applied sciences

MDPI

Article

Atomicity Violation in Multithreaded Applications and Its Detection in Static Code Analysis Process

Damian Giebas † and Rafał Wojszczyk *,†

Faculty of Electronics and Computer Science, Koszalin University of Technology, Sniadeckich 2, 75-453 Koszalin, Poland; damian.giebas@s.tu.koszalin.pl

* Correspondence: rafal.wojszczyk@tu.koszalin.pl

† These authors contributed equally to this work.

Received: 24 October 2020; Accepted: 10 November 2020; Published: 12 November 2020

Abstract: This paper is a contribution to the field of research dealing with the parallel computing, which is used in multithreaded applications. The paper discusses the characteristics of atomicity violation in multithreaded applications and develops a new definition of atomicity violation based on previously defined relationships between operations, that can be used to atomicity violation detection. A method of detection of conflicts causing atomicity violation was also developed using the source code model of multithreaded applications that predicts errors in the software.

Keywords: multithreaded application; resource conflicts; atomicity violation; pthread; parallel computing

1. Introduction

Software errors exist as long as the software itself. The cause of most errors in multithreaded applications is the so-called resource conflict, while the result is undesirable phenomena, such as race condition. Race condition and deadlock are by far the most common phenomena that occur in multithreaded applications. In comparison to the phenomenon of atomicity violation, race condition and deadlock are easy to locate. They result directly from the code structure of the application [1,2], as opposed to the atomicity violation phenomenon, which is detailed in Section 4. However it was important to start studies on atomicity violation, and they had to be started from less complicated phenomena. Race condition and deadlock were the best case study. Race condition and atomicity violation belong to these same type of phenomena and the incorrect fixing of both of them may lead to deadlock. In other words resolving the problem with race condition phenomenon and deadlock phenomenon allow for a better understanding of how to fix data race errors and without adding new ones.

Multithreading is one of the basic mechanisms for parallel computing, which is possible on one processor and on several processors [3,4]. It is used very often in utility applications (mentioned in the rest of the work), but also in scientific research [5]. Popular, commonly used open-source multithreaded applications are a very good source for research on undesirable phenomena in multithreaded applications. In a popular Apache server, atomicity violations have been located several times, with the examples described in the works concerning buffer handling [6,7]. The effect of an incorrectly written function was, in the worst case scenario, the unexpected termination of the application. The situation was similar for the MySQL relational database management system. The state of the log file was stored in the form of a global variable, which was used by several operations [6] constituting a logical whole.

The problem of atomicity violation concerns not only open-source programs. Various JDK packages were also tested and in version 1.4.2 of this package it was discovered that the handling of five popular containers had errors in its implementation that caused the atomicity violation

phenomenon [8]. Four years earlier, a conflict causing an atomicity violation was discovered in a *StringBuffer* implementation in the Java standard library [9]. Despite the high popularity of Java, frequent use in created network projects [10], or huge support from the community, the above examples show how important it is to avoid multithreading errors. This then forces developers to look for solutions to optimize their code to help them prevent these phenomena.

The development of static code analysis method for detecting phenomenon of atomicity violation is important for a few reasons. First of all, testing and dynamic methods known from the literature are insufficient. Effective static analysis can be part of the compiling process, because it is easy to convert Abstract Syntax Trees into any other representation and make an architecture check. Dynamic analysis can affect applications during working and can add other hard-to-find errors. On the other hand, fixing bugs in the code allows us to use every compiler instead of those introducing mechanisms for dynamic analysis.

The next section contains a detailed description of the atomicity violation phenomenon. Section 3 describes known methods and programs that allow for detecting the atomicity violation phenomenon. Section 4 contains a description of relations whose violation causes the atomicity violation phenomenon. This is the original part of research on this phenomenon. Section 5 provides information about how the source code model of multithreaded applications has been extended to detect the atomicity violation phenomenon. The problem was formulated in Section 6 preceding Section 7 with a sufficient condition. Sections 8 and 9 contain a leading example and a summary, respectively.

2. Description of the Phenomenon

The phenomenon of atomicity violation is a result of inconsistent order of access to data [11] as shown in Figure 1. Most often it refers to a situation in which the programmer intended to execute two instructions in series, excluding other instructions with access to a shared memory area, and yet another instruction affects the memory area of these two functions [12]. In this work atomicity violation is a phenomenon resulting from violating the execution of a pair of operations (forming a logical whole) on the shared resource, by an operation of another thread operating on that resource. These operations in the source code of the program do not have to occur directly after each other—i.e., there may be other operations between them that do not affect the shared resource in any way. The operation is to be understood as any C language instruction.

The phenomenon of atomicity violation can be described as a situation in which there is a scenario of two threads working, the course of which deviates from the scenario expected by the programmer.

The phenomenon of atomicity violation is closely related to the phenomenon of race condition, because in both cases both threads should be mutually exclusive when working on a shared resource. The exclusion of threads takes place by placing the operations performed on the resource in the critical section, where the critical section is understood as a set of operations performed between the locking and unlocking mutex.

In [13], the atomicity violation phenomenon is also called "high-level data races". However, in the case of atomicity violation, the critical section should include a pair of operations carried out on a shared resource. In both cases, however, the order in which the operations are performed affects the final result.

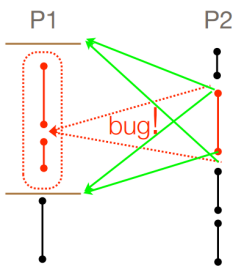

Figure 1. Visualization of the atomicity violation phenomenon. Source [14].

