Received July 15, 2019, accepted August 10, 2019, date of publication August 15, 2019, date of current version August 28, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2935498

OpenACC Errors Classification and Static Detection Techniques

AHMED MOHAMMED ALGHAMDI AND FATHY ELBOURAEY EASSA
Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia

Corresponding author: Ahmed Mohammed Alghamdi (aalghamdi2864@stu.kau.edu.sa)

This work was supported by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under Grant DG-015-611-1440.

ABSTRACT With the continued increase of usage of High-Performance Computing (HPC) in scientific fields, the need for programming models in a heterogeneous architecture with less programming effort has become important in scientific applications. OpenACC is a high-level parallel programming model used with FORTRAN, C, and C++ programming languages to accelerate the programmers' code with fewer changes and less effort, which reduces programmer workloads and makes it easier to use and learn. Also, OpenACC has been increasingly used in many top supercomputers around the world, and three of the top five HPC applications in Intersect360 Research are currently using OpenACC. However, when programmers use OpenACC to parallelize their code without correctly understanding OpenACC directives and their usage or following OpenACC instructions, they can cause run-time errors that vary from causing wrong results, performance issues, and other undefined behaviors. In addition, building parallel systems by using a higher level programming model increase the possibility to introduce errors, and the parallel applications thus have non-determined behavior, which makes testing and detecting their run-time errors a challenging task. Although there are many testing tools that detect run-time errors, this is still inadequate for detecting errors that occur in applications implemented in high-level parallel programming models, especially OpenACC related applications. As a result, OpenACC errors that cannot be detected by compilers should be identified, and their causes should be explained. In this paper, our contribution is introducing new static techniques for detecting OpenACC errors, as well as for the first time classifying errors that can occur in OpenACC software programs. Finally, to the best of our knowledge, there is no published work to date that identifies or classifies OpenACC-related errors, nor is there a testing tool designed to test OpenACC applications and detect their run-time errors.

INDEX TERMS OpenACC, OpenACC run-time errors, OpenACC error classifications, OpenACC testing tool, static approach for OpenACC application.

I. INTRODUCTION

Over the past few years, OpenACC has become increasingly used in many high-performance computers, including the top supercomputer at the World Summit. Also, OpenACC attracts more non-computer science specialists for accelerating their systems in several scientific fields, including weather forecasting and simulations. OpenACC is a high-level parallel programming model designed for heterogeneous systems, first released in 2011. This programming model is used for supporting parallelism in sequential programming languages by adding OpenACC directives without low-level details and is easy to learn and use. Therefore, programmers could cause some errors when using OpenACC to parallelize their code without fully understanding OpenACC instructions.

Testing parallel applications built by using programming models is a difficult task because if badly programmed they can have a non-determined behavior, which makes it challenging to detect their errors when they occur and determine the causes of these errors, whether from the user source code or the programming model directives. Also, it is difficult to see if the errors have been corrected or are still present but hidden, even when these errors have been detected and the source code modified.

Many studies have investigated several programming models-related applications for detecting and identifying

The associate editor coordinating the review of this article and approving it for publication was Tao Zhang.

run-time errors, as well as other semantic and syntax errors. However, OpenACC has not been investigated or identified clearly, as the other programming models have. Although there are many testing tools that detect run-time errors, this is still not enough to detect errors that occur in applications implemented in high-level parallel programming models used for a heterogeneous system architecture, especially OpenACC-related applications.

In this paper, our contribution is introducing new static techniques for detecting OpenACC errors, and for the first time classifying errors that can occur in OpenACC software programs. We briefly mention some OpenACC run-time errors in our previous study, which was published in [1], but in this paper, we broadly cover OpenACC run-time errors and explain their causes with examples. Part of our study focuses on OpenACC errors that cannot be detected by compilers. Our experiments have been conducted using C++ applications with OpenACC directives, and we used the PGI 19.4 compiler (community edition) to conduct our experiments and compile our applications. Also, we propose a solution for detecting these errors by building a static testing tool for detecting run-time errors in OpenACC applications. Finally, to the best of our knowledge, we are the first to identify and classify OpenACC run-time errors, as well as building a static testing tool designed to detect these errors in OpenACC applications.

The rest of this paper is structured as follows. Sections 2 and 3 will discuss related work and briefly give an overview of OpenACC. Our classification of OpenACC run-time errors will be discussed in Section 4. In Section 5, we will explain our testing tool architecture design, and in Section 6 our static approach in detecting OpenACC errors will be explained in detail. In Section 7, we will discuss some aspects of our study, and our tool will be evaluated in Section 8. Finally, conclusions and future work will be discussed in Section 9.

II. RELATED WORK

Building massively parallel applications is challenging and has several difficulties that can affect the system's efficiency and accuracy. One of these difficulties is that parallel applications if badly programmed they can have non-determined behavior, which makes it hard to detect parallel errors or test parallel applications. Also, run-time errors vary from one programming model to another, and it is not easy to determine whether errors have been corrected or are hidden when modifying the source code.

There are many studies that have investigated parallel applications errors and identified them and their causes. Also, many programming models-related errors have been identified, including MPI, OpenMP, CUDA and OpenCL. However, OpenACC has not been investigated or identified as thoroughly as the other programming models.

OpenMP application common errors have been identified and classified in [2], where OpenMP errors were divided into defects and failures with explanations and some examples. Also, the published survey in [3] presented 15 different mistakes and how to avoid them by recommending best practices in these cases. In terms of MPI errors, the communication deadlock was investigated and identified in [4], while in [5] the authors introduced a classification for summarizing the error types that apply to MPI's non-blocking collectives. Finally, CUDA run-time errors were identified and ways to avoid these errors were published in [6], in which CUDA issues were classified into three categories: errors in using CUDA directives, general parallel errors, and algorithmic errors.

In terms of testing parallel applications, our previous work published in [7] found that several programming model-related applications have been investigated to identify their run-time errors, and many testing tools have been designed to detect these errors. Our study covered more than 50 testing tools and several run-time errors, including deadlock, race condition, livelock, and data mismatching. These testing tools are varied in their purposes and scopes, including testing tools that detect specific type of errors, the targeted programming models used, and the testing techniques.

There are many tools that have been created to detect a specific type of run-time error, including data race detection [8] and deadlock detection such as UNDEAD [9]. Also, many testing tools have been designed to test a specific targeted programming model such as testing tools for MPI [10], [11], OpenMP [12], [13], CUDA [14] and OpenCL [15]. In terms of OpenACC testing, there is no testing tool dedicated to testing OpenACC applications or detecting their related run-time errors. However, there are some studies that relate to compilers' evaluation published in [16], in which test cases have been created for evaluating OpenACC 2.0. Also, another study that evaluated CAPS, PGI, and CRAY compilers [17] and OpenACC 2.5 was evaluated in [18] for validating and verifying the compiler implementation of OpenACC's new features.

Despite efforts made to test parallel applications, there is still more work to be done with respect to high-level programming models used in heterogeneous systems. We have noticed that OpenACC has several advantages and benefits and has been used widely in the past few years, but its errors have not been investigated or identified as clearly as the other programming models or targeted by any testing tool. Finally, to the best of our knowledge, there is no parallel testing tool to test applications programmed by OpenACC or designated to test OpenACC applications and detect their run-time errors.

III. OVERVIEW OF OPENACC

OpenACC is a high-level parallel programming model designed for heterogeneous systems that is used to add parallelism to the FORTRAN, C, and C++ programming languages. In November 2011, OpenACC was released at the International Conference for High-Performance Computing, Networking, and Storage. OpenACC stands for open accelerators, which was developed by Cray, CAPS, NVIDIA, and PGI, and the latest version was released in November 2018. In addition, OpenACC has been used widely in the past few

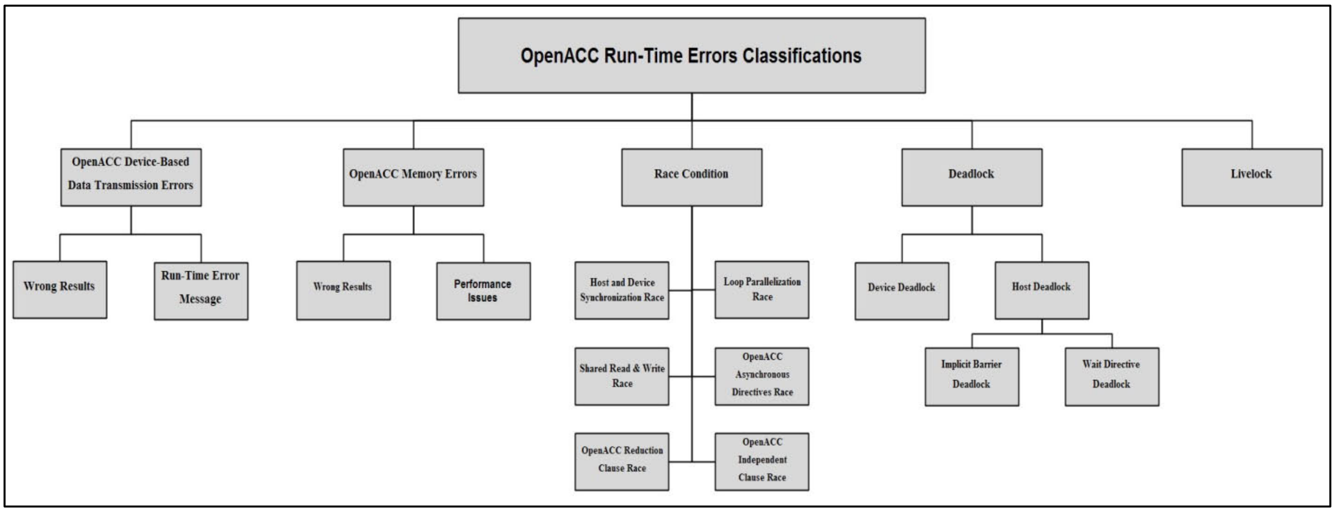

OpenACC Run-Time Errors Classifications
OpenACC Device-Based Data Transmission Errors
OpenACC Memory Errors
Race Condition
Deadlock
Livelock
Wrong Results
Run-Time Error Message
Wrong Results
Performance Issues
Host and Device Synchronization Race
Loop Parallelization Race
Shared Read & Write Race
OpenACC Asynchronous Directives Race
OpenACC Reduction Clause Race
OpenACC Independent Clause Race
Device Deadlock
Host Deadlock
Implicit Barrier Deadlock
Wait Directive Deadlock

FIGURE 1. **OpenACC error classifications.**

years by non-computer science specialists due to decreased programming efforts needed to accelerate their original code, as well as the simplicity of learning and using OpenACC. Also, OpenACC has been increasingly used in many top supercomputers around the world and in five out of 13 applications in the Summit Supercomputer, which is the top supercomputer in the world, as published in [19].

OpenACC has several directives and clauses used to accelerate source code without many changes. OpenACC directives have been divided into the data region, which is responsible for the data movements between host and device, and the compute region, which is used for executing the code on the device [20]–[22]. OpenACC's data region is defined by data directives that determine data lifetime on the device and is divided into structured and unstructured data regions, both used for data movements but in different aspects and behaviors. OpenACC can have multiple data and compute regions within the same source code.

OpenACC has several advantages and features that give it the ability to parallelize the code, including portability. Unlike other programming models like CUDA, which only work on NVIDIA, OpenACC is portable across platforms and different types of GPU [23], [24]. Also, OpenACC can work with various compilers and requires less programming effort, which gives it the ability to add parallelism to existing code with less code, decreasing programmers' workloads and improving their productivity. Finally, OpenACC supports three levels of parallelism by using three OpenACC clauses, including Gang, Worker, and Vector, which are coarse, medium, and fine-grained parallelism, respectively.

