Runtime Verification for Detecting Suspension Bugs in Multicore and Parallel Software

Conference Paper · March 2017

DOI: 10.1109/ICSTW.2017.20

3 authors:

Sara Abbaspour Asadollah
Malardalen University
13 PUBLICATIONS **56** CITATIONS
SEE PROFILE

Daniel Sundmark
Mälardalen University
80 PUBLICATIONS **538** CITATIONS
SEE PROFILE

Hans Hansson
Mälardalen University
165 PUBLICATIONS **3,873** CITATIONS
SEE PROFILE

Runtime Verification for Detecting Suspension Bugs in Multicore and Parallel Software

Sara Abbaspour Asadollah*, Daniel Sundmark*, Hans Hansson*
*Mälardalen University, Västerås, Sweden
{sara.abbaspour, daniel.sundmark, hans.hansson}@mdh.se

***Abstract*—Multicore hardware development increases the popularity of parallel and multicore software while testing and debugging these software becoming more difficult, frustrating and costly. 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. Suspension-based-locking bugs are one type of concurrency bugs.**

This position paper proposes a model based on runtime verification and reflection technique in the context of multicore and parallel software to monitor and detect suspension-based-locking bugs. This model is not only able to detect faults, but diagnose and even repair them. The model is composed of four layers: Logging, Monitoring, Suspension Bug Diagnosis and Mitigation. The logging layer will observe the events and save them into a file system. The monitoring layer will detect the presents of bugs in the software. The suspension bug diagnosis will identify the *Suspension* bugs by comparing the captured data with the suspension bug properties. Finally, the mitigation layer will reconfigure the software to mitigate the suspension bugs. A functional architecture of runtime verification tool is also proposed in this paper. This architecture is based on the proposed model and is comprised of different modules.

1. Introduction

Multicore software is typically defined as the parallel applications executing on multicore hardware. Since utilizing the potential advantages of multicore hardware is desired in the multicore software field, multicore software is emerging from the necessity of obtaining a good performance on multicore processors. Achieving this aim brings some challenges such as designing concurrent and parallel software on multicore processors as well as testing and debugging them. Parallel execution of multicore software makes them complicated, error prone and thus expensive.

Concurrent, multicore and parallel software introduce the possibility of new types of software bugs, known as concurrency bugs [1]. These types of software may exhibit problems such as race conditions and deadlocks that may not occur in sequential software. The errors typically appear under very specific (nondeterministic) thread interleavings between shared memory accesses. The effects of the bugs spread through the software until they cause the software to crash, become unresponsive (hang) or produce incorrect output. Such nondeterministic bugs are typically considered to be problematic errors [2], [3], [4]. Suspension-based-locking bug is one type of concurrency bugs which is briefly called *Suspension* bug in this paper. This type of bug can also occur on multicore and parallel software. It typically happens when a calling thread waits for an unacceptably long time in a queue to acquire a lock for accessing a shared resource [5].

A previous investigation [6] indicates that debugging *Suspension* bugs compared with debugging other types of concurrency bugs has not attracted attention and the current body of knowledge does not report studies on *Suspension* bugs during 2005 to 2014 and hence, there is a gap among the researches in the field. The existing gap in research study on *Suspension* bug may be due to the fact that this bug is not well-known yet, or identifying it is not an easy task. Although, the other investigation of an open source software [7] indications that *Suspension* bugs after *Data race* has occurred more than the other type of concurrency bugs and about 15% of the bugs belongs to the *Suspension* type. Accordingly, there is a need to propose novel solutions or significant extension to an existing technique especially with the focus on new demands in parallel and multicore software.

Due to the complexity of multicore software, it may be harder to detect potential concurrency bugs in the early stages of software life-cycle and such bugs can arise during system execution. Traditionally, verification techniques such as testing, model checking and theorem proving are used to increase the trust of the correctness of software. Some of these techniques typically need strong requirements like the existence of formal models and some of them could not cover all potential errors. In order to address these problems, runtime verification and reflection technique was developed [8]. This technique operates at runtime, which makes it possible for developer (tester or user) to react whenever a software behaves incorrectly. Since detecting and monitoring the faults at runtime would be possible with this technique thus it would be a suitable and interesting feature for multicore and parallel software with unexpected behavior or nondeterministic output.