It can also be said that the phenomenon of race condition is a special case of atomicity violation. This is because there are instructions in C and C++ languages, which at the programming language level are atomicity operations—i.e., consisting of one language instruction (e.g., an incrementation operation). However, the situation changes when the code is compiled. *The time needed to retrieve a word value from memory to cache and then to register can be (in terms of processors) very long. At best, about 500 instructions can be executed before the value reaches the registry, and it will take the same amount of time to pass the new value to its destination* [15]. When the atomicity violation occurs at the programming language level, the symptoms may include:

1. Undefined application behavior;
2. Incorrect application behavior.

In works on atomicity violation, a thread in which a pair of operations is performed (to form a logical whole) is called local, and a thread with a violating operation is called the remote thread.

Chew and Lie [16] describing the phenomenon of atomicity violation, stress that this phenomenon may be required, harmless and undesirable [16]. An example of a required violation of atomicity is a situation in which two threads communicate by passing information using a shared resource.

This example is a part of the scenario described in the first column in the second row of Table 1. However, this example is incorrect because the pair of operations of the local thread requires a record operation in the remote thread to work correctly. To sum up, the example scenario described by [16] does not meet one of the conditions of atomicity violation—i.e., the remote operation does not interfere with the work of local pair of operations. On the contrary, if the remote thread records are not performed, the local thread will be affected. Therefore, it seems reasonable to question the claim that there are scenarios in which atomicity violation is required. If two operations in the indicated memory area are to be performed without any interference, the interference may be neutral or undesirable. If the interference is required, its non-occurrence causes the application to malfunction, but it is not an atomicity violation. It is worth adding, however, that the discussed example will be taken into account during static analysis of the application code, as a result of which *false-positive error* will be reported.

Table 1. Scenarios of the atomicity violation. Source [16].

$read_{local}(A)$	$read_{local}(A)$
$write_{remote}(A)$	$write_{remote}(A)$
$read_{local}(A)$	$write_{local}(A)$
$write_{local}(A)$	$write_{local}(A)$
$write_{remote}(A)$	$read_{remote}(A)$
$read_{local}(A)$	$write_{local}(A)$

In the program described below, called AV1 (https://bit.ly/2QBlVOC), there is a conflict of resources causing atomicity violation. The *usleep* operation present in the application, which suspends the thread for 100 milliseconds (the equivalent of request handling in a server-type application) is intended to increase the probability of observing atomicity violation—i.e., the probability of observing atomicity violation increases at the moment when the execution time of operations between a pair of operations performed on a shared resource, which form a logical whole, increases. According to the definition, a logically atomic pair of operations that should not be affected is an operation of incrementing shared resource $r1$ and printing the value of this resource on the standard output. These operations are not in the same critical section, which leads directly to an atomicity violation. In this case, unwanted application behavior will result in the user console displaying twice the same value of the shared resource. However, this phenomenon is undesirable when the application user has the ability to view all values printed on the standard output—e.g., on the console. However, if these values were displayed, for example, on a seven-segment display, the user might not notice that the same value is displayed twice. Therefore, whether the phenomenon of atomicity violation is neutral or undesirable may depend on the case of using a given application.

3. Existing Solutions

There are two different patents for detecting atomicity violations. The first one consists in the analysis of processor instructions and simulation of selected scenarios [17]. The second one describes a method of detecting the atomicity violation phenomenon called "access interleaving invariants" (AII) [12]. The method described there was originally implemented in AVIO [18] and SVD tools [19], and a few years later it was also used in the CTrigger framework [7].

Park et al. [7] also criticized the methodology of searching for atomicity violation through stress tests. Its determinism was indicated as a disadvantage of this solution. Although multithreaded applications are only seemingly non-deterministic, it is precisely this apparentness that makes it impossible to cover all possible states of a multithreaded application with deterministic tests.

The process of static analysis of the code for searching for atomicity violations has also been criticized, indicating that these methods report *false-positive error* in many places [7,20]. As a solution to this problem, it has been suggested to perform tests or use separate tools to investigate *false-positive error* [7]. It was also pointed out that there are scenarios in which CTrigger requires huge resources and has to work many hours to detect the phenomenon of atomicity violation. *For example, 100 different 1-h (within CTrigger) input tests and 10 different configurations would take CTrigger 2 days to complete testing on 20 machines, while stress tests would take between 20 and even 2000 days to achieve a similar ability to expose atomicity violation that is far too long to be acceptable* [7]. Although two days of operation of 20 machines seems a better solution compared to the stress tests, it is still unacceptable in most projects. The process of static code analysis, in order to detect atomicity violation conflicts, should be much faster and cheaper compared to long-term analysis of application behavior. Despite the imperfections of such a process and many *false-positive errors* reported, a programmer should be able to verify a false report.

Flanagan and Freund [9] presented the Atomizer tool. Despite promising results, the authors of Atomizer decided to focus on hybrid methods that combine static code analysis with the "on-the-fly" analysis method they developed.

The use of the method based on a system of types was repeated by another team [21]. Sasturkar et al. [21], as with the authors of Atomizer, bypassed one of the limitations of the developed method, so that it is no longer limited to the compilation process. The authors also mention that this method allows us to omit some conflicts that cause atomicity violation, but instead the amount of *false-positive error* is marginal.

Wang et al. [22] developed a model called "trace-based symbolic predictive model", which was used to develop a mixed method based on tracking specific application execution paths and analyzing the source code of that application to detect an atomicity violation. The authors claim that the method presented by them does not report any *false-positive error*, but they do not guarantee that all conflicts causing the atomicity violation are detected.

Solutions based on the analysis of the application during its operation, to avoid the phenomenon of atomicity violation, are applied regardless of the language in which the programs are written. With the growing popularity of JavaScript [23], this language experienced the use of Node.js runtime environment, which is used for writing server applications. In this environment, applications are created with the help of an event programming paradigm, in which many operations are performed in parallel in different threads [24]. However, this environment does not provide tools for marking operations as atomic [24]. Nor does Rust, which was developed for secure multithreading and published in 2010 [25]. The fact that the phenomenon of atomicity violation is language-independent and accounts for 70% of all multithreading errors [7] (without deadlock) shows how dangerous this phenomenon is.