IV. OPENACC ERROR CLASSIFICIATIONS

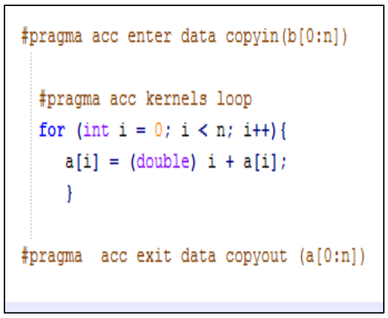
After analyzing the recent OpenACC version 2.7 documents [20], consulting OpenACC-related books, and conducting several experiments, we classify and identify several run-time errors based on our experiments and build several programs to simulate these errors to discover their behavior and effects on OpenACC-related applications, as shown in Figure 1. Programmers can cause these errors when they try to parallelize their applications by using OpenACC. Also, there are some directions and instructions that OpenACC documents indicated to avoid some errors, and when programmers do not follow these instructions, that can cause errors. In the following, we classify OpenACC run-time errors that cannot be detected by the compilers and explain each class and discuss their causes.

A. OPENACC DEVICE-BASED DATA TRANSMISSION ERRORS

Data management is one of the features supported by OpenACC, which uses data clauses to conduct data movement between CPU and GPU and vice versa. The programmers must be aware of the usage of each data clause and how to use them in both structured and unstructured data regions. In this classification, we identify run-time errors resulting from the mishandling of OpenACC data clauses and directives, which leads in turn to non-deterministic behaviors or wrong results. OpenACC data regions are divided into structured and unstructured data regions, as we explained in Section 3. The following identifies and explains OpenACC data clause-related errors that cannot be detected by the compiler.

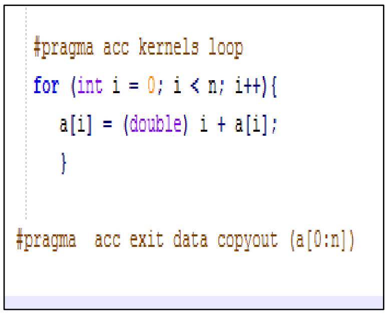
1) ERRORS LEAD TO RUN-TIME ERROR MESSAGES AFTER EXECUTING THE PROGRAM

Sometimes when programmers are misusing OpenACC data clauses, the compiler will not detect the presence of errors, but after compilation and during runtime, an error message will be issued indicating the invalid value without giving extra information about the error type or the cause of this error. Several scenarios explain this type of error and its causes. This error can occur in both structured and unstructured OpenACC data regions.



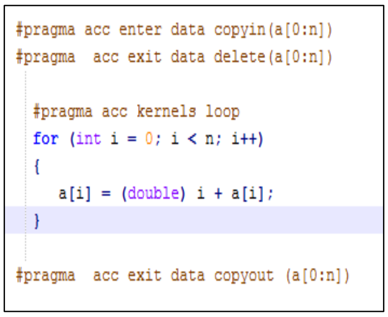
```
#pragma acc enter data copyin(b[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++){
     a[i] = (double) i + a[i];
     }

#pragma  acc exit data copyout (a[0:n])
```

(a) Different copyin and copyout variables



```
 #pragma acc kernels loop
 for (int i = 0; i < n; i++){
    a[i] = (double) i + a[i];
    }

#pragma  acc exit data copyout (a[0:n])
```

(b) Forget writing the enter data region variables



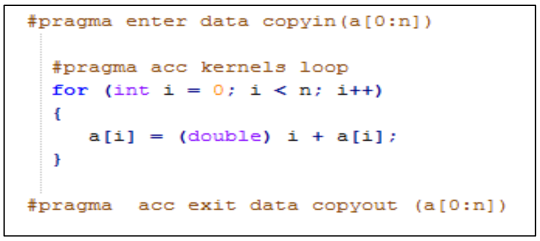
```
#pragma acc enter data copyin(a[0:n])
#pragma  acc exit data delete(a[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++)
  {
     a[i] = (double) i + a[i];
  }

#pragma  acc exit data copyout (a[0:n])
```

(c) Delete the array before the copyout

FIGURE 2. Errors leading to run-time error messages in the unstructured data region.

In the unstructured OpenACC data region, several scenarios can cause this type of error. One scenario arises when there is a variable in the copyout clause in the exit data region without having the same variable in any data clause in the enter data region, which will cause an error at the runtime without being detected by the compiler. As shown in Figure 2a, the array "a" at the copyout clause will cause this error because you simply ask the GPU to copy the array "a" to the CPU, but this variable is not present in the GPU, which will then result in an error message during runtime of "invalid value". Also, another case is when programmers write the data clause in the exit data region without writing the enter data region, as shown in Figure 2b.

The third situation that causes this error is deleting the variable before using the same variable in the copyout data clause, as in Figure 2c. Finally, there is a syntax error that can also lead to this type of error and is not detected by the compiler when the programmers write the enter data region directive without the "acc", as shown in Figure 3. In this case, the same array "a" is in both the enter and exit data regions,



```
#pragma enter data copyin(a[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++)
  {
     a[i] = (double) i + a[i];
  }

#pragma  acc exit data copyout (a[0:n])
```

FIGURE 3. Syntax error causing a run-time error in the unstructured data region.

but the compiler will ignore the enter data region directive, which affects the application in the same way as in Figure 2b.

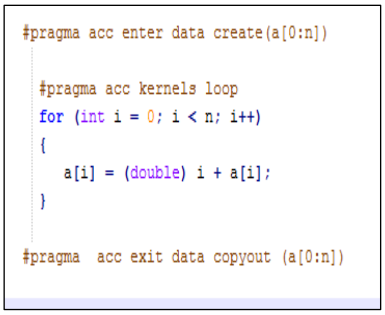
In the structured OpenACC data region, when programmers use OpenACC data clauses to manage the data movement between GPU and CPU or vice versa, and the array is not present in the GPU or partially present for any reason, this will be not detected by the compiler, and the error message will be issued during runtime. One reason for this situation is multiple routines controlling the data movements in the same application; therefore, the programmers should be aware of tracing the variables' movements both from and to the GPU. This error can be avoided by using the OpenACC API function to test for the presence of the variable in the GPU before further execution and preventing this error from occurring.

2) ERRORS LEADING TO WRONG RESULTS WITHOUT PROGRAMMERS' AWARENESS

Programmers using OpenACC data clauses without paying attention to their characteristics and features could lead to wrong results and cause the application to fail to meet the final user requirements. Also, programmers and compilers might not be aware that there is an error and therefore cannot detect the error without using the testing tool, which currently is unavailable, as we discussed in our survey published in [7]. Several scenarios cause this type of error, including structured and unstructured OpenACC data regions.

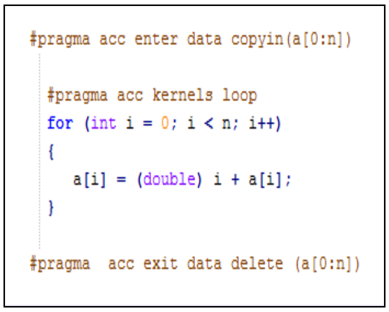
In unstructured OpenACC data regions, if the programmers want to copy data from CPU to GPU but mistakenly use the data clause create instead of copy or copyin, this will lead to wrong results without the programmers' or compilers' awareness. Figure 4a shows an example of this error, in which the programmers' use creates the array "a", which will create a space in the GPU without considering the previous values of the array "a" in the CPU, which will in turn eventually lead to wrong results. Another case of this error arising is when the programmers mistakenly use the delete clause at the exit data region when the variable needs to be used in the CPU, as shown in Figure 4b, which causes the variable to not be copied from the GPU to the CPU, therefore leading to wrong results. Similarly, when the exit data region directive has not been written or the "acc" keyword is forgotten, this also will lead to wrong results as shown in the codes in Figures 4c and 4d, respectively.

In the structured OpenACC data region, there are different causes of this type of error, including using the wrong



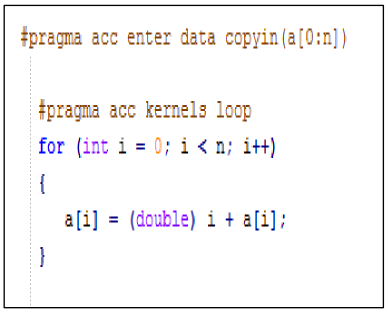
```
#pragma acc enter data create(a[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++)
  {
     a[i] = (double) i + a[i];
  }

#pragma  acc exit data copyout (a[0:n])
```

(a) Using create clause instead of copy or copyin clause



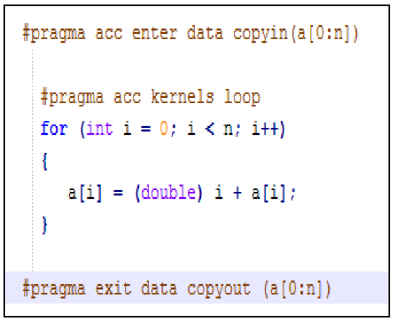
```
#pragma acc enter data copyin(a[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++)
  {
     a[i] = (double) i + a[i];
  }

#pragma  acc exit data delete (a[0:n])
```

(b) Delete the array without copyout



```
#pragma acc enter data copyin(a[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++)
  {
     a[i] = (double) i + a[i];
  }
```

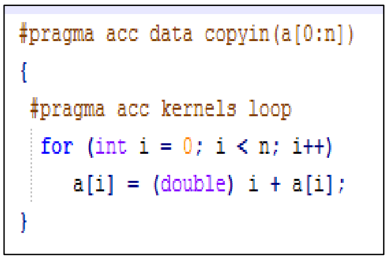
(c) Forget the exit data region directive



```
#pragma acc enter data copyin(a[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++)
  {
     a[i] = (double) i + a[i];
  }

#pragma exit data copyout (a[0:n])
```

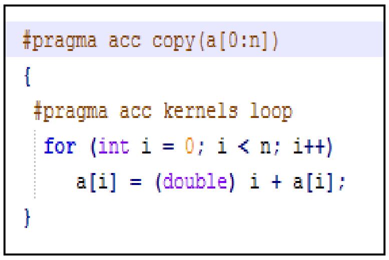
(d) Forget "acc" keyword at the exit data region directive

FIGURE 4. Errors leading to wrong results in the unstructured data region.



```
#pragma acc data copyin(a[0:n])
{
 #pragma acc kernels loop
  for (int i = 0; i < n; i++)
     a[i] = (double) i + a[i];
}
```

(a) Using copyin instead of copy



```
#pragma acc copy(a[0:n])
{
 #pragma acc kernels loop
  for (int i = 0; i < n; i++)
     a[i] = (double) i + a[i];
}
```

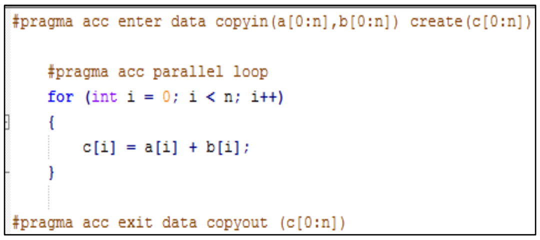
(b) Missing the keyword "data"

FIGURE 5. Errors leading to wrong results in the structured data region.

data clause or forgetting to write the OpenACC data directives correctly, which are considered syntax errors, but the compilers do not detect them. For example, in Figure 5a, the programmers use the copyin data clause instead of the copy data clause, while the array "a" needs to be copied from the CPU to the GPU and then copied back to the CPU. In this situation, the final result will be incorrect because by using copyin, the array will be copied to the GPU and stay there without being copied back to the CPU. Also, if the keyword "data" has not been written in the structured data region directive, as in Figure 5b, any data clause will not be activated because the compiler simply ignores the data region directive and completes the compilation, which results in incorrect values. However, when the keyword "acc" is missing, the PGI compiler 18.4 will generate implicit copy for the array "a" that only occurred in the structured data region, but in the unstructured data region the compiler will do nothing, as we see in the previous examples.