This position paper points out the potential of runtime verification and more specifically of multicore and parallel software. Runtime verification and reflection may address and alleviate the mentioned challenges by collecting, processing and measuring significant data at an actual execution time. It is specifically focused on not only detecting *Suspension* bugs, but also finding the causes and outline research directions. It also is applicable to design and develop a tool with the aim of detecting and fixing the *Suspension* bugs in parallel and multicore software as the future work. The main contributions of this article are listed as follows: (1) *Suspension* bugs in multicore software are introduced and an example of *Suspension* bug is explained. (2) A runtime verification model for detecting and identifying *Suspension* bugs is proposed. The proposed model is composed of four parts: Logging, monitoring, Suspension bug diagnosis and mitigation. (3) A functional architecture of the runtime verification tool for detecting *Suspension* bugs is proposed.

Different module for a runtime verification tool is explained as a part of tool architecture. The proposed modules are *Periodic Request Module*, *Trace Module*, *Data Visualization Module*, *Monitor Module*, *Suspension Bug Diagnosis Module* and *Mitigation Process/Module*.

2. Preliminaries

In parallel and multicore software, a *Suspension-based locking or Blocking suspension* occurs when a calling thread waits for an unacceptably long time in a queue to acquire a lock for accessing a shared resource [5]. In general, four conditions should be fulfilled when a *Suspension* bug occurs [9]:

1) At least one of the threads is executing on one of the processor cores.
2) The number of requests for a specific resource is larger than the number of available resources of that type.
3) At least one of the threads has acquired a lock.
4) At least one thread is in waiting for an unacceptably long time.

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 the term bug is used 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 *Suspension* bug.

The hardware architecture focus of this study is on Symmetric Multiprocessing (SMP) architecture (and not on Asymmetric Multiprocessing (AMP)). On SMP architecture the memory and I/O devices are shared among all processors [10]. In this SMP model the system have a single-chip multicore processor with "k"' identical cores and two levels of cache[1]. Each core has its private level one cache, while the last level cache (LLC) is shared among all cores. Furthermore a single operating system managing resources and execution on all cores is assumed in this study.

2.1. Suspension bug example on multicore software

In this section, one *Suspension* bug example, as a part of a multicore software and its execution scenario is presented. Figure 1 shows an example of a *Suspension* bug which is data race free and causes an unexpected output. In order to consider synchronization issues and avoid data race bug (data race free) this example is given by using the lock mechanism. It supposes to save all updates by Thread M and N into a file by Thread P. The updated values belong to two shared variables (*customerName*, *Balance*). Thread P is a separated thread for recording the history of updated data by other threads (M and N). One scenario for executing these three threads as parts of multicore software would be: suppose all three threads have the same priority and the initial values for (*customerName*, *Balance*) are *Null* and *0*, respectively. There are three free cores available (Core1, Core2 and Core3) and Threads M, N and P are executing on