In [16], a paper was published describing the Kivati tool for Linux and x86 platform, which aims to detect atomicity violations in applications written using C language. This tool interferes with the source code of the system kernel in order to use so-called watchpoints to observe resources that have been used in atomic regions. These regions are obtained during the analysis of the program whose operation is observed by Kivati. The fact of observing the operation of the program (as well as

influencing its operation) classifies the method used as a mixed one, which contradicts the information provided by the authors that the method used belongs to static methods [16].

Java grammar allowed [26] to develop a method of static analysis of the Java code allowing one to detect the phenomenon of atomicity violation. Their method described in the work entitled [26] is based on locating in the definition of classes of methods with the keyword *synchronized* in the declaration, whose structure is then analyzed for the occurrence of patterns that guarantee the phenomenon of atomicity violation [26]. The disadvantage of this method is that it cannot be translated into languages differing in similarity to Java, including the C language.

The method called "Hybrid Atomicity Violation Explorer" (HAVE) [27] is based on the use of the so-called "Static Summary Tree", created during static analysis of Java code. Then, based on the created trees, the work of selected fragments of the application is simulated. The effects of the simulation and the created trees are the basis for building the so-called hybrid trees, which are then analyzed using the author's algorithm, which results in information about potential conflicts that cause the atomicity violation and race condition. The authors of the method mention the occurrence of *false-positive error* in the results of their method and further work on their elimination. They also indicate further development of static code analysis as one of the most important elements of the developed method.

All described methods today can be included into few groups. The first group contains applications which are not available publicly from different reasons. The second group contains applications for other languages or for application which are not considered in the scope because they used libraries other than pthread. The third group contains methods which were developed for needs of specific languages and cannot be implemented in C language easily. For these reasons, there is no any other static analysis method which can be compared directly with the method proposed later in this paper. There is also no other known database with well described known bugs where C applications with pthreads library can be found.

4. Agreement Relationships between Operations

Relationships between operations, which can be read from the source code of any language, are limited to the order in which the operations are performed. According to the definition, the phenomenon of atomicity violation is a result of a violation of an operation pair on a specified memory area—i.e., on a shared resource occupying a specified area. It is not a disruption of the order of these operations, so one may suspect that there is a certain relationship between the pair of operations that is being violated; as a result of which, the program does not work properly. As a result of the analysis of the atomicity violation phenomenon, three relations have been developed, the violation of which leads to the atomicity violation phenomenon. Thus, if a program has two operations a and b, where $a \prec b$ and these operations form a logical whole, the following relationships may occur between these operations:

- Forward—a relationship in which the operation a always must be followed by an operation b;
- Backward—a relationship where operation a always has to precede operation b;
- Symmetric—a relationship where operation a always has to precede operation b, and operation b always has to follow operation a.

Despite a large number of studies on the phenomenon of atomicity violation, none of the works discuss the relationship between operations. The existing programming languages do not have any mechanisms that would allow for the declaration of such relations. However, the programmer may use a locking mechanism which allows one to secure the desired operations from violating the relations.

Complementing the source code with information about these relationships should allow for faster locating of the atomicity violation during the static code analysis process, or at least reduce the number of *false-positive errors* reported. An alternative solution is to enrich the process of converting the code into a model with the possibility of accepting such data as an input.

In the review of literature and technical docs, no mechanism has been found so far in programming languages, by means of which it is possible to declare the relations presented in this point. Knowledge about relations can be found in libraries' documentation or by asking programmers, and that is why they are referred to as agreement relationships.

Lack of knowledge of the relationships developed is the reason why programmers do not understand the phenomenon of atomicity violation. If these relationships could be defined by means of programming language mechanisms, it would be much easier to locate atomicity violation. It is therefore reasonable to draw up a new definition which will have information about the relationships whose disruption cause phenomenon of atomicity violation. Pairs of operations in relation to each other may be located by the Kivati program as components of atomic regions which are located for dynamic analysis. In the proposed solution, instead of checking how the code of the atomic regions works, the structure of the code is more important. It allows for checking only the related operation pairs and skips other operations which speed up process of analysis.

Taking into account the mentioned gap in the literature, it is worth presenting the following definition of the phenomenon of the atomicity violation.

Definition 1. *Atomicity violation is a phenomenon in which there is a relationship between two operations* $(o_{i,j}, o_{a,b})$ *of one thread, using a common resource* r_c*, whose disruption caused by the operation of another thread on the same resource (unexpected change of resource value) results in the undefined behavior of the algorithm using those operations.*

The operations should be understood as instructions or functions called in the program code. Sometimes the relation between two operations may result from the context and may not be a rule—e.g., setting a new value to one shared resource involves resetting the shared counter, but only when this value is set in the selected thread. Taking such cases into account is associated with an increase in the number of *false-positive errors* reported.

The C language standard library has many defined functions that can be included in one of the three relationships mentioned above. Most of the relations presented in Table 2 result directly from C language documentation. In the case of *calloc*, *free* and *malloc*, *free* pairs, the free relation results from the way the memory is managed in C.

The relations shown in the table are only a small subset of all possible relations that may occur in multithreaded applications. The C language standard library provides only basic mechanisms to allow the programmers to build their own, more complex mechanisms. For this reason, applications written using the C language are very extensive.

Therefore, in order to detect atomicity violation, programmers must determine themselves, in which C language functions, functions of external libraries and functions created by themselves exist in relation to each other.

Table 2. Examples of relations developed on the basis of the C language standard library function description [28].

Forward	Backward	Symmetric
calloc i free	fgetpos i strerror fsetpos i strerror ftell i strerror atof i strerror strtod i strerror	va_start i va_arg
malloc i free	strtol i strerror strtoul i strerror calloc i realloc malloc i realloc srand i rand	va_arg i va_end

It should also be remembered that the relations presented in Table 2 can be violated, because not all violations of the relations causing atomicity violation are undesirable—i.e., the effect of violating the relations can be harmless.