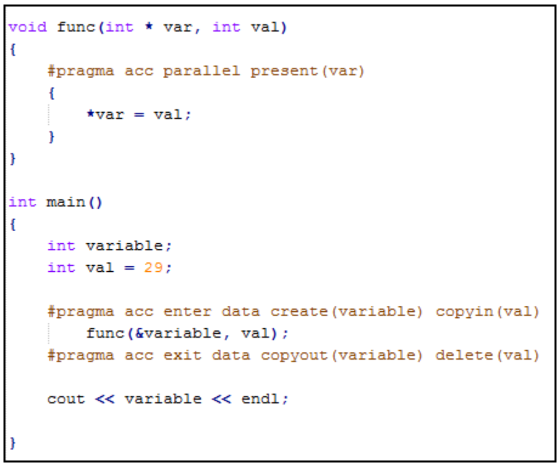
B. OPENACC MEMORY ERRORS

In this classification, this type of error will result in wrong results in some cases, similar to the previous classification, but this error will also affect application performance and GPU memory. The main concept behind this classification is that we identify cases of some unnecessary programming operations such as keeping unused variables and matrices in the GPU. Also, they create temporary arrays in the GPU for some operations and keep them unnecessarily after finishing operations, which can affect system performance by allocating unnecessary data to GPU memory and consuming space and energy. In Figure 6, the arrays "a" and "b" have been copied to the GPU for calculating the array "c", but after this operation has been completed, these two arrays still consume GPU memory without any further usage. In this case, the programmers should delete these two arrays per the exit data region directive. Also, further errors can result from using the same GPU memory with another part of the code, which conflicts with GPU memory or slows down the execution of the other operation. This type of error only happens in the unstructured data region because the programmers' responsibility is to use the enter and exit data regions directives correctly and to determine allocation and deallocation operations based on the data clause directives.



```
#pragma acc enter data copyin(a[0:n],b[0:n]) create(c[0:n])

    #pragma acc parallel loop
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }

#pragma acc exit data copyout (c[0:n])
```

FIGURE 6. Memory error causing memory consumption for unnecessary variables.



```
void func(int * var, int val)
{
    #pragma acc parallel present(var)
    {
        *var = val;
    }
}

int main()
{
    int variable;
    int val = 29;

    #pragma acc enter data create(variable) copyin(val)
        func(&variable, val);
    #pragma acc exit data copyout(variable) delete(val)

    cout << variable << endl;

}
```

FIGURE 7. Calling by value for data clause variable causing an error.

In the structured data region, the programmers determine only the data clause that they need, and the compiler deals with internal operations. When exiting the data region, the exit data directive handles GPU memory deallocation by using either the delete or copyout data clauses. Finally, one of the most important issues that some programmers might not be aware of is if they have as many exit data directives for a given array as enter data directives.

Another cause of this error is programmers' use of OpenACC data clause variables designated by functions; this designation should be determined by reference, not by value. In Figure 7, the variable ''val'' in the function ''func'' is not the same as the variable ''val'' in the copyin data clause in the main program. When the programmers call passes ''val'' by value, this means it will make a copy of ''val'' and send it to the function, which will then add it to the present data clause. Therefore, an error message will be issued to indicate that the variable ''val'' is not present in the GPU. The solution to this error is to pass the variable ''val'' by reference, which lets the compiler know that the variable ''val'' is on the GPU, and the function ''func'' can then use it directly.

C. RACE CONDITION

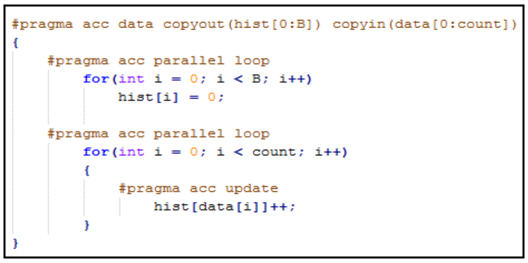
The race condition is a common error in parallel applications, as well as in different programming models. However, the OpenACC race condition behaves differently and has different causes as a result of the nature of OpenACC and its execution in a heterogeneous system architecture. The race condition can arise from executing processes concurrently in multiple threads without considering the sequence of the execution, while the thread execution sequence is critical to the final result. Also, a race condition can occur when there is competition between several threads to access the same memory location. Additionally, when programmers use OpenACC for parallelizing their applications, there is no guarantee of the thread execution order [22]. As a result, the programmers are responsible for examining their code to make sure there is no data dependency, and they should not make any assumptions about the thread order execution. Therefore, OpenACC is more likely to have a race condition resulting from different causes and situations. We classify the OpenACC race condition causes into the following six categories:

1) HOST AND DEVICE SYNCHRONIZATION RACE

In some situations, the synchronization between the CPU (host) and the GPU (device) is important to maintain data coherence. Therefore, data updating operations between host and device and vice versa should be done carefully, and programmers must determine when and how to update their data; otherwise, this can cause a race condition. The code in Figure 8 is an example of a race condition that occurs because of CPU/GPU synchronization. The elements in the ''hist'' array might be updated by multiple threads concurrently, which causes a data race. The OpenACC update directive updates the values of the ''hist'' array between GPU and CPU in the same data region without ensuring data independence between each element because of the OpenACC parallel directive, which depends on the programmers to ensure data independence for each element in the array. Finally, when programmers use ''kernels'' to parallelize their code, they will avoid the data race, but they will lose loop parallelism because the code will execute sequentially, as the usage of ''kernels'' relies on the compiler to determine dependency in this case.

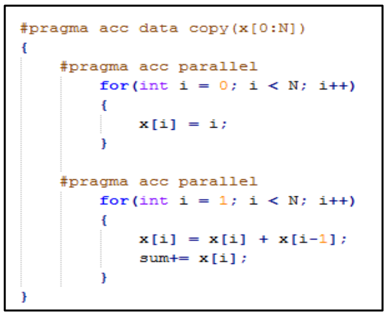
2) LOOP PARALLELIZATION RACE

When programmers want to parallelize their loops, they can use OpenACC directives to enable the loop body to be executed in parallel using concurrent hardware execution threads. The loop iterations might run in parallel at the same time without considering the iteration order. Moreover, the last iteration can be executed and completed before the first iteration, which will lead to a potential race condition if the execution order is important to the application. Also, the data dependency between iterations causes a race condition. As a result, based on the application analysis and requirements, the programmers should parallelize their loops carefully to ensure there is no dependency in their loop body because if so, that will lead to a race condition, causing the application to fail to meet the user requirements. Figure 9a shows an example of a race condition resulting from data dependency. In this example, some x[i] may be read or written



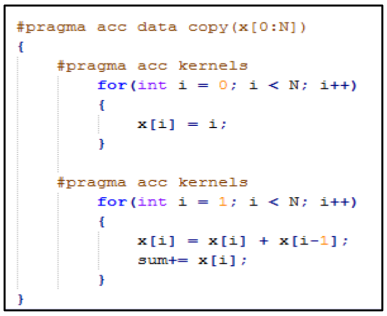
```
#pragma acc data copyout(hist[0:B]) copyin(data[0:count])
{
    #pragma acc parallel loop
        for(int i = 0; i < B; i++)
            hist[i] = 0;

    #pragma acc parallel loop
        for(int i = 0; i < count; i++)
        {
            #pragma acc update
                hist[data[i]]++;
        }
}
```

FIGURE 8. **Race condition because of synchronization between host and device.**



```
#pragma acc data copy(x[0:N])
{
    #pragma acc parallel
        for(int i = 0; i < N; i++)
        {
            x[i] = i;
        }

    #pragma acc parallel
        for(int i = 1; i < N; i++)
        {
            x[i] = x[i] + x[i-1];
            sum+= x[i];
        }
}
```

(a) Data dependency race using parallel directive



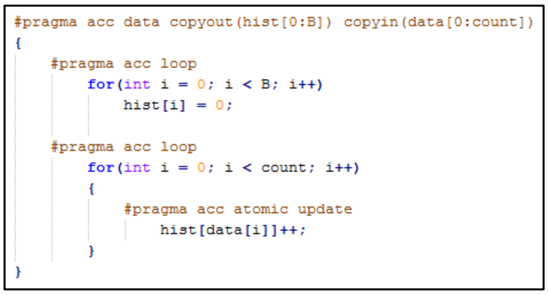
```
#pragma acc data copy(x[0:N])
{
    #pragma acc kernels
        for(int i = 0; i < N; i++)
        {
            x[i] = i;
        }

    #pragma acc kernels
        for(int i = 1; i < N; i++)
        {
            x[i] = x[i] + x[i-1];
            sum+= x[i];
        }
}
```

(b) Data dependency race using kernels directive

FIGURE 9. **Race condition caused by data dependency.**

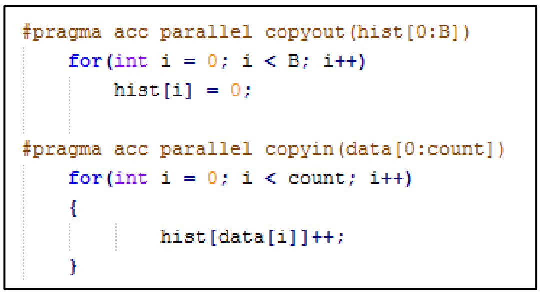
by two different threads simultaneously, which causes a race condition because of the data dependency in the second loop. Also, in Figure 9b, when using the keyword "kernels" instead of "parallel" and the loop body has data dependency, this will generate an error message on runtime, indicating illegal address during kernel execution.

In addition, some syntax errors affect loop parallelism and prevent OpenACC from being executed correctly without compiler detection or programmer awareness. The results of these errors range from wrong results to performance issues. For instance, when programmers write the OpenACC loop directives without using the "parallel" or "kernels" directives, as shown in Figure 10a, it simply executes the loop, ignoring the OpenACC directive and without compiler detection, which leads to wrong results. Similarly, if the programmers write the OpenACC directives using "parallel" instead of using "parallel loop" when they parallelize their loop, this will also not be detected by the compiler, but also will not yield wrong results. However, the loop will



```
#pragma acc data copyout(hist[0:B]) copyin(data[0:count])
{
    #pragma acc loop
        for(int i = 0; i < B; i++)
            hist[i] = 0;

    #pragma acc loop
        for(int i = 0; i < count; i++)
        {
            #pragma acc atomic update
                hist[data[i]]++;
        }
}
```

(a) Missing the keywords "parallel" and "kernels"



```
#pragma acc parallel copyout(hist[0:B])
    for(int i = 0; i < B; i++)
        hist[i] = 0;

#pragma acc parallel copyin(data[0:count])
    for(int i = 0; i < count; i++)
    {
            hist[data[i]]++;
    }
```

(b) Missing the keyword "loop"

FIGURE 10. **Syntax error caused loop parallelism errors.**

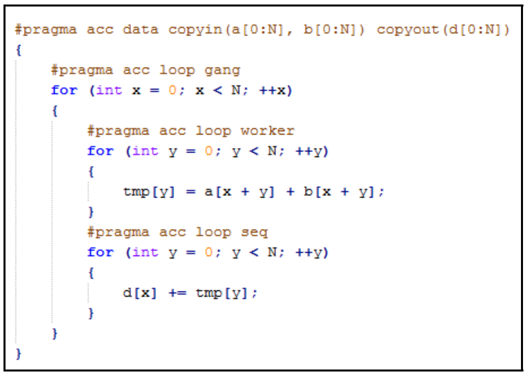
be executed sequentially without the programmers' awareness, which will affect the overall application performance, as shown in Figure 10b.

3) SHARED DATA READ AND WRITE RACE

Reading, writing, and updating to or from a shared array by multiple threads concurrently can be critical and error-prone, and all need to be handled carefully by programmers. For instance, writing data by thread "A" in a specific location and the same location is read by another thread "B"; in this case, there is a potential race condition. This shared data race can be classified into read-after-write, write-after-read, and write- after-write, but in read-only cases, there is no data race. The C and C++ programmers should use the "restrict" keyword [25] whenever the pointers are not aliased and the compiler needs to be able to parallelize the code; otherwise, it will not work in parallel. Finally, we have multiple loops, each of which has a temporary array used during its own calculation, as in Figure 11. If the array "tmp" is not declared to be private in each loop, then the "tmp" shared array might be accessed by different threads executing different loop iterations in an unpredictable way, which will cause a race condition and lead to the wrong results.