1. Cache is "an area of memory that holds recent used data and instruction" [11].

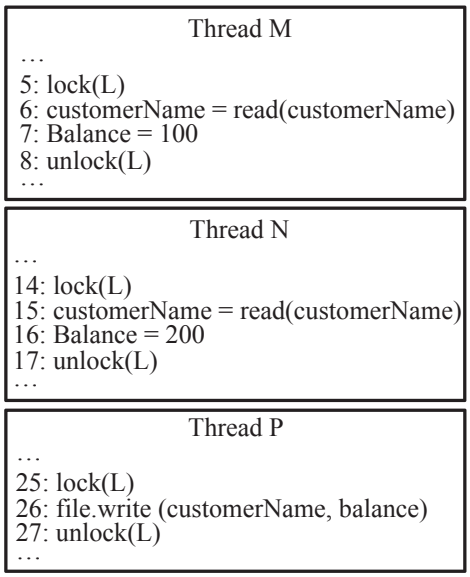

Figure 1: A *Suspension* bug example on multicore software

Core1, Core2 and Core3, respectively in parallel. If Core1 reaches line 5 before Core2 reaches line 14 and Core3 reaches line 25 then the *customerName* will read from I/O and store the value (which is given by user) to DRAM, LLC and L1 Cache of Core1 after executing line 6. Then by executing line 7, Core1 will update the *Balance* value (with 100) by storing to L1 Cache of Core1, LLC and DRAM. On the other hand, while Core1 is executing these lines (5 to 7), Thread N and Thread P will stay in Blocked queue until Core1 reaches to line 8 and release the lock. Core1 will continue to execute other commands from Thread M. When Core1 reaches line 8 and if Core2 reaches line 14 before Core3 reaches line 25 (assuming that Thread N and P are running on Core2 and Core3, respectively) then the *customerName* will read from I/O and store the value (which is given by user) to DRAM, LLC and L1 Cache of Core2 after executing line 15. Then by executing line 16, Core2 will update the *Balance* value (with 200) by storing to L1 Cache of Core2, LLC and DRAM. While Core2 is executing these lines (14 to 16), Thread P will stay in Blocked queue until Core2 reaches to line 17 and release the lock. Thus, the updated value by Thread M cannot store into the file because Thread P could not execute and the data was corrupted by Thread N.

Other scenarios which work perfectly and save the updated data into the file at the right time without any data corruption are available for this example. However, the explained scenario shows at least one situation, which causes a type of concurrency bugs (*suspension* bug) for parallel and multicore software.

3. Runtime Verification and Reflection Model for Detecting the Suspension Bugs

As explained in the prior sections, this study is based on runtime verification and reflection technique that respectively monitor multicore and parallel software in order to detect suspension bugs. Leucker and Schallhart defined the runtime verification [12] as "the discipline of computer sci-

ence that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property". In other words, runtime verification is an analysis and execution approach based on extracting information from a running system and using it to detect, and possibly react to, observed behaviors satisfying or violating certain properties.

The logical architecture of a multicore software following the runtime reflection pattern is shown in Figure 2. It is decomposed into four layers viz., Logging, Monitoring, Suspension Bug Diagnosis and Mitigation.

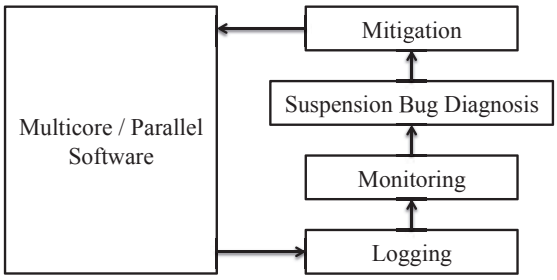

Figure 2: Architecture of the runtime verification framework for detecting *Suspension* bugs

The Monitoring layer is followed by a logging layer. Logging layer will observe the multicore system events and record the events data, which would be suitable for monitoring layer. The Logging layer can be implemented as separated stand-alone loggers (application) or it can be implemented as part of the multicore software by adding code annotations to the software.

The Monitoring layer is the heart of the architecture. It considers as fault detection and could consist of a number of monitors that observe the stream of multicore software events provided by the logging layer. It will detect the presence of bugs in the software without affecting its behavior. If a violation of a correctness property is detected in the multicore software then the monitor will respond with an alarm signal/message for subsequent diagnosis.

The Suspension Bug Diagnosis layer will not directly communicate with the application. It will collect the results of the Monitoring layer and will deduce an explanation for the current multicore software state. In other words, this layer will compare the results of Monitoring layer to the suspension bug properties.

The Mitigation layer will reconfigure the multicore software in order to mitigate the suspension bug (if possible). This layer will use the results of the Suspension Bug Diagnosis layer (identified suspension bugs) and will re-establish a determined system behavior. However, the identified cause may not always be possible to re-establish a determined system behavior. In this instance, a recovery system may save the detailed diagnostic data to use offline mode in other software life-cycle phase such as debugging.