To eliminate the phenomenon of atomicity violation, locks are used to create critical sections. None of the programming languages have other mechanisms that would support defining sets of operations as a logical whole. This also means that the C and the *pthread* do not provide mechanisms to declare the relationship between two functions.

C language does not provide any mechanisms to declare the relationship between any pair of two operations in a thread. The only reason for an existing relationship may be a structure in which two operations of the same thread use the same shared resource. This assumption, will cause a very large amount of *false-positive error* to be reported. To reduce the number of *false-positive error* when converting the source code of a multithreaded application to a model, it is necessary to develop a relation table similar to Table 2, which will contain relations between user-defined functions.

The scenarios presented in Table 1 concern the relations that can occur between the triple operation of two threads. In fact, in any advanced multithreaded program there are many such relations.

5. Multithreaded Application Source Code Model

Locating an atomicity violation is possible using the source code model of a multithreaded application [2]; however, this model should be extended with a set that will describe the relationship between the two operations. This model looks as follows:

$$C_P = (T_P, U_P, R_P, O_P, Q_P, F_P, B_P) \tag{1}$$

where:

1. P is the application index;
2. $T_P = \{t_i | i = 0...\alpha\}$, $(\alpha \in \mathbb{N})$ is a a set of threads t_i of C_P application, where t_0 is the main thread $|T_P| > 1$;
3. $U_P = (u_b | b = 1...\beta)$, $(\beta \in \mathbb{N}^+)$ is a sequence of u_b, sets which are subsets of T_P containing threads working in the same period of time in the application C_P, whereas $|U_P| > 2$, $u_1 = \{t_0\}$ and $u_\beta = \{t_0\}$;
4. $R_P = \{r_c | c = 1...\gamma\}, r_c = \{v_1, v_2, ..., v_\eta\}, (\gamma, \eta \in \mathbb{N}^+)$ is a collection of shared application resources C_P, and the subsequent elements are sets of variable names relating to a single resource;
5. $O_P = \{o_{i,j} | i = 1...\delta, j = 1...\epsilon\}$, $(\delta, \epsilon, \in \mathbb{N}^+)$ is a set of all C_P application operations which are atomic at a certain level of abstraction—i.e., dividing them into smaller operations is impossible (it should be understood as an instruction or function defined in a programming language); Index i indicates the number of the thread in which the operation is performed, and index j is the ordinal number of operations working within the same thread;
6. $Q_P = \{q_s | s = 1...\kappa\}, q_s = (w_s, x_s), (\kappa, \in \mathbb{N}^+)$—a set of all locks available in the program, defined as a pair of variables, type of lock, where the type is understood as one of the set values (PMN, PME, PMR, PMD);
7. $F_P = \{f_n | n = 1...\iota\}$ and $F \subseteq (O_P \times O_P) \cup (O_P \times R_P) \cup (R_P \times O_P) \cup (O_P \times Q_P) \cup (Q_P \times O_P)$, $(\iota \in \mathbb{N}^+)$—a set of edges including:

    (a) Transition edges—specifying the order in which the operation is performed. These edges are pairs $f_n = (o_{i,j}, o_{i,k})$, where the elements describe two successive operations $o_{i,j} \in O_P$;

    (b) Usage edges—indicating resources that change during the operation. These edges are pairs $f_n = (o_{i,j}, r_c)$, in which one element is operation $o_{i,j} \in O_P$, and the other is resource $r_c \in R_P$;

(c) Dependency edges—indicating operations depending on the current value of one of the resources. These edges are pairs $f_n = (r_c, o_{i,j})$, in which one element is the resource $r_c \in R_P$, and the other is the operation $o_{i,j} \in O_P$;

(d) Locking edges—indicating the operation applying the selected lock. These edges are pair $f_n = (q_s, o_{i,j})$, in which one element is the lock, and the other is the locking operation;

(e) Unlocking edge—indicating the operation releasing the selected lock. These edges are pairs $f_n = (o_{i,j}, q_s)$, in which one element is the unlocking operation, and the other is the released lock;

8. $B_P = (B_P^{FWD}, B_P^{BWD}, B_P^{SYM})$—sequence of sets:

- B_P^{FWD} a set of pairs of forward relationship operations:
  $B_P^{FWD} = \{(o_{i,j}, o_{a,b}); o_{i,j}, o_{a,b} \in O_P\}$;
- B_P^{BWD} a set of pairs of backward relationship operations:
  $B_P^{BWD} = \{(o_{i,j}, o_{a,b}); o_{i,j}, o_{a,b} \in O_P\}$;
- B_P^{SYM} family of pairs of operations related by a symmetrical relation:
  $B_P^{SYM} = \{\{o_{i,j}, o_{a,b}\}; o_{i,j}, o_{a,b} \in O_P\}$.

Introduced into the model set B_P in the graphical representation is as shown in Figure 2. If a pair of operations is in a **forward** relationship with each other, then the edge connecting the two operations has a filled ring at the operation after which the second operation is required. Similarly, in a **backward** relationship, the edge connects the two operations, with the empty circle being placed next to the operation that needs to be preceded by another operation. In the case of a **symmetrical** relation, the edge is the combination of the two above, and there are both a ring and a circle at its ends.

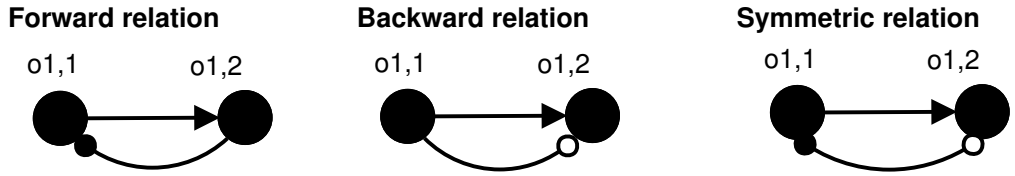

Figure 2. Edges reflecting the relationship between operations.