4) ASYNCHRONOUS DIRECTIVES RACE

All OpenACC directives are synchronous by default, which means that when there are operations or instructions sent from the CPU to the GPU to be processed or calculated, the CPU will wait for the GPU to complete the assigned work before continuing further CPU execution. By using synchronous operations, OpenACC ensures the operation execution order



```
#pragma acc data copyin(a[0:N], b[0:N]) copyout(d[0:N])
{
    #pragma acc loop gang
    for (int x = 0; x < N; ++x)
    {
        #pragma acc loop worker
        for (int y = 0; y < N; ++y)
        {
            tmp[y] = a[x + y] + b[x + y];
        }
        #pragma acc loop seq
        for (int y = 0; y < N; ++y)
        {
            d[x] += tmp[y];
        }
    }
}
```

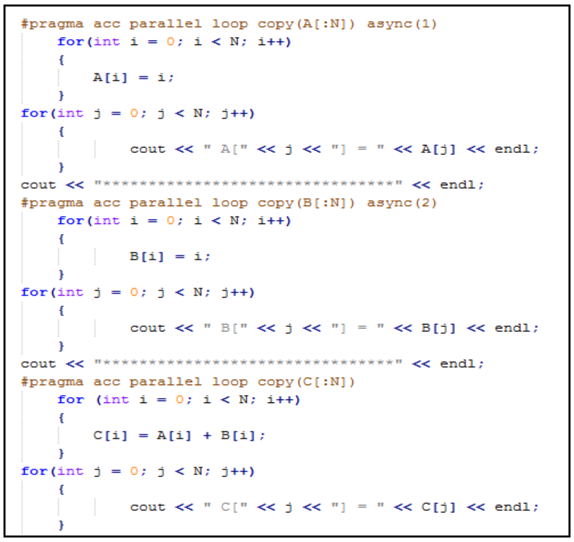
FIGURE 11. Race condition caused by shared data read and write.

when running on the GPU as run in the original program, ensuring that the program will work correctly with or without an accelerator [21]. However, while waiting for GPU computation to be completed, the system resources will be unused for a while, which is not an efficient way to run the code. Therefore, OpenACC supports asynchronous operations by assigning the work to the GPU while the CPU can complete other operations, which allows the applications to be worked asynchronously and can thus enhance efficiency.

By using the OpenACC asynchronous and wait directives, the CPU can continue working while the GPU works at the same time, allowing the system to be executed in a pipeline manner, thus enhancing performance. As a result, the programmers can ensure the synchronization of their applications when using asynchronous and wait directives to avoid causing a race condition between host and device. The race condition can be caused by misusing these directives without considering the dependency between different parts of the code, including OpenACC compute regions or data movements. An example of a race condition caused by misuse of OpenACC asynchronous and wait directives is shown in Figure 12. The array "A" will be assigned to asynchronous queue number 1, while array "B" is assigned to queue number 2, and the CPU continues working without waiting for these two queues to be completed before computing array "C". The race condition occurs as a result of dependency between different parts of the code where arrays "A" and "B" are needed before calculating array "C", which leads to wrong results.

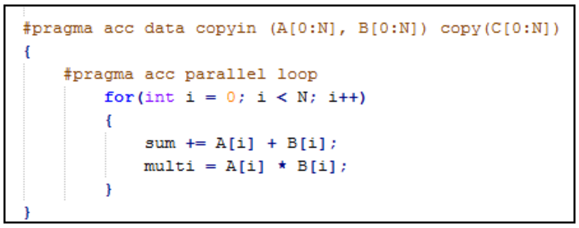
5) REDUCTION CLAUSE RACE

The OpenACC reduction clause can generate private variable copies for each loop iteration in the OpenACC compute regions, collecting and reducing all these copies into one final result based on the specified operations, which will be returned from the compute region to the CPU. The operator on the scalar variable can be specified by the reduction clause, which supports some common operations such as summation, multiplication, maximum, minimum, and various bitwise operations. Some compilers can detect reduction of the summation variable and implicitly insert the



```
#pragma acc parallel loop copy(A[:N]) async(1)
    for(int i = 0; i < N; i++)
    {
        A[i] = i;
    }
for(int j = 0; j < N; j++)
    {
        cout << " A[" << j << "] = " << A[j] << endl;
    }
cout << "********************************" << endl;
#pragma acc parallel loop copy(B[:N]) async(2)
    for(int i = 0; i < N; i++)
    {
        B[i] = i;
    }
for(int j = 0; j < N; j++)
    {
        cout << " B[" << j << "] = " << B[j] << endl;
    }
cout << "********************************" << endl;
#pragma acc parallel loop copy(C[:N])
    for (int i = 0; i < N; i++)
    {
        C[i] = A[i] + B[i];
    }
for(int j = 0; j < N; j++)
    {
        cout << " C[" << j << "] = " << C[j] << endl;
    }
```

FIGURE 12. Race condition because of the misuse of asynchronous directive.



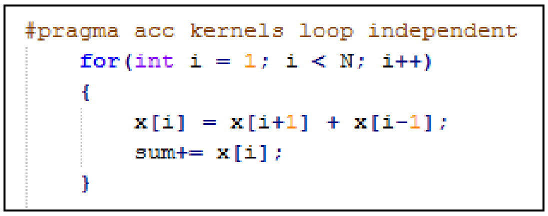
```
#pragma acc data copyin (A[0:N], B[0:N]) copy(C[0:N])
{
    #pragma acc parallel loop
        for(int i = 0; i < N; i++)
        {
            sum += A[i] + B[i];
            multi = A[i] * B[i];
        }
}
```

FIGURE 13. Reduction clause error.

reduction clause, but for other operations and other compilers, the programmers should always indicate the reductions in their codes. Even though the reduction clause can avoid some data dependencies by combining the results of each copy of the reduction variable with the original variable at the end of the compute region, the absence or misuse of this clause will lead to a race condition in some cases. Also, the variables involved in the reduction clause must be initialized properly based on the reduction clause operations before using them in the clause, or undefined behavior will result. Figure 13 shows an example of a race condition that occurs as a result of reduction clause absence. The variables "sum" and "multi" will cause a race condition resulting in wrong results, and they should be included in a reduction clause. Some compilers can detect that the variable "sum" needs to be in the reduction clause and will implicitly generate a reduction clause, but other compilers cannot detect it or the variable "multi".

6) OPENACC INDEPENDENT CLAUSE RACE

As previously discussed, the data dependency can cause a race condition in OpenACC and run-time errors, as well as preventing the code from being parallelized. The compiler data dependency analysis does not always have enough information to make a decision as to whether the code can be parallelized or work sequentially, as in the case of using



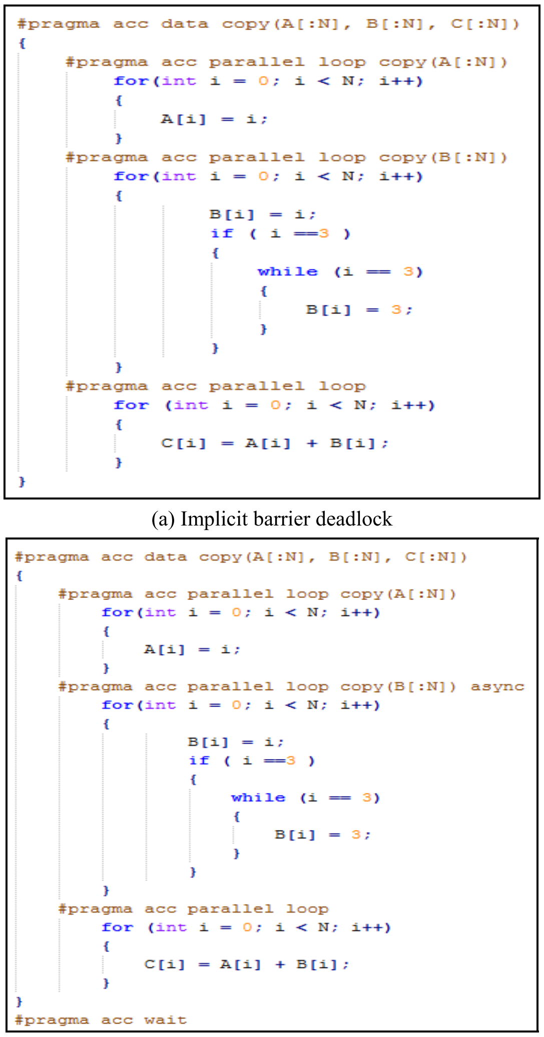
```
#pragma acc kernels loop independent
    for(int i = 1; i < N; i++)
    {
        x[i] = x[i+1] + x[i-1];
        sum+= x[i];
    }
```

FIGURE 14. **Race condition caused by misuse of OpenACC independent clause.**

OpenACC kernel directives. Therefore, programmers sometimes need to provide the compiler with this information, which can be done by using the OpenACC independent clause that tells the compiler that a specific loop is data-independent, meaning that there is no dependency or relationship between any two loop iterations, thus overriding the compiler's loop dependency analysis. However, the use of the OpenACC independent clause can be a solution, but also can cause a race condition when there is data dependency and the programmers use this clause, which allows the compiler to generate code to compute these loop iterations using independent asynchronous threads. The code shown in Figure 14 is an example of using the independent clause in a loop that includes data dependency, resulting in a race condition. In this example, the programmers tell the compilers that these loop iterations are data-independent with respect to each other, but they still have a dependency that causes a race condition, resulting in wrong and inconsistent results. Finally, programmers must be cautious when using this clause, because if any array element is written by iteration, and if there is another iteration that also writes or reads, this will cause a race condition, except for variables in the reduction clause.

D. DEADLOCK

Deadlock in OpenACC can be divided into a host (CPU) and a device (GPU) deadlock. The device deadlock can occur when two threads get stuck waiting for each other to release the lock on a shared resource. In addition, OpenACC is considered lock-free programming, which is more challenging than simply using locks in protecting critical regions, as in OpenMP [22]. In OpenACC, the host deadlock can be a result of having device livelock because of the nature of the OpenACC hidden implicit barrier at the end of each compute region, and the execution of the CPU will not proceed until all threads on the GPU have reached the end of the parallel compute region. In other words, the CPU will be waiting for the GPU to finish its work while the GPU is continuously busy because of the livelock. We call this CPU deadlock implicit barrier deadlock, as shown in Figure 15a, in which the second compute region calculating the array "B" is in an infinite loop, and the CPU is stuck waiting for all threads to reach the end of the compute region. The second case of host deadlock is similar to the previous case, but it behaves differently than when programmers use the OpenACC asynchronous and wait directives, as shown in Figure 15b. In this case, the deadlock will not occur at the end of the compute region because it



```
#pragma acc data copy(A[:N], B[:N], C[:N])
{
    #pragma acc parallel loop copy(A[:N])
        for(int i = 0; i < N; i++)
        {
            A[i] = i;
        }
    #pragma acc parallel loop copy(B[:N])
        for(int i = 0; i < N; i++)
        {
                B[i] = i;
                if ( i ==3 )
                {
                    while (i == 3)
                    {
                        B[i] = 3;
                    }
                }
        }
    #pragma acc parallel loop
        for (int i = 0; i < N; i++)
        {
            C[i] = A[i] + B[i];
        }
}
```

(a) Implicit barrier deadlock

```
#pragma acc data copy(A[:N], B[:N], C[:N])
{
    #pragma acc parallel loop copy(A[:N])
        for(int i = 0; i < N; i++)
        {
            A[i] = i;
        }
    #pragma acc parallel loop copy(B[:N]) async
        for(int i = 0; i < N; i++)
        {
                B[i] = i;
                if ( i ==3 )
                {
                    while (i == 3)
                    {
                        B[i] = 3;
                    }
                }
        }
    #pragma acc parallel loop
        for (int i = 0; i < N; i++)
        {
            C[i] = A[i] + B[i];
        }
}
#pragma acc wait
```

(b) Wait directive deadlock

FIGURE 15. **CPU deadlock because of GPU livelock.**

has been assigned to an asynchronous queue, and the CPU will proceed with the execution until it arrives at the wait directive, where the deadlock will occur. If there is no wait directive, the deadlock will occur at the end of the code. Finally, the interaction between the asynchronous and wait directives will determine how the deadlock will behave.