4. Runtime Verification tool architecture for detecting the suspension bugs

An overview of the runtime verification tool architecture for detecting *Suspension* bugs is proposed in this section. As it is shown in Figure 3, the functional architecture tool is comprised of six separate modules, viz., *Periodic Request Module*, *Trace Module*, *Data Visualization Module*, *Monitor Module*, *Suspension Bug Diagnosis Module* and *Mitigation Process/Module*. All modules within gray rectangle are executed after every "n" ms, while all other modules are executed upon users request.

The *Trace Module* will be enabled with a single link time option given by *Periodic Request Module*. *Periodic Request Module* is responsible for sending the request to capture the data every "n" ms. The "n" is a value which is defined by user before the tool starts working. The *Trace Module* will observe and record the utilization for each active thread into a buffer. Each record of buffer, represents "n" ms of wall-time, contains the list thread with executing state, the list of threads with waiting state and list of waiting reason.

The observed raw data will store into the memory buffers when the multicore software is executing. The buffered data will be written out to a file system (log file) when buffers fill or when the software terminates. The logs can then be manipulated, displayed and post-processed within other modules of the tool, i.e., monitoring, diagnosis and mitigation. Detailed log files store chronological and time stamped recording of runtime execution metrics within a requested time encountered in the software. These log files tend to be large thus care has to be taken for producing an effective analysis and detecting the suspension bugs.

The detected suspension bug data gathered in this tool can describe the fraction of the execution time spent in each activity over some period of time.Some fine-grained data on the execution activities will be lost if the user chose a big number (n) for snapshot, however user can repeat the logging and saving log files processes within smaller snapshot for another round. In other words, the user can set a suitable value for a variety of analysis tasks such as *Monitoring* and *Suspension Bug Diagnosis*.

Monitoring Module will check the stream of the extracted data saved in log files to detect the presence of bugs in the software without affecting the software's behavior. In order to identify the type of bug(s) provided by *Monitoring Module* and ensure that the bug(s) is a suspension bug, the *Suspension Bug Diagnosis* will compare the bugs properties to the suspension bug properties. If these data were exactly mapped then a suspension bug(s) will identify and report.This operation will be performed as follows: all threads with executing state within specific snapshot will extract from the log file. On the other hand, all threads with waiting state within specific snapshot will extract from the log file. Among the waiting threads, the reason of waiting will be considered. If the reason(s) is due to lock mechanism, then a counter will count the total number of threads and just these threads will consider for the rest of the processes. If the waiting time for a thread is longer that the user tolerance time (user already specified the tolerance for accepting delay during execution time) and if at least one of the threads is executing on one of the processor cores during the snapshot, then the *Diagnosis Module* will identify a *Suspension* bug and will save the thread(s)s properties which cases the *Suspension* bugs. The *Visualization Module* will also show a list of thread(s) which have identified in *Suspension Bug Diagnosis Module*. The *Mitigation Process/Module* will apply an appropriate mitigation process. This process can change the priorities of threads and tasks dynamically, change scheduling policy at runtime, reconfigure the software at runtime, or other