An example in the form of a pseudocode can be found in Figure 3. Assuming the function calls of *fgetpos*, the operations are $o_{1,2}$ and $o_{1,6}$ and assuming the function call of *strerror*, the operations are $o_{1,9}$ and $o_{1,11}$, and B_P^{BWD} will consist of the following pairs $(o_{1,2}, o_{1,9})$, $(o_{1,2}, o_{1,11})$, $(o_{1,6}, o_{1,9})$, $(o_{1,6}, o_{1,11})$. The graphical representation of the code from Figure 3 is shown in Figure 4.

```
1  if (condition1) {        // o1,1
2      fgetpos(...);        // o1,2
3      ...                  // o1,3
4  } else {                 // o1,4
5      ...                  // o1,5
6      fgetpos(...);        // o1,6
7  }
8  if (condition2) {        // o1,7
9      ...                  // o1,8
10     res = strerror();    // o1,9
11 } else {                 // o1,10
12     res = strerror();    // o1,11
13     ...                  // o1,12
14 }
```

Figure 3. Advanced example of relations.

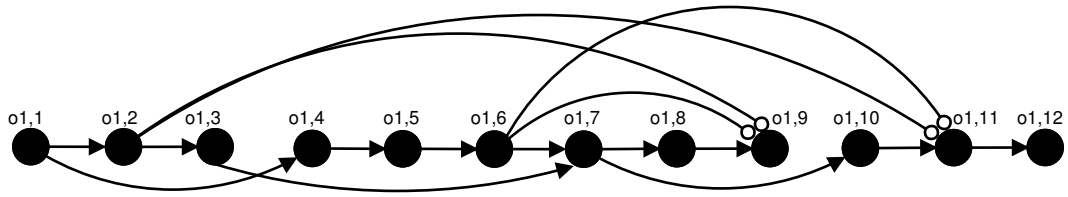

Figure 4. Subgraph representing the code from Figure 3.

5.1. Relation Breakage

The order of execution of an operation within one thread may cause the result of one operation to be overwritten by the result of another operation. As a result, a further operation in relation to both operations will always receive the result only from the one that occurs later. Such a situation is called a relation breakage and takes place in the code from the Figure 5, the graphical representation of this code is shown in Figure 6. According to the model from Section 5 and Table 2 the set of relations backwards is as follows:

$$B_P^{BWD} = \{(o_{1,1}, o_{1,12}), (o_{1,4}, o_{1,12}), (o_{1,6}, o_{1,12}), (o_{1,11}, o_{1,12})\} \tag{2}$$

Relationship between operations of pairs $(o_{1,1}, o_{1,12})$ it is always broken because in a series of subsequent operations there will always be other operations forming a relationship with the operation $o_{1,12}$. The relationship between operations $(o_{1,4}, o_{1,12})$ will also be broken. The other two pairs of operations $(o_{1,6}, o_{1,12})$ and $(o_{1,11}, o_{1,12})$ are placed in two mutually exclusive code blocks, so there will be no breakage, but each of them is the cause of breaking the previous two pairs.

```
1  fgetpos(...);              // o1,1
2  switch(x) {                // o1,2
3      case 1:                // o1,3
4          fgetpos(...);      // o1,4
5      case 'A':              // o1,5
6          fgetpos(...);      // o1,6
7          ...                // o1,7
8          break;             // o1,8
9      default:               // o1,9
10         ...                // o1,10
11         fsetpos(...)       // o1,11
12 }
13 char * res = strerror();// o1,12
```

Figure 5. Pseudocode showing the process of breaking relationships.

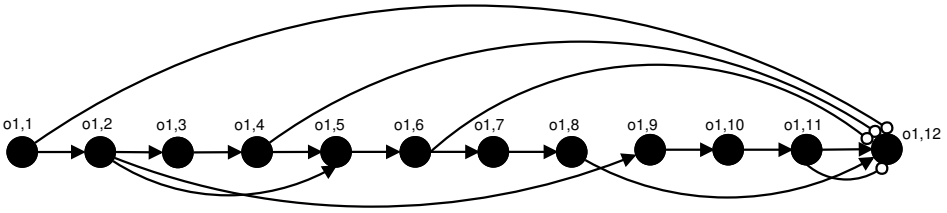

Figure 6. A subgraph representing the code from Figure 5.

Based on the previous paragraph and on the definition of the subgraph ${}_i^s G_P$ [2], the definition of a relationship breaking up is as follows.

Definition 2. *If there are pairs of operations in the i-th thread, $(o_{i,j}, o_{i,l})$ and $(o_{i,k}, o_{i,l})$ belong to any subset of the B_P sequence, and in each subgraph path there are ${}^{s}_{i}G_P$ operations $o_{i,j}$, $o_{i,k}$ and $o_{i,l}$, such that $j < k < l$ occur, then the relationship $(o_{i,j}, o_{i,l})$ is called completely broken.*

Definition 3. *If, in the i-th thread, there are pairs of operation $(o_{i,j}, o_{i,l})$ and $(o_{i,k}, o_{i,l})$ belonging to any subset of B_P sequence, and there is a path of subgraph ${}^{s}_{i}G_P$, in which between operations $o_{i,j}$ and $o_{i,l}$ the operation $o_{i,k}$ does not occur, then the relationship $(o_{i,j}, o_{i,l})$ is called conditionally broken.*

Broken relationships are a consequence of two operations, of which the second invalidates the result of the first operation—e.g., by overwriting the value of the shared resource. If a relationship is broken, it should not be present in the sets of B_P sequences. Therefore, if there are relationships that are not broken or are conditionally broken, they may be disrupted by an operation from another thread, leading to atomicity violation.