E. LIVELOCK

There is a similarity between livelock and deadlock, except that livelock occurs when two or more threads change their status continuously in response to other thread changes without performing any useful work. In OpenACC applications, there is a relationship between deadlock and livelock, as discussed in the previous section and shown in Figure 15. The GPU livelock is causing a deadlock in the CPU, while the CPU livelock will keep the process busy forever, and none of the processes will make any progress and will not be completed. Also in livelock, the threads might not be blocked forever, and it is hard to distinguish between livelock and other long-running processes. Finally, livelock can lead to performance and power consumption problems because of its useless busy-wait cycles.

V. ARCHITECTURE DESIGN

We designed a static testing tool for detecting OpenACC related applications errors as discussed in the previous section, and also discussed our design in [26]. Our design has the flexibility to detect actual and potential run-time errors and report them to the programmers with related error information that helps the programmer to correct them. Our architecture used a static testing technique for building a new testing tool for OpenACC systems, which will enhance the system's execution time.

Our architecture is responsible for analyzing the input source code to detect static errors before compilation. As we discussed OpenACC errors previously, we noticed that we could detect some OpenACC run-time errors from the source code, and these errors should therefore be resolved because they will definitely occur at run time. After compilation and during run time, potential run-time errors might or might not occur based on the execution behavior. We can detect the causes of these potential errors from the source code before compilation by using static testing. However, these potential errors will become run-time errors if they have not been detected, and the programmers should be warned and consider them.

Finally, it is difficult to test parallel applications due to the different factors and complicated scenarios that can cause run-time errors, as well as the nature of parallel applications and their behavior. These reasons lead to more effort to build the testing tool in terms of covering every possible scenario of the test cases and the data.

VI. STATIC TESTING APPROACH FOR DETECTNG OPENACC ERRORS

In this approach, we will check and examine different OpenACC directives and clauses to identify actual and potential run-time errors. Since there is a wide range of errors and directives to be covered, we also classify our testing approach into several classes that include OpenACC data clause checking, reduction checking, and asynchronous checking, as well as instrumenting data race and deadlock for further checking in the dynamic phase of our approach. The targeted OpenACC source code will be classified into potential error data region code, free data region code; potential error compute region code, free compute region code, and serial code. In detail, potential error regions refer to the regions with actual and potential errors, while the free regions refers to regions without errors.

Our static tool will understand the tested source code that includes C++ and OpenACC, analyzing the source code syntax and semantics to be checked to ensure its correctness. Different information will be extracted from the source code and displayed in a log file for the programmer for further debugging. This information will include the total number of compute region and structured and unstructured data regions, as well as their starts and ends in the source code. Also, the variables in each compute region will be stored with their related information, including their related compute region and which part of the equation, as well as in which loop if it is within a loop. In addition to any information related to loops, equations, and parallel threads that will be displayed in the log file with much details of the tested source code. In the following we will explain our static approach in detecting several type of OpenACC errors based on our previous classifications.

A. OPENACC DATA CLAUSE DETECTION

We assume that any OpenACC data clause is potentially error-prone because developers seem to use them inefficiently, and the compilers cannot detect any errors that related to OpenACC data clause directives. Our approach uses static testing that can statically detect OpenACC data clause related to run-time errors, including the copy, copyin, copyout as well as the other OpenACC data clauses. The data clauses related variables will be monitored and tested to ensure their correctness and to detect any unsafe behavior because these errors cannot be detected by any compiler, and they will lead to wrong results. Also, these monitored data clauses variables can be used to be instrumented for further dynamic testing during run-time. In more details, when OpenACC directives are founded in the targeted source code, our algorithm will determine the areas that have OpenACC data clauses and their type as well as the region that they belong to whether compute, structure or unstructured data regions. This mechanism will help our algorithm to call the related function to test this part of the code and check for a run-time error statically. In the case of testing the unstructured data region, there is an additional test which is counting the number of variables that appear in the enter data region, which appears in copyin or create clauses. Also, counting the number of variables that appear in copyout or delete clauses at the exit region. Then, comparing these two numbers, and they should be equal if not an error message with the related information will be sent to the user as we will explain in Listing 1.

The following algorithm in Listing 1 shows our main data clause checking algorithm, which started by exploring the targeted source code to find OpenACC directives that including data clauses. Then, saving some OpenACC data clause related information for further using in our algorithm. When a compute region, structure or unstructured data region are founded we first check their OpenACC syntax by using (*Chk_ACC_Syntax*) to find some related errors, because not all syntax errors are detected by the compiler and we will cover these errors that cannot be detected.

After completing our OpenACC syntax checking, our main checking algorithm will mapping between each directive type with its appropriate checking algorithm, which includes (*Chk_Struct*) and (*Chk_Unstruct*) to test structured, unstructured data regions and compute region that will be discussed in algorithms in Listings 2 and 3. In the case of checking

```
Chk_ACC_Code Algorithm START
SEARCH for OpenACC directives contain data clause in source code
STORE Data_Clause_Var, Data_Clause_Type, Directive_Type,
Directive_Line_No,  ACC_File_Name IN Directive_Data_Clause_List
CALL Chk_ACC_Syntax(ACC_File_Name)
WHILE Data_Clause_Var IN Directive_Data_Clause_List
  CASE Directive_Type:
   Strcut_Data_Region:
     CALL Chk_Struct(Data_Clause_Var, Data_Clause_Type)
   Unstrcut_Data_Region:
     CALL Chk_Unstruct(Data_Clause_Var, Data_Clause_Type)
   Compute_Region:
     CALL Chk_Struct(Data_Clause_Var, Data_Clause_Type)
     IF Data_Clause_Type = create or Data_Clause_Type = copyin
       STORE No. of Data_Clause_Var IN Enter_Ustruct_Data_Clause
     ELSEIF Data_Clause_Type = delete or Data_Clause_Type = copyout
       STORE No. of Data_Clause_Var IN Exit_Ustruct_Data_Clause
     ENDIF
  ENDCASE
ENDWHILE
IF Enter_Ustruct_Data_Clause = Exit_Ustruct_Data_Clause
  PASS
ELSE
  Display Error Message
ENDIF
Chk_ACC_Code Algorithm END
```

Listing. 1. The main data clause algorithm.

unstructured data regions, our approach will count the number of variables that appear in enters and exit data regions. Finally, a comparison of these two counters, if they are not equal, an error message will be issued to the developers because in unstructured data region the variables at the enter data region (in create or/and copyin data clauses) must appear at the exit data region clause (in copyout or/and delete data clauses).

The algorithm in Listing 2 is responsible for checking data clauses founded in the structured data region or compute region because they have the same behavior and roles. The targeted OpenACC data clauses in this algorithm are copy, copyin, copyout, create, and present. However, the present data clause will be instrumented for further dynamic testing because it cannot be detected during our static approach. Structured data region is defined as that part of the code that has explicit start and end points where data lifetime both begins and ends, and the memory only exists within the data region. Compute region is defined as the region in which the computation processes are executed in GPUs, whether it is a parallel region, kernel region, or serial region. Our static approach targets data clause variables, checking for their initialization, assignments, and usages in detecting any error related to any data clause. Some of these tests for the variables are similar, and some of them are different depending on the data clause related to the tested variable.

The algorithm in Listing 2 begins by receiving the tested data clause variable as well as the data clause type, and then determining the variable places in the source code to be examined in three locations: in the source code, the data region, and before and after this region. The data clause variable will be tested differently in these three locations to ensure correctness and report any errors that cannot be detected by any compiler. In addition, different data clause types will be

```
Chk_Struct Algorithm START
RECEIVE Data_Clause_Var, Data_Clause_Type
SEARCH for Data_Clause_Var locations in source code
STORE Data_Clause_Var, Data_Clause_Type, Data_Clause_Var_Line_No,
        Data_Clause_Var_File_Name, Data_Clause_Var_Location IN
        Data_Clause_Var_List
WHILE Data_Clause_Var in Data_Clause_Var_List
 CASE Data_Clause_Var_Location:
  In_Data_Region:
   IF Data_Clause_Var found
    IF Data_Clause_Var is part of an equation
     IF Data_Clause_Var is define && Data_Clause_Type = copyin
      Display Error Message
     ELSEIF Data_Clause_Var is reused && Data_Clause_Type = copyout
      Display Error Message
     ELSEIF Data_Clause_Var is reused && Data_Clause_Type = copyin
             && the define_Var is not in copy or copyin
      Display Error Message
     ELSEIF Data_Clause_Var is define && Data_Clause_Var is reused &&
             (Data_Clause_Type = copyin or Data_Clause_Type = copyout)
      Display Error Message
     ENDIF
    ELSEIF Data_Clause_Var is called by function
     IF Data_Clause_Var is called by value
      Display Error Message
     ELSEIF Data_Clause_Type = copyout
      Display Error Message
     ENDIF
    ENDIF
   ELSE
    Display Error Message
   ENDIF
  Before_Data_Region:
   IF Data_Clause_Var found
    IF Data_Clause_Var initialized
     PASS
    ELSEIF Data_Clause_Var used in the equation as define &&
            (Data_Clause_Type = create or Data_Clause_Type = copyout)
     Display Error Message
    ELSE
     Display Error Message
    ENDIF
   ELSE
    Display Error Message
   ENDIF
  After_Data_Region:
   IF Data_Clause_Var found
    IF Data_Clause_Type = create or Data_Clause_Type = copyin
     Display Error Message
    ELSE
     PASS
    ENDIF
   ELSE
    Display Error Message
   ENDIF
ENDCASE
ENDWHILE
Chk_Struct Algorithm END
```

Listing. 2. Check structured data clause algorithm.

tested differently in our algorithm based on their purpose and behavior. We studied and examined these data clauses to ensure that our approach can detect any potential run-time errors related to OpenACC data clause directives, which to the best of our knowledge have not been detected before in any compiler or testing tool. The developers will receive an error message if any incorrect situation arises.

In the case of data region testing, different situations can cause run-time errors and can be detected during the static testing phase of our algorithm and display an error message indicating the error place and type, with some additional information for the developers to consider when correcting these errors. These situations including, for example, when a copyin data clause variable is part of an equation, this will be indicated as an error because the copyin data clause will not return the values to the CPU. Therefore, any results stored in this variable will end in the GPU, and the CPU will complete execution without considering the newest result of this variable, which leads the developer to get the wrong result without the compiler knowing or detecting.

The second area tested by our algorithm is before the structure data region or the compute region. The variables' initialization and assignment will be tested in this area, detecting any potential error situations varying from one data clause type to another. One error satiation can occur in this area of code caused by the copyin data clause variable. As a result of the copyin data clause behavior that takes the variable from the CPU into the GPU, we need to determine whether these variables are initialized or have value before going to the GPU. As a result, if the copyin variables are not initialized or part of an equation, an error message will be displayed to the developer that includes the error type and the consequences of this error.

Finally, the last part of the code to be covered is after the region. In this part, we focus on the data clause variables' appearance as to whether it appears or is used after the region, which will cause a run-time error in some situations, and the developer will receive an error message indicating that. These error situations include the appearance of copyin and create data clauses variables because these data clauses will not move out of the GPU, so their results will not be considered by the CPU, and the final result will be wrong.

Regarding the unstructured data region, the algorithm in Listing 3 detects run-time errors that occur in data clauses, including create and copyin while entering the data region, as well as copyout and delete at the exit data region. In addition, the data clause variable that appears in the entering data region must appear in any data clause at the exit data region. Otherwise, it will cause undetermined behavior or run-time errors in the worst case. The unstructured data region is different from the structured data region in different aspects, including the fact that the unstructured data region can have multiple starting and ending points, can branch across multiple functions, and memory exists until explicitly deallocated. Furthermore, the enter data region directive handles device memory allocation, and the developer can only use the create or copyin data clauses. The exit data region directive handles device deallocations, and developers can only use either the copyout or the delete data clauses.

