debugging phase is proposed to replay and debug the detected Starvation bugs. It is decomposed into four steps: 1) Replay the starvation bugs, 2) Cause identification, 3) Exploring correction and 4) Fixing the Starvation bugs. “Replay the Starvation bug” will start if the user wants to reproduce and debug the Starvation bug. Afterwards the root cause of the Starvation bug will be identified in “cause identification”. Then the potential solutions among all possible solutions for fixing the Starvation bug will be compared and the best one will be selected in “Exploring corrections”. Eventually, the process for repairing and fixing the Starvation bug will lead to remove the bug in “Fixing the Starvation bug”.

By this model we aim to tracing down the Starvation bugs that have caused a detected failure and replay the thread executions and the system behavior repeatedly.

5. CONCLUSIONS AND FUTURE WORK
In this paper, we introduced a model for detection of concurrency bugs through runtime monitoring, as well as for debugging and fixing those bugs. The proposed model provides a systematic way to address and tackle such bugs, which is currently lacking, and in practice testers mostly address concurrency bugs more or less the same way as other types of bugs (e.g., syntactic bugs). Our focus in this paper has been mainly on Starvation bugs. In our model, Starvation bugs are detected by checking if specific properties of these bugs, that distinguish them from other bugs, can be observed and derived from the collected runtime monitoring information. As a future work we plan to extend the model for detection of other concurrency bugs based on their distinct properties which we have already identified in [1].

Our model also acts as the basis for developing tool(s) to enable solving Starvation bugs in software executable on multicore platforms. Another interesting future direction of this work is to investigate the feasibility of automatic detection
and fixing of Starvation bugs without user intervention by continuing to execute the software and applying runtime re-configuration to fix the bug; or in a semi-automatic fashion by letting users know that the bug cannot be fixed automatically through re-configuration and requires debugging the code. There are other possible directions for the future work such as including static analysis techniques as part of the model to complement the monitoring part for detecting concurrency bugs; and also implementing the model as part of a framework.

6. ACKNOWLEDGMENTS

This research is supported by Swedish Foundation for Strategic Research (SSF) through the SYNOPSIS project.

7. REFERENCES