5.2. Reversal of Relation

The reversal of the relationship between operations of one is not considered for several reasons. In the case of the forward relationship, the second operation of the pair has the right to occur independently before the first operation. Likewise, in the reverse relationship, the first operation of the pair can occur independently after the second. In the case of symmetrical relation, reversing the relation will always result in an error, which will be revealed when such a code is first run.

There may also be a situation where two operations are linked by one of the relations and each of them is in a separate thread. A scenario discussing such a case may lead to the phenomenon of order violation, which is not the subject of this work.

6. Problem Definition

A multithreaded P application is given, written in C using the *pthread* library, in which pairs of operations are marked with each other in the relations described in Section 4. This application is affected by the phenomenon of atomicity violation. Is it possible to detect a conflict causing a violation of atomicity using the model from Section 5?

7. Sufficient Condition

A multithreaded application code is given as a model in which:

- There are no conflicts causing race condition and deadlock;
- In graph G_P [2], there is a pair of threads t_i and t_j working in the same range of u_b;
- In thread t_i, there is a pair of operations $(o_{i,\alpha}, o_{i,\beta}) \in B_P$ and these operations are in an agreement relationship which is not completely broken;
- In thread t_j, there is operation $\{o_{j,\gamma}\}$;
- In the set of operations $\{o_{i,\alpha}, o_{i,\beta}, o_{j,\gamma}\}$, at least one of them is connected to a resource with a shared use edge (one of the scenarios listed in Table 1 is fulfilled).

The phenomenon of atomicity violation occurs at the moment when each of the operations of pair $(o_{i,\alpha}, o_{i,\beta})$ is present in a different critical section.

Figure 7 shows the first scenario from Table 1 using a graphical representation of the source code model of a multithreaded application. All characteristics listed in the points above can be found there.

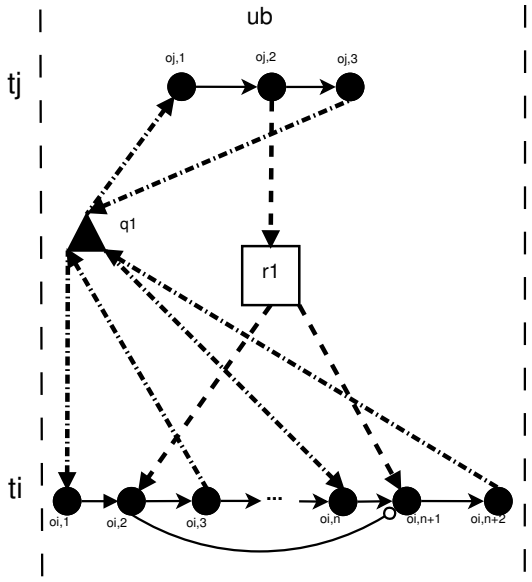

Figure 7. Subgraph showing the first scenario of atomicity violation.

So, assuming that there are two threads t_i i t_j and set of operations $\{o_{i,\alpha}, o_{i,\beta}, o_{j,\gamma}\}$ protected by q_s mutex and using resource r_c in such a way that they fulfil one of the scenarios set out in Table 1, it can be concluded that:

Lemma 1. *Operations $o_{i,\alpha}$ and $o_{i,\beta}$ are connected by one of the agreement relationship, which is not completely broken.*

If there is a cycle in G_P graph that consists of mutex q_s and only one operation from set $\{o_{i,\alpha}, o_{i,\beta}\}$ then the phenomenon of atomicity violation occurs.

Theorem 1. *The atomicity is violated if the above lemma is fulfilled.*

Proof. This is the consequence of Definition 1. If there is a cycle in G_P graph that consist of q_s and only one operation from the set of $\{o_{i,\alpha}, o_{i,\beta}\}$—that is, the operations $o_{i,\alpha}, o_{i,\beta}$ do not belong to the same critical section. This means that it is possible to execute an $o_{j,\epsilon}$ operation of another thread (t_γ) on the same resource between the $o_{i,\alpha}, o_{i,\beta}$ operations—i.e., order $o_{i,\alpha} \prec o_{j,\epsilon} \prec o_{i,\beta}$. It is then possible for atomicity violation to occur for the $o_{i,\alpha}, o_{i,\beta}$ operation by execution of the $o_{j,\epsilon}$ operations. □

8. Leading Example

The atomicity violation can be found in the sample AV1 application (source code: http://bit.ly/2P849ma). To do that there is only the need to use the *rdao_detector* (https://github.com/PKPhdDG/rdao_detector), application which uses MASCM to detect, among other things, the phenomenon of atomicity violation. The detection discussed in the previous section is implemented as an algorithm in the author's tool and is used to optimize code. The code used is equivalent to a piece of TCP/IP server application and was running on an embedded device. The server was run as a data collector in a commercial environment. Two running threads were responsible for handling requests from two different sensor groups. In the presented example there are many simplifications to show an idea that is significant for multithreaded applications. After converting the application to a model described in Section 5, it looks as follows.

$$T_{AV1} = (t_0, t_1, t_2);$$
$$U_{AV1} = (\{t_0\}, \{t_1, t_2\}, \{t_0\});$$
$$R_{AV1} = \{(r1)\};$$