Similar to the previous algorithm, this checking process firstly receives data clause variable and type from the main algorithm and then scans the source code to allocate the variable locations in the data region, before and after. An addition check will be conducted to test the variables' appearance upon entering the region and an exit region when any variable appears at one only; an error message will be displayed to the developer. The second step is checking the data clause variable for the three locations based on their data clause type to detect any potential error and report to the developers.

```
Chk_Unstruct Algorithm START
RECEIVE Data_Clause_Var, Data_Clause_Type
SEARCH for Data_Clause_Var locations in source code
STORE Data_Clause_Var, Data_Clause_Type, Data_Clause_Var_Line_No,
        Data_Clause_Var_File_Name, Data_Clause_Var_Location IN
        Data_Clause_Var_List
WHILE Data_Clause_Var in Data_Clause_Var_List
 IF (Data_Clause_Type = copyin or Data_Clause_Type = create) &&
     (Data_Clause_Var not found at exit data region directive)
  Display Error Message
 ELSEIF (Data_Clause_Type = copyout or Data_Clause_Type = delete) &&
        (Data_Clause_Var not found at enter data region directive)
  Display Error Message
 ENDIF
 CASE Data_Clause_Var_location:
  In_Data_Region:
   IF Data_Clause_Var found
    IF Data_Clause_Var is part of an equation
     IF Data_Clause_Var is define
      IF Data_Clause_Type = delete
       Display Error Message
      ELSEIF (Data_Clause_Type = create or Data_Clause_Type =
              copyin) &&  Data_Clause_Var not found in copyout at exit
       Display Error Message
      ENDIF
     ELSEIF Data_Clause_Var is reused
      IF Data_Clause_Type = copyout
       Display Error Message
      ELSEIF Data_Clause_Type = create or Data_Clause_Type = copyin
       IF defined_var not in copyout at exit
        Display Error Message
       ELSEIF Data_Clause_Var not in delete at exit
        Display Error Message
       ENDIF
      ENDIF
     ELSEIF Data_Clause_Var is define and reused
      IF Data_Clause_Type = delete
       Display Error Message
      ELSEIF Data_Clause_Type = copyout && Data_Clause_Var not at
             enter region
       Display Error Message
      ELSEIF (Data_Clause_Type = copyin or Data_Clause_Type =
             create) && Data_Clause_Var is not in copyout at exit region
       Display Error Message
      ENDIF
     ENDIF
    ELSEIF Data_Clause_Var is called by function
     IF Data_Clause_Var is called by value
      Display Error Message
     ELSEIF Data_Clause_Var is called by reference
      IF (Data_Clause_Type = copyin or Data_Clause_Type = create) &&
          Data_Clause_Var not in delete at exit
       Display Error Message
      ELSEIF Data_Clause_Type = copyout
       Display Error Message
      ENDIF
     ENDIF
    ENDIF
   ELSE
    Display Error Message
   ENDIF
  Before_Data_Region:
   IF Data_Clause_Var found
    IF Data_Clause_Var initialized
     PASS
    ELSEIF Data_Clause_Var used in the equation as define
     IF Data_Clause_Type = create
      Display Error Message
     ELSEIF (Data_Clause_Type = copyout or Data_Clause_Type =
            delete) && Data_Clause_Var not in copyin or create at enter
      Display Error Message
     ENDIF
    ELSE
     Display Error Message
    ENDIF
   ELSE
    Display Error Message
   ENDIF
```

Listing. 3. Check unstructured data clause algorithm.

```
      After_Data_Region:
        IF Data_Clause_Var found
          IF Data_Clause_Type = delete
            Display Error Message
          ELSEIF (Data_Clause_Type = copyin or Data_Clause_Type = create)
                  && Data_Clause_Var not in copyout at exit
            Display Error Message
          ENDIF
        ELSEIF Data_Clause_Var not found
          IF Data_Clause_Type = copyout
            Display Error Message
          ELSEIF (Data_Clause_Type = copyin or Data_Clause_Type = create)
                 && Data_Clause_Var not in delete at exit
            Display Error Message
          ELSEIF Data_Clause_Var in delete and copyout at exit
            Display Error Message
          ENDIF
        ENDIF
    ENDCASE
  ENDWHILE
  Chk_Unstruct Algorithm END
```

Listing. 3. ***(Continued.)*** **Check unstructured data clause algorithm.**

One example of the checking process is checking the copyin clause in the unstructured data region, which is used to allocate memory in the GPU and copy data from the host (CPU) to the device (GPU) when entering the data region. In this case, the copyin variables on entering data should be deleted or copied out when exiting the data region. However, the compiler does not detect the related run-time errors that can occur. When the algorithm in Listing 3 deals with an unstructured data region copyin clause, there are some similarities and differences from the previous structured situation. Before the unstructured data region, our algorithm will use the same mechanism to detect any related errors that can be affected by the copyin variables. However, in the after data region, the variable will be tested if it appears after the data region, and if the same variable is not part of the copyout clause at the exit data region, there is an error, and the developer will receive an error message to address this problem. Finally, when the variable is in the unstructured data region, there are some cases where Algorithm 3 detects them as errors and sends error messages to the developers. The variables in the copyin clause must appear in either the copyout or delete clauses, but not both because this will be detected and reported as an error.

Finally, in data clause detection, the present data clause refers to the listed variables already present on the device, so no further action need be taken. This is most frequently used when a data region exists in a higher-level routine. Because there is no data movement, we do not check the variable behavior; we check only the syntax with the previously mentioned algorithms. We cannot detect run-time errors resulting from the present clause using our static approach, and s a result, we will mark this data clause for further testing because this clause needs to be tested using dynamic testing techniques.

B. OPENACC RACE CONDITION DETECTION

Different causes can result in a race condition, including executing processes by multiple threads and where the execution sequence makes a difference in the result, the execution timing, and order. Also, this happens when there are two memory accesses in the program, where they both are performed concurrently by two threads or are targeting the same location. In addition, by using OpenACC, developers cannot guarantee the thread's execution order. Because of the developers' responsibility to ensure there is no data dependency, OpenACC is more likely to have race conditions. Therefore, our static approach focuses on covering different causes of OpenACC race condition to ensure an OpenACC race-free code.

Our tool analyzes the user source code to explore and get different information to be used in the detection of OpenACC run-time errors. From the user source code, our tool obtains the following:

1) Counting the number of compute regions, structured and unstructured data regions. Determining the start and end for each of them.
2) Detecting if there is an independent clause and in which compute region they are located.
3) Creating a variable list for all variables located in each compute region and storing them in the data structure.
4) Detecting loops in each compute region for further investigation regarding the data race test, to include the following information:
   - Count the total number of for loops in the user code in the compute region only; any loop outside the compute region are not considered.
   - Determine the starting value and ending value for each for loop. Also, for the increment or decrement values, even if the ending value is a variable, our tool scans the user source code to identify the value of the ending variable in the loop.
   - Determine compute region for this loop.
5) Detecting all equations located in the compute regions.
6) Storing each array variable in each equation with its status (read/write) and its index.

Our approach has several techniques to address race conditions based on their causes, as discussed earlier. The following is our solution for each classification of race condition.

1) HOST AND DEVICE SYNCHRONIZATION RACE DETECTION

The programmer should be careful when dealing with updating data between host and device and should know when and how to do this updating. Sometimes this can cause a race condition. In this case, our static approach will investigate any update between host and device to ensure data coherence and warning the developers in case of any potential error. The data-dependency analysis can be used in some cases if it involves the updating operation. Also, our static approach will insert an instrumentation statement to check the values between host and device to ensure correctness. These instrumentation statements will capture the updated variables on the one side (GPU or CPU) before the updating process and compare the results with the developer's updated variable on the other side (GPU or CPU) and compare the two values.

If they have any differences, this will be reported as an error to the developers. These instrumentation statements will be executed during our dynamic phase.

2) LOOP PARALLELIZATION RACE DETETION

When parallelizing for loop what happened in reality, the threads are using different values of iteration variable in this loop variable and may be running in parallel at the same time. OpenACC does not make any guarantee regarding the execution order of the threads. It is even possible that the last iteration of the loop may be completed before the zero loop iteration. This will lead to a potential race condition. Therefore, our static approach will instrument the source code for data dependency analysis as well as execution order analysis. In our situation, we will focus on the OpenACC parallel region because when it has been used, the developer is responsible for ensuring that the code has enough parallelism, while the kernel directives depend on the compiler to detect whether the code has enough parallelism or should be executed sequentially. Our static approach will also mark a parallel clause for further dynamic investigation during run-time.

Our tool will investigate each compute region, loop, and equation in detail, analyzing the variables in these compute regions and their behavior and values, reporting any possible race condition to the programmer. Dependency analysis also will be conducted for any equation in any given loop to detect any actual or potential race condition. Also, the errors in loops that prevent execution will be detected in the case of writing wrong conditions or any other mistakes in the loop statement.

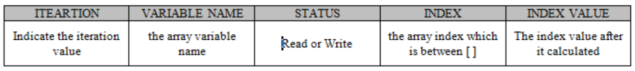
For detecting code parallelism issues, our tool will test the generation of threads, which includes gang and vectors. Our static phase is based on the analysis of the user source code and will generate gang and vectors for each compute regions to compare with the actual gang and vectors from the execution of the user code. Our static phase will insert statements in the user code to extract the actual gang and vectors from the user code to compare them with our generation of gang and vectors. This will help us to detect any related errors to include user code work sequentially rather than work in parallel because of not using OpenACC directives properly. This could help the user to enhance their code's performance because the user assumes that some compute regions work in parallel, but they actually work sequentially. The following algorithm in Listing 4 shows the process of detecting race condition in loops:

3) SHARED DATA READ AND WRITE RACE DETETION

For detecting read/write race conditions, our tool builds a table for each equation that has more than one array variable because it could have data dependency, using the related loop information to find the index values. Then, compare their values to find if there are two or more threads with writing and reading in the same variable. We store this information in the data structure, as shown in Figure 16:

```
        Chk_Data_Dep_Race_Algorithm START
1:      SEARCH for Loop inside an OpenACC Compute Region
2:      STORE Loop_ID, Loop_line_no, Loop_Iteration_Var, Loop_Start_Value,
                 Loop_End_Value, Loop_Compute_Region, Loop_Has_Independant
3:      SEARCH Equation inside the loop
4:      STORE Equation_ID, Equation_Compute_Region, Equation_Loop_ID,
                 Equation_Array_Var_Name, Equation_Array_Var_Status,
                 Equation_Array_Var_Index, Equation_Array_Var_Type,
                 Equation_Array_Var_Place
5:      IF (Equation _Array_Var_Name_1 = Equation_Array_Var_Name_2) &&
           (Equation_Compute_Region_1 = Equation_Compute_Region_2)
6:       IF Equation_Array_Var_Type = Array
7:        IF (Equation_Array_Var_Index_1 != Equation_Array_Var_Index_2) &&
             (Equation_Array_Var_Place _1 != Equation_Array_Var_Place_2 )
8:          Display Error Message Report Loop Parallelization Race Condition
9:        ELSEIF Equation In OpenACC Independent Clause
10:          Display Error Message Report Independent Clause Race Condition
11:       ENDIF
12:      ENDIF
13:     ENDIF
        Chk_Data_Dep_Race_Algorithm END
```

Listing. 4. **Data dependency race detection algorithm.**



ITEARTION	VARIABLE NAME	STATUS	INDEX	INDEX VALUE
Indicate the iteration value	the array variable name	Read or Write	the array index which is between [ ]	The index value after it calculated

FIGURE 16. **Data structure to store information for each equation.**

Our tool computes the values for each equation in a given compute region and compares to test if there is a thread read and another write in the same place; this indicates a race condition called read/write race condition. Also, if there is one thread writing to place and another thread writing to the same place, this will cause a write/read race condition. Also, in the case of two threads writing in the same variable, this causes a write/write race condition. However, in the case of two threads reading from the same variable, this will not cause a race condition. The data structure in Figure 16 will be used to detect read and write from multiple threads' race condition in our static testing tool. This will be shown in the following algorithm in Listing 5, which can detect read after write, write after read, and write after write race conditions.