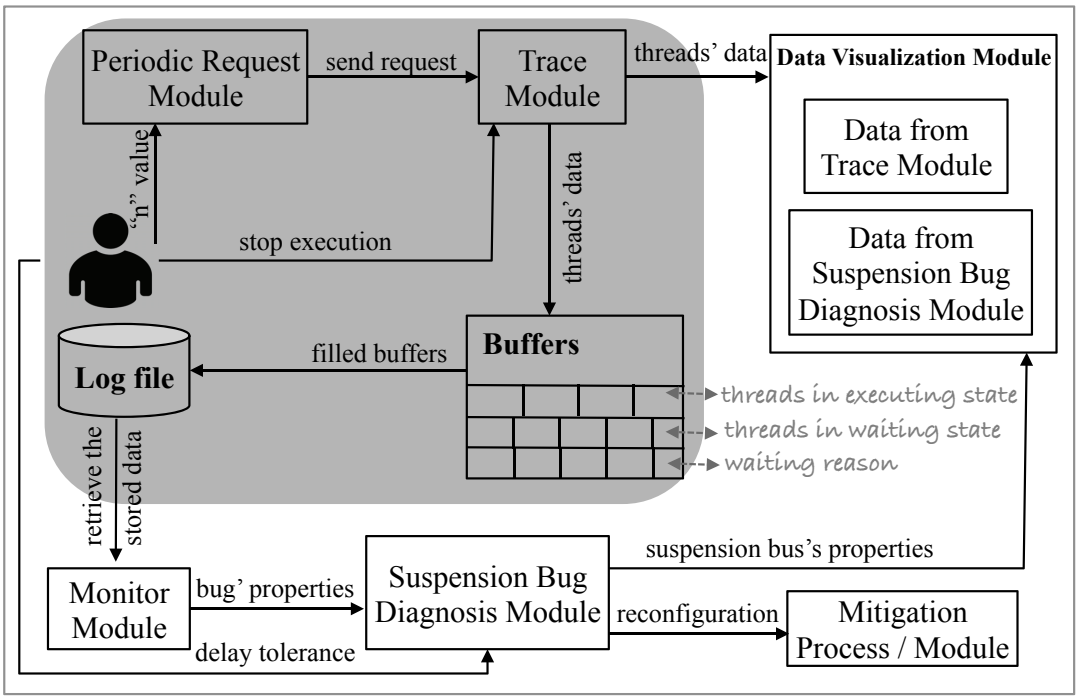

Figure 3: A functional architecture of the proposed runtime verification tool for detecting suspension bugs

re-establishing process. It is worth noting that based on the diagnosis and the occurred bug; it may not always be possible to reconfigure the software to mitigate the bug.

5. Conclusions and Future work

This paper introduced an architectural model for runtime verification of one type of concurrency bugs. The focus of this paper has been mainly on *Suspension* bugs and the proposed model provides a systematic way to detect and tackle *Suspension* bugs. In this model the *Suspension* bugs are detected by checking if specific properties of these bugs can be observed, monitored and derived from the collected runtime verification information.

The proposed model acts as the basis for developing a tool to detect and identify the *Suspension* bugs on multi-core platforms, thus a functional architecture of a runtime verification tool is also proposed in this paper. The implementation of the tool would be based on the proposed model and is considered as future work. Another future direction of this work is to extend the model and tool for detection of other concurrency bugs based on their distinct properties which are already identified in [9]. There are other possible directions for the future work such as implementing the model as part of a framework.

Acknowledgment

We acknowledge the Swedish Research Council (VR, EXACT project) for supporting this work.

References

[1] D. A. Weiser, *Hybrid Analysis of Multi-threaded Java Programs*. ProQuest, 2007.

[2] M. Süß and C. Leopold, "Common mistakes in OpenMP and how to avoid them," in *OpenMP Shared Memory Parallel Programming*. Springer, 2008, pp. 312–323.

[3] P. Godefroid and N. Nagappan, "Concurrency at Microsoft: An exploratory survey," in *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[4] J. Desouza, B. Kuhn, B. R. De Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, scalable debugging of MPI programs with Intel Message Checker," in *Proceedings of the second international workshop on Software engineering for high performance computing system applications*. ACM, 2005, pp. 78–82.

[5] S. Lin, A. Wellings, and A. Burns, "Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 16, pp. 2227–2251, 2013.

[6] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, H. Hansson, and W. Afzal, "10 years of research on debugging concurrent and multicore software: a systematic mapping study," *Software Quality Journal*, pp. 1–34, 2016.

[7] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, H. Hansson, and E. Paul Enoiu, "A study of concurrency bugs in an open source software," in *International Conference on Open Source Systems (OSS)*, 2016.

[8] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, 1999, pp. 114–122.

[9] S. Abbaspour A., H. Hansson, D. Sundmark, and S. Eldh, "Towards classification of concurrency bugs based on Observable properties," in *International Workshop on Complex Faults and Failures in Large Software Systems*, Italy, 2015.

[10] R. Brown, "Method and apparatus for processing requests for video presentations of interactive applications in which vod functionality is provided during nvod presentations," Jun. 23 1998. [Online]. Available: https://www.google.com/patents/US5771435

[11] D. Gove, *Multicore Application Programming: For Windows, Linux, and Oracle Solaris*. Addison-Wesley Professional, 2010.

[12] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293 – 303, 2009.