$$O_{AV1} = \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}, o_{1,7}, o_{1,8}, o_{1,9}, o_{2,1}, o_{2,2}, o_{2,3}, o_{2,4}, o_{2,5}, o_{2,6}, o_{2,7}, o_{2,8}, o_{2,9}\};$$
$$Q_{AV1} = \{(m, PMD)\};$$
$$F_{AV1} = \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (o_{0,3}, r_1), (o_{0,3}, o_{0,4}), (o_{0,4}, r_1), (o_{0,4}, o_{0,5}), (o_{1,1}, o_{1,2}), (q_1, o_{1,2}), (o_{1,2}, o_{1,3}), (o_{1,3}, r_1), (o_{1,3}, o_{1,4}), (o_{1,4}, q_1), (o_{1,4}, o_{1,5}), (o_{1,5}, o_{1,6}), (q_1, o_{1,6}), (o_{1,6}, o_{1,7}), (r_1, o_{1,7}), (o_{1,7}, o_{1,8}), (o_{1,8}, q_1), (o_{1,8}, o_{1,1}), (o_{1,1}, o_{1,9}), (o_{1,8}, o_{1,9}), (o_{2,1}, o_{2,2}), (q_1, o_{2,2}), (o_{2,2}, o_{2,3}), (o_{2,3}, r_1), (o_{2,3}, o_{2,4}), (o_{2,4}, q_1), (o_{2,4}, o_{2,5}), (o_{2,5}, o_{2,6}), (q_1, o_{2,6}), (o_{2,6}, o_{2,7}), (r_1, o_{2,7}), (o_{2,7}, o_{2,8}), (o_{2,8}, q_1), (o_{2,8}, o_{2,1}), (o_{2,1}, o_{2,9}), (o_{2,8}, o_{2,9})\};$$
$$B^{BWD}_{AV1} = \{(o_{1,3}, o_{1,7}), (o_{2,3}, o_{2,7})\}.$$

In AV1, there is a backward relationship between the *printf* function and the incrementation operation. As a result of its presence, the user is likely to receive incorrect data on the standard output.

Locating the atomicity violation phenomenon consists of several steps, after making sure that the application is free of resource conflicts causing race condition or deadlock. In the first step, it is necessary to determine whether there are intervals in which more than one thread works, which is true for AV1 applications. Then, as a second step, one should determine which of the pairs belonging to the sets of sequences of B_{AV1} can be violated by changing the shared resource. For AV1 applications these are pairs:

1. $(o_{1,3}, o_{1,7})$ can be violated by $o_{2,3}$;
2. $(o_{2,3}, o_{2,7})$ can be violated by $o_{1,3}$.

In step three, it should be determined whether the pairs coming from the sets of B_{AV1} sequences meet one of the scenarios in Table 1. If the pair meets any scenario, it should be determined whether there is a path in the operation graph that meets one of the lemmas described in Section 7. In the case of AV1 applications, there are two such paths that meet Lemma 1 and both of them start on a mutex q_1. As a result of the fulfilment of Lemma 1, AV1 can therefore be defined as an incorrect application—i.e., one in which there are resource conflicts causing the phenomenon of atomicity violation.

On the basis of the leading example, it can therefore be concluded that the use of a model to monitor the source code of a multithreaded application allows for the detection of the atomicity violation phenomenon. The condition for detecting conflicts causing this phenomenon is to determine which functions and instructions of the source code are with each other in one of the three developed relationships. The described example of atomicity violation is another phenomenon that can be detected with the proposed model of multithreaded applications. The work on the model [2] has shown that the proposed approach is 2350 times faster in operation than alternatives based on a complete review. Due to the fact that the proposed solution is based on static code analysis, it can be assumed that a similar result will be obtained in case of detecting atomicity violations. For example, locating phenomenon of atomicity violation in AV1 application using *rdao_detector* took 0.84 s which is not so long when compared with the compilation time which was 0.52 s. Parts of both processes have common steps, such as converting source code into abstract syntax trees. If this process was common and the result could be shared, the time of compilation and validation of source code together could take approximately double the compilation time.

9. Conclusions

In order to locate the conflicts causing the phenomenon of race condition, the developed source code model of multithreaded applications had to be extended with a new relationship that may occur between the two operations. The analysis of the literature showed that in the context of the phenomenon of race condition, the developed relations were not considered anywhere. Knowledge of this relationships may be used to improve programming language grammar or to create an extension for languages. Programmers aware of the existence of relationships have the possibility to create better architectures of applications and better unit tests. It is also possible that the developed relations may be

used for research on various types of errors not only in multithreaded applications. Another interesting idea is use MASCM and developing relationships between memory allocation operations and memory deallocation operations to search for memory leaks in C applications.

MASCM can be used also in mixed methods—e.g., for searching atomic regions. Additonally, code converted to instances of MASCM can be used with neural networks for intelligent methods of detecting race condition, atomicity violation, etc. To avoid false-positive errors there is need more researches on the atomicity violation and static analysis methods.

The developed method based on a sufficient condition enabled us to locate two resource conflicts causing the phenomenon of race condition in the leading example, which improves code optimization in, for example, applications with parallel computing. Further research on the relations should also make it possible to predict the conflicts causing the phenomenon of order violation.

During studies on atomicity violation, any false-positive errors do not occur, but there is no proof that developed rules of detecting do not allow that. It is possible to find part of code which in instance of MASCM can meet atomicity violation requirements, as it is in most other methods using static analysis of code.

Due to the fact that 70% of all multithreaded errors [7] are errors that are a violation of atomicity, the development of a method that allows for their rapid detection only through static analysis of the source code may turn out to be a significant contribution to the field of computer science.

Compilers that allow for detecting these errors during compilation process may permit us to avoid atomicity violation even before testing process. It means that programmers will be able to be more efficient, their programs will be less unreliable and the process of testing multithreaded applications will be less complicated because fewer cases will need to be checked.