4) ASYNCHRONOUS DIRECTIVES RACE DETETION

OpenACC has support for both asynchronous and wait directives. When using asynchronous computation or data movement, developers are responsible for ensuring that the program has enough synchronization to resolve any data races between the host and the GPU. In OpenACC, there is an implicit default barrier at the end of the parallel accelerator region, and the execution of the local thread will not proceed until all threads have reached the end of the parallel region. By default, all OpenACC directives are synchronous; that means that the CPU thread sends the required data and instructions to the GPU, and after that, the CPU thread will wait for the GPU to complete its work before continuing execution. Using asynchronous and wait directives allows the CPU to continue working while the GPU works at the same time, which allows the pipelining execution of the system and enhances performance. However, when developers use these

```
     Chk_R_W_Race_Algorithm START
1:   SEARCH for Equation in OpenACC Compute Region
2:   STORE Equation_ID, Equation_Compute_Region, Equation_Loop_ID,
           Equation_Array_Var_Name, Equation_Array_Var_Status,
           Equation_Array_Var_Index, Equation_Array_Var_Type
3:   IF (Equation _ID_1 = Equation_ID_2) &&
        (Equation _Array_Var_Name_1 = Equation_Array_Var_Name_2)
4:     IF (Equation_Array_Var_Index_1 != Equation_Array_Var_Index_2)
5:        IF Equation_Array_Var_Status_1 = Read &&
             Equation_Array_Var_Status_2 = Write
6:           Display Error Message Report Read after Write Race Condition
7:        ELSEIF Equation_Array_Var_Status_1 = Write &&
                 Equation_Array_Var_Status_2 = Read
8:           Display Error Message Report Write after Read Race Condition
9:        ELSEIF Equation_Array_Var_Status_1 = Write &&
                 Equation_Array_Var_Status_2 = Write
10:          Display Error Message Report Write after Write Race Condition
11:       ENDIF
12:    ENDIF
13:  ENDIF
     Chk_R_W_Race_Algorithm END
```

Listing. 5. Read write race detection algorithm.

directives without considering their system requirements, that can lead to a race condition.

In this case we focus on testing two main aspects: first, to test if there is no dependency between the asynchronous directives, and if they can work pipelining without errors. In other words, the asynchronous operation region should not have variables needed by another region; otherwise, a race condition will occur because the data will arrive before or after the time that it is needed. Second, our approach will test the completion of the asynchronous directives at the end of the region; otherwise, a deadlock situation can arise. The first situation can be tested during our static phase, and some instrumentation statements will be needed for further run-time investigation. The second case will be instrumented by our static approach and will need further dynamic testing to detect any errors that may occur. Finally, the asynchronous operation can cause deadlock and race condition and needs to be marked by our static analyzer for further testing with a dynamic approach.

5) REDUCTION CLAUSE RACE DETECTION

One potential cause of a race condition in OpenACC is the misuse of the reduction clause. Therefore, our static approach conducts extra testing for this situation to ensure that there is no misuse of this clause, as the compiler will not detect errors if there are any. In OpenACC reduction clause and for each loop iteration, variable copies are generated, and all of these private copies from different threads are reduced into one final result that returns to the CPU to be available at the end of the compute region. Reduction clause operators can be specified on the scalar variables, which include several operations, but we focus on the main following four operations: summation, multiplication, maximum and minimum operators, and others in our future works. Some compilers will detect reduction of the reduction variable and implicitly insert the reduction clause, but in other cases, it is the programmer's responsibility to indicate reductions in OpenACC codes.

Although some data dependency can be avoided by using a reduction clause, misusing the reduction clause in some cases will lead to a race condition. Also, the race condition can be caused by the absence of the reduction clause. The reduction clause combines the results for each copy of the reduction variable from different threads with the original variable at the end of the OpenACC compute region. Therefore, the variable should be initialized to some value based on the used operator before using the reduction clause. Otherwise, undefined behavior will result and lead to a wrong final result that cannot be detected by compilers. The PGI compiler can detect the summation reduction clause in some cases and add it as implicit reduction but cannot detect the other reduction operations, and the other compilers cannot detect any reduction operations.

Our static approach considers those problems and the following algorithm in Listing 6 to check the potential errors related to a reduction clause, including the reduction variables' initialization based on their operator types. Concerning reduction variables involved in multiple nested loops where two or more of these loops have associated loop directives, the reduction clause containing these variables must appear in each of those loop directives; otherwise, a race condition will occur.

Finally, our algorithm will check if any of the variables included in the OpenACC compute region have one of the reduction operations without using reduction directives; this will be reported as an error to the developers, as an absence of reduction clause can lead to the race condition. In addition to inserting some instrumentation statements for the parts that cannot be detected during our static phase for further dynamic checking during run-time, and this instrumentation will be used in case of the absence of the reduction clause when our static approach determines that the compute region has reduction operation and the developers forget to use the reduction clause. Our static approach will instrument this region by using our reduction-inserted statement.

6) INDEPENDENT CLAUSE RACE DETECTION

When the developers use the independent clause that tells the compiler that this loop is data-independent, this can cause problems if there is a dependency. The developers' responsibility is to ensure not using this clause if there is a data dependency because the independent clause allows developers to indicate that the iterations of the loop are data-independent of each other. As a result, our static approach will investigate whether the developer decision is correct by conducting data-dependency analysis in this case.

Our static tool will detect any OpenACC independent clause and determine their place in the source code and in which compute region they are located. Then, the source code will be analyzed to detect any dependency in each equation in the independent compute region to ensure there is no dependency; if there is a dependency, our tool will detect the race condition that could result from this dependency and report them to the user. This examination will be done because

```
Chk_Reduction Algorithm START
SEARCH for Red_Clause and Red_Clause_Var locations in source code
STORE Red_Clause_Var, Red_Clause_Oprator, Red_Clause_Line_No,
Red_Clause_Var_Value, Red_Clause_Var_Location IN Red_Clause_List
WHILE Red_Clause_Var in Red_Clause_List
 CASE Red_Clause_Var_Location:
  Before_Red_Clause:
   IF Red_Clause_Var initialized
    IF Red_Clause_Oprator is "+" && Red_Clause_Var_Value != 0
     Display Error Message
    ELSEIF Red_Clause_Oprator is "*" && Red_Clause_Var_Value != 1
     Display Error Message
    ELSEIF Red_Clause_Oprator is "max" &&
           Red_Clause_Var_Value ! = 0
     Display Error Message
    ELSEIF Red_Clause_Oprator is "min" &&
             Red_Clause_Var_Value < 100
     Display Error Message
    ENDIF
   ELSE
    Display Error Message
   ENDIF
  After_Red_Clause:
   IF Red_Clause part of parallel or kernel or loop directives
    IF Red_Clause_Var is part of an equation
     IF Red_Clause_Var is not a define
      Display Error Message
     ELSEIF Red_Clause_Var is a define && Red_Clause_Var
             span on nested loop
      IF Red_Clause_Var in Red_Clause not appear in multiple
          loop directives
       Display Error Message
      ENDIF
     ENDIF
    ELSE
     Display Error Message
    ENDIF
   ELSE
    Display Error Message
   ENDIF
 ENDCASE
ENDWHILE
WHILE Var in Source code
 IF Var is part of equation in OpenACC compute region
  IF Var IN " Var += " or " Var = Var + " operations
   Display Error Message
  ELSEIF Var IN " Var *= " or " Var = Var * " operations
   Display Error Message
  ELSEIF Var IN MAX function
   Display Error Message
  ELSEIF Var IN MIN function
   Display Error Message
  ENDIF
 ENDIF
ENDWHILE
Chk_Reduction Algorithm END
```

Listing. 6. Check reduction clause algorithm.

of the nature of the independent clause of the OpenACC, because it tells the compiler that the programmer is responsible for making sure this area is independent; as a result, the compiler cannot detect that and resulted in race condition. The algorithm in Listing 4 shows our static analysis of detecting data dependency and independent clause analysis.

C. OPENACC DEADLOCK DETECTION

Due to OpenACC's hiding some details, OpenACC compute regions have an implicit barrier at the end of each compute region, and we cannot be sure about the thread executions as well as the thread arrival at the end of each compute region at a certain time. Therefore, we can consider the end of each compute region as a potential deadlock point that might or might not occur. As a result, our static tool will mark the end of each compute region for further checking during runtime.

Also, one of the reasons for OpenACC deadlock in the CPU is having livelock in the GPU. This happens because of the nature of the implicit barrier at the end of each compute region. That means the GPU will be busy with the livelock while the CPU is waiting for the GPU to finish its operation. In the usage of the asynchronous directive, the GPU livelock also causes CPU deadlock, but by different behavior than in the usage of the wait directive, which will cause the CPU to have a deadlock in that statement. In this case, the deadlock behaviors are based on the asynchronous and wait directives interactions. To detect these situations, our static analysis will investigate the source code to ensure that there are no infinite loops in each compute region. Any situation that cannot be detected during our static phase will be marked for further checking.

Our static analyzer will partially detect OpenACC deadlock and mark some places of the code for further dynamic testing. The main purpose of our deadlock detection is to ensure there is no livelock in each compute region because it will cause a deadlock in the host. As a result, we analyze each loop in the compute region and examine their conditions to avoid any possibility of ‘‘always true’’ situations. However, our static analysis is limited to loops and detecting their conditions to avoid deadlock. More testing will be needed during run-time for any undetected situations, which will be marked by our static analyzer, and a warning will then be sent to the programmer.

VII. DISCUSSION

In Section 4, we identified and classified run-time errors that can occur in OpenACC applications and explained them with examples. These errors can happen because of programmers' lack of understanding of OpenACC and misuse of some OpenACC directives and clauses. Also, some of these errors occur when the programmers try to parallelize their code without fully understanding it and the regions that can be parallelized, considering different aspects as well as the data movements. In addition, some errors are related to the nature of OpenACC and how it behaves, as well as the high-level programming used in OpenACC where the programmers only use the directives without considering the operations behind these directives, which is one of the OpenACC features of which programmers should be aware.

We noticed that some syntax or logical errors caused some of the OpenACC run-time errors, but compilers do not detect these syntax errors and just ignore the directive if it is not written correctly. Some of the run-time errors might have similar names, but behave differently in OpenACC and have different causes. In our study, we included only the run-time errors in OpenACC applications that cannot be detected by the compilers. Finally, the programmers' responsibility is to understand their code and the OpenACC different data regions as well as compute regions to maximize their code

TABLE 1. Our static tool evaluation.

OpenACC Errors	Detected by Our Static Approach*
Data Clause Related Errors	
Create Clause	√
Copy Clause	√
Copyin Clause	√
Copyout Clause	√
Delete Clause	√
Present Clause	X
Memory Errors	√
Data Transmission Errors	√
Race Conditions	
Host/Device Synchronization Race	X
Loop Parallelization Race	P
Data Read and Write Race	√
Asynchronous Directives Race	X
Reduction Clause Race	√
Independent Clause Race	√
Deadlock	
Device Deadlock	X
Host Deadlock	P
Livelock	P

* The symbols are √: Fully detected; P: Partially detected and need further testing during run-time; X: Cannot be detected by our static tool due to the nature of error and its cause and need to be tested by dynamic analysis.

TABLE 2. OpenACC statics from the chosen benchmarks.

Benchmarks *	No. of OpenACC Data Clause	No. of Parallel Construct	No. of Kernels Construct	No. of OpenACC Directives
StringMatch [1]	2	1	0	2
3dstencil [1]	2	1	0	4
27stencil [2]	3	2	0	8
himeno [2]	3	2	0	4
atax [3]	5	2	0	7
bicg [3]	4	2	0	7
BT [4]	1	0	0	2
SP [4]	2	0	0	7
Reduction [5]	2	0	2	4
Stencil [5]	8	4	0	6

* The symbols are: 1: TORMENT, 2: EPCC, 3: PolyBench-ACC, 4: NAS, 5: SHOC

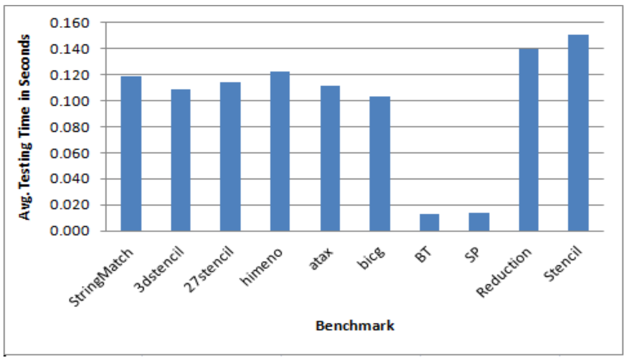


FIGURE 17. Testing time for our OpenACC static testing approach.

being parallelized and benefiting from OpenACC's capabilities and features.

In addition, we briefly explained our testing tool architecture design in Section 5, and we discussed our static approach to detect OpenACC errors in Section 6. We used several techniques and algorithms to detect different types of OpenACC errors based on their causes and how they behave. Since we are dealing with big-sized codes as well as error-prone applications, we choose to use the static approach to resolve as many errors as possible before compilation and without executing the code. This will give us the ability to analyze the code in detail and obtain full coverage of the source code. The static analysis gives us the ability to detect actual run-time errors as well as potential errors from the source code, which will be beneficial in enhancing the testing execution time by minimizing the errors that will need further testing during run-time. Finally, our static approach will mark the code that has potential errors or needs further testing during run-time, as well as determining the code parts that need inserted statements for further testing.

VIII. EVALUATION

In this section, we present the evaluation of our static testing tool and discuss our tool's capability to detect several types of OpenACC errors based on our classifications, as we discussed before, as well as measuring the performances of our testing tool. The experiments have been performed on an Intel(R) Core(TM) i7-7700HQ CPU (2.80GHz), 16 GB main memory, with an NVIDIA GeForce GTX 1050 Mobile GPU. Also, we used 50 benchmarks from five different benchmarks suites including NAS [27], SHOC [28], PolyBench-ACC [29], TORMENT OpenACC2016 [30], and EPCC [31] to conduct the experiments and measure our static approach error coverage and testing performance.

The following Table 1 shows the OpenACC errors and our static tool's ability to detect these errors. Our evaluation takes each type of OpenACC error and our tool ability to detect them fully or partially, where full detection means that our static tool can detect this error, while partial detection means that our tool can detect some cases, while other cases need to be tested during run-time or investigated more than static testing. In some cases, our tool cannot detect these errors by using static testing, and the errors need dynamic testing due to the nature of the OpenACC error and the causes behind them.

As we noticed in Table 1, the majority of OpenACC data clause-related errors can be detected and resolved by using our static approach, while race condition and deadlock can be detected partially and need further testing during run-time. However, our tool will minimize the number of errors that need to be detected during run-time, as well as the parts that need further testing as marked by our static tool. Finally, our static testing tool has successfully detected OpenACC errors including data clause-related errors, data transmission errors, and memory errors, as well as some race condition and deadlock cases.

The following Table 2 shows selective comparative study for our testing approach that from all 50 benchmarks that we experimented, we choose ten benchmarks, two from each benchmarks suites we discussed before, as a sample to be displayed on this table. This table shows some statistics about the chosen benchmarks, including some OpenACC related information we used in our static testing approach. Finally, Figure 17 shows the testing time for each of the tested benchmarks, where also will be used in our future overhead measurements when we apply our insertion statements for our future dynamic testing.

IX. CONCLUSION

OpenACC has many advantages and features that lead to increased use to achieve high parallel systems working in a heterogeneous architecture. OpenACC is designed for performance and portability that maintains existing sequential code and parallelizes it by using high-level directives without considering many details. This will attract more non-computer science specialists to use OpenACC to accelerate their systems when building powerful simulations with minimum investment of effort or time in learning how to program GPUs. Therefore, misunderstanding OpenACC directives and clauses can lead programmers to misuse them or to fail to follow the OpenACC instructions correctly. In this case, several run-time errors can occur due to OpenACC's nature as well as behavior. Programmers could cause these errors when trying to parallelize applications without following directions to avoid some of these errors.

We conducted many experiments and built several applications to test and simulate run-time errors that can occur in OpenACC, and we studied their behavior to better understand them and discover their causes and effects on the applications. Our contribution is to identify OpenACC errors that cannot be detected by compilers and that can happen without the programmers' awareness. Also, we classify these errors into five main types based on error similarity and their causes, as well as how they behave and their effects on the system. We explain these errors with examples and determine their causes with respect to application output when they occur.

Also, another main contribution is to design and build a new static testing tool capable of detecting OpenACC errors by using static testing techniques. We built a testing tool that can detect as many OpenACC errors as possible by using static testing techniques. Our tool has successfully detected OpenACC errors, including data transmission errors and memory errors, as well as some race condition and deadlock cases. Also, our tool has detected OpenACC potential errors and marked them for further testing during run-time, which will be part of our future work.

Finally, to the best of our knowledge, there is no published work that identifies or classifies OpenACC-related errors, as well as a testing tool that designated for detecting errors in OpenACC applications. In our future work, we will build a dynamic testing tool for detecting OpenACC run-time errors that we cannot detect or partially detect with our static testing tool.

ACKNOWLEDGMENT

This project was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under Grant No. (DG-015-611-1440). The authors, therefore, gratefully acknowledge the DSR technical and financial support.

REFERENCES

[1] A. M. Alghamdi and F. E. Eassa, "Parallel hybrid testing tool for applications developed by using MPI + OpenACC dual-programming model," *Adv. Sci. Technol. Eng. Syst. J.*, vol. 4, no. 2, pp. 203–210, 2019.
[2] J. F. Münchhalfen, T. Hilbrich, J. Protze, C. Terboven, and M. S. Müller, "Classification of common errors in OpenMP applications," in *Using and Improving OpenMP for Devices, Tasks, and More* (Lecture Notes in Computer Science), vol. 8766. Cham, Switzerland: Springer, 2014, pp. 58–72.
[3] M. Süß and C. Leopold, "Common mistakes in OpenMP and how to avoid them," in *OpenMP Shared Memory Parallel Programming*. Berlin, Germany: Springer, 2008, pp. 312–323.
[4] V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise predictive analysis for discovering communication deadlocks in MPI programs," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 4, pp. 1–27, Aug. 2017.
[5] T. Hilbrich, M. Weber, J. Protze, B. R. de Supinski, and W. E. Nagel, "Runtime correctness analysis of MPI-3 nonblocking collectives," in *Proc. 23rd Eur. MPI Users' Group Meeting (EuroMPI)*, 2016, pp. 188–197.
[6] S. Cook, "Common problems causes and solutions," in *CUDA Programming: A Developer's Guide to Parallel Computing With GPUs*. San Francisco, CA, USA: Morgan Kaufmann, 2012.
[7] A. M. Alghamdi and F. E. Eassa, "Software testing techniques for parallel systems: A survey," *Int. J. Comput. Sci. Netw. Secur.*, vol. 19, no. 4, pp. 176–186, 2019.
[8] R. Surendran, "Debugging, repair, and synthesis of task-parallel programs," Rice Univ., Houston, TX, USA, Tech. Rep. 10673991, 2017.
[9] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu, "UNDEAD: Detecting and preventing deadlocks in production software," in *Proc. 32nd IEEE/ACM Int. Conf. Automat. Softw. Eng.*, Oct. 2017, pp. 729–740.
[10] The Open MPI Organization. (2018). *Open MPI: Open Source High Performance Computing*. [Online]. Available: https://www.open-mpi.org/
[11] E. Saillard, P. Carribault, and D. Barthou, "PARCOACH: Combining static and dynamic validation of MPI collective communications," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 4, pp. 425–434, 2014.
[12] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang, "Symbolic analysis of concurrency errors in OpenMP programs," in *Proc. 42nd Int. Conf. Parallel Process.*, Oct. 2013, pp. 510–516.
[13] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar, "An extended polyhedral model for SPMD programs and its use in static data race detection," in *Proc. 23rd Int. Workshop Lang. Compil. Parallel Comput.*, vol. 9519, 2017, pp. 106–120.
[14] R. Sharma, M. Bauer, and A. Aiken, "Verification of producer-consumer synchronization in GPU programs," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 88–98, 2015.
[15] P. Collingbourne, C. Cadar, and P. H. J. Kelly, "Symbolic testing of OpenCL code," in *Hardware and Software: Verification and Testing*. Berlin, Germany: Springer, 2012, pp. 203–218.
[16] J. Yang, "A validation suite for high-level directive-based programming model for accelerators," Univ. Houston, Houston, TX, USA, Tech. Rep., 2015.
[17] C. Wang, R. Xu, S. Chandrasekaran, B. Chapman, and O. Hernandez, "A validation testsuite for OpenACC 1.0," in *Proc. Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2014, pp. 1407–1416.
[18] K. Friedline, S. Chandrasekaran, M. G. Lopez, and O. Hernandez, "OpenACC 2.5 validation testsuite targeting multiple architectures," in *Proc. Int. Conf. High Perform. Comput.*, 2017, pp. 557–575.
[19] M. McCorkle, "ORNL launches summit supercomputer," U.S. Dept. Energy's, Oak Ridge Nat. Lab., Oak Ridge, TN, USA, Tech. Rep. 8655747308, 2018. [Online]. Available: https://www.ornl.gov/news/ornl-launches-summit-supercomputer

[20] *The OpenACC Application Programming Interface Version 2.7*, OpenACC Specifications 2.7, 2018.
[21] S. Chandrasekaran and G. Juckeland, *OpenACC for Programmers: Concepts and Strategies*, 1st ed. Reading, MA, USA: Addison-Wesley, 2017.
[22] R. Farber, "From serial to parallel programming using OpenACC," in *Parallel Programming With OpenACC*. Amsterdam, The Netherlands: Elsevier, 2017, pp. 1–28.
[23] M. Daga, Z. S. Tschirhart, and C. Freitag, "Exploring parallel programming models for heterogeneous computing systems," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2015, pp. 98–107.
[24] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Achieving portability and performance through OpenACC," in *Proc. WACCPD 1st Work. Accel. Program. Using Directives-Held Conjunct SC Int. Conf. High Perform. Comput. Netw., Storage Anal.*, Jul. 2013, pp. 19–26, 2015.
[25] *OpenACC Programming and Best Practices Guide*, OpenACC, 2015.
[26] A. M. Alghamdi and F. Elbouraey, "A parallel hybrid-testing tool architecture for a dual-programming model," *Int. J. Adv. Comput. Sci. Appl.*, vol. 10, no. 4, pp. 394–400, 2019.
[27] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman, "OpenACC Parallelization and optimization of NAS parallel benchmarks," in *Proc. GPU Technol. Conf.*, 2014, pp. 1–27.
[28] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proc. 3rd Workshop Gen.-Purpose Comput. Graph. Process. Units (GPGPU)*, 2010, p. 63.
[29] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proc. Innov. Parallel Comput. (InPar)*, 2012, pp. 1–10.
[30] D. Barba, A. Gonzalez-Escribano, and D. R. Llanos, "TORMENT OpenACC2016: A benchmarking tool for OpenACC compilers," in *Proc. 25th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, Mar. 2017, pp. 246–250.
[31] EPCC. (2013). EPCC OpenACC benchmark suite. The University of Edinburgh. [Online]. Available: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite.

AHMED MOHAMMED ALGHAMDI was born in Jeddah, Saudi Arabia, in 1983. He received the B.Sc. degree in computer science and the first M.Sc. degree in business administration from King Abdulaziz University, Jeddah, in 2005 and 2010, respectively, and the second master's degree in Internet computing and network security from Loughborough University, U.K., in 2013. He is currently pursuing the Ph.D. degree in computer science with King Abdulaziz University. His research interests include high performance computing, parallel computing, programming models, software engineering, and software testing.

FATHY ELBOURAEY EASSA received the B.Sc. degree in electronics and electrical communication engineering from Cairo University, Egypt, in 1978, and the M.Sc. and Ph.D. degrees in computers and systems engineering from Al-Azhar University, Cairo, Egypt, in 1984 and 1989, respectively, with joint supervision with the University of Colorado, USA, in 1989. He is currently a Full Professor with the Computer Science Department, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. His research interests include agent-based software engineering, cloud computing, software engineering, big data, distributed systems, and exascale system testing.