Author Contributions: All authors contributed to the research work. D.G. proposed the main idea. D.G. and R.W. conceived the multithreaded application model and wrote the article together. G.D. implemented software tools and made experiments. R.W. reviewed the research work. All authors worked together and responded to the reviewers comments. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AII	Access Interleaving Invariants
AST	Abstract Syntax Trees
AV1	Example application name
BWD	Backward
FWD	Forward
HAVE	Hybrid Atomicity Violation Explorer
JDK	Java Development Kit
MASCM	Multithreaded Application Source Code Model
PMD	Pthreads Mutex Default
PME	Pthreads Mutex Error Check
PMN	Pthreads Mutex Normal
PMR	Pthreads Mutex Recursive
SYM	Symmetric

References

1. Giebas, D.; Wojszczyk, R. Multithreaded Application Model. In *Distributed Computing and Artificial Intelligence, 16th International Conference, Special Sessions*; Springer: Cham, Germany, 2019; Volume 1004, pp. 93–103.
2. Giebas, D.; Wojszczyk, R. Deadlocks Detection in Multithreaded Applications Based on Source Code Analysis. *Appl. Sci.* **2020**, *10*, 532. [CrossRef]

3. Woo, J.; Choi, H.; Lee, J. Empirical Performance Analysis of Collective Communication for Distributed Deep Learning in a Many-Core CPU Environment. *Appl. Sci.* **2020**, *10*, 6717. [CrossRef]
4. Rudy, J.; Bożejko, W. Reversed Amdahl's Law for Hybrid Parallel Computing. In *Computer Aided Systems Theory—EUROCAST 2017*; Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 101–108.
5. Oh, Y.R.; Park, K.; Park, J.G. Online Speech Recognition Using Multichannel Parallel Acoustic Score Computation and Deep Neural Network (DNN)- Based Voice-Activity Detector. *Appl. Sci.* **2020**, *10*, 4091. [CrossRef]
6. Jin, G.; Song, L.; Zhang, W.; Lu, S.; Liblit, B. Automated Atomicity-Violation Fixing. *ACM SIGPLAN Not.* **2011**, *46*, 389–400. [CrossRef]
7. Park, S.; Lu, S.; Zhou, Y. *CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places*; ACM SIGARCH Computer Architecture News; ACM: New York, NY, USA, 2009; Volume 37, pp. 25–36.
8. Park, C.; Sen, K. Randomized active atomicity violation detection in concurrent programs. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, GA, USA, 9–14 November 2008; pp. 135–145.
9. Flanagan, C.; Freund, S.N. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. *ACM SIGPLAN Not.* **2004**, *39*, 256–267. [CrossRef]
10. Rak, T. Performance Modeling Using Queueing Petri Nets. *Commun. Comput. Inf. Sci.* **2017**, *718*, 321–335.
11. Von Praun, C.; Gross, T. Static Detection of Atomicity Violations in Object-Oriented Programs. *J. Object Technol.* **2004**, *3*, 103–122. [CrossRef]
12. Zhou, Y.; Lu, S.; Tucek, J.A. Atomicity Violation Detection Using Access Interleaving Invariants. U.S. Patent 8,533,681, 10 September 2013
13. Dias, R.J.; Pessanha, V.; Lourenço, J.M. Precise detection of atomicity violations. In Proceedings of the Haifa Verification Conference, Haifa, Israel, 6–8 November 2012; pp. 8–23.
14. Ceze, L. Detecting and Avoiding Atomicity Violations. Available online: https://www.nii.ac.jp/userimg/lectures/LuisCeze/nii-lecture4.pdf (accessed on 9 October 2019).
15. Stroustrup, B. *The C++ Programming Language*, 4th ed.; Addison-Wesley: Boston, MA, USA, 2013.
16. Chew, L.; Lie, D. Kivati: Fast detection and prevention of atomicity violations. In Proceedings of the 5th European Conference on Computer Systems, Paris, France, 13–15 April 2010; pp. 307–320.
17. Yang, Y.; Gringauze, A.; Wu, D.; Rohde, H.K. Detecting Data Race and Atomicity Violation via Typestate-Guided Static Analysis. U.S. Patent 8,510,722, 13 August 2013
18. Lu, S.; Tucek, J.; Qin, F.; Zhou, Y. AVIO: Detecting atomicity violations via access interleaving invariants. *ACM SIGOPS Oper. Syst. Rev.* **2006**, *40*, 37–48. [CrossRef]
19. Xu, M.; Bodík, R.; Hill, M.D. A serializability violation detector for shared-memory server programs. *ACM SIGPLAN Not.* **2005**, *40*, 1–14.
20. Tongping, L.; Zhou, J.; Silvestro, S.; Liu, H. Defeating Deadlocks in Production Software. U.S. Patent Application No. 16/159,234, 18 April 2019.
21. Sasturkar, A.; Agarwal, R.; Wang, L.; Stoller, S.D. Automated type-based analysis of data races and atomicity. In Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Chicago, IL, USA, 15–17 June 2005; pp. 83–94.
22. Wang, C.; Limaye, R.; Ganai, M.; Gupta, A. Trace-based symbolic analysis for atomicity violations. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Paphos, Cyprus, 20–28 March 2010; pp. 328–342.
23. TIOBE Index. Available online: https://www.tiobe.com/tiobe-index/ (accessed on 12 October 2019).
24. Chang, X.; Dou, W.; Gao, Y.; Wang, J.; Wei, J.; Huang, T. Detecting atomicity violations for event-driven Node.js applications. In Proceedings of the 41st International Conference on Software Engineering, Montreal, QC, Canada, 25–31 May 2019; pp. 631–642.
25. Yu, Z.; Song, L.; Zhang, Y. Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software. *arXiv* **2019**, arXiv:1902.01906.
26. Roberson, M.; Boyapati, C. A Static Analysis for Automatic Detection of Atomicity Violations in Java Programs. 2010. Available online: https://www.eecs.umich.edu/techreports/cse/2011/CSE-TR-569-11.pdf (accessed on 11 November 2020).

27. Chen, Q.; Wang, L.; Yang, Z.; Stoller, S.D. HAVE: Detecting atomicity violations via integrated dynamic and static analysis. In Proceedings of the International Conference on Fundamental Approaches to Software Engineering, York, UK, 22–29 March 2009; pp. 425–439.
28. Huss, E. The C Library Reference Guide. Available online: http://www.fortran-2000.com/ArnaudRecipes/Cstd/ (accessed on 11 November 2020).