Received April 15, 2020, accepted April 25, 2020, date of publication April 28, 2020, date of current version May 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2991009

ACC_TEST: Hybrid Testing Approach for OpenACC-Based Programs

FATHY ELBOURAEY EASSA[1], AHMED MOHAMMED ALGHAMDI[2], SEIF HARIDI[3], MAHER ALI KHEMAKHEM[1], ABDULLAH S. AL-MALAISE AL-GHAMDI[4], AND EESA A. ALSOLAMI[5]

[1]Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia
[2]Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Jeddah 21493, Saudi Arabia
[3]KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden
[4]Department of Information Systems, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia
[5]Department of Cybersecurity, College of Computer Science and Engineering, University of Jeddah, Jeddah 21493, Saudi Arabia

Corresponding author: Ahmed Mohammed Alghamdi (amalghamdi@uj.edu.sa)

This project was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under grant No. (RG-9-611-40). The authors, therefore, gratefully acknowledge the DSR technical and financial support.

ABSTRACT In recent years, OpenACC has been used in many supercomputers and attracted many non-computer science specialists for parallelizing their programs in different scientific fields, including weather forecasting and simulations. OpenACC is a high-level programming model that supports parallelism and is easy to learn to use by adding high-level directives without considering too many low-level details. Testing parallel programs is a difficult task, made even harder if using programming models, especially if they have been badly programmed. If so, it will be challenging to detect their runtime errors as well as their causes, whether the error is from the user source code or from the programming model directives. Even when these errors are detected and the source code modified, we cannot guarantee that the errors have been corrected or are still hidden. There are many tools and studies that have investigated several programming models for identifying and detecting related errors. However, OpenACC has not been targeted clearly in any testing tool or previous studies, even though OpenACC has many benefits and features that could lead to increasing use in achieving parallel systems with less effort. In this paper, we enhance ACC_TEST with the ability to test OpenACC-based programs and detect runtime errors by using hybrid-testing techniques that enhance error coverage occurring in OpenACC as well as overheads and testing time.

INDEX TERMS OpenACC, OpenACC testing tool, hybrid-testing techniques, parallel programming, ACC_TEST.

I. INTRODUCTION

Recently, OpenACC has been increasingly used in many supercomputers, including Summit [1], which is the top supercomputer in the world. OpenACC has attracted many non-computer science specialists [2] for parallelizing their programs in different scientific fields. OpenACC [3] is defined as a high-level programming model that supports parallelism in sequential programming languages, including C, C++, and Fortran. OpenACC is easy to learn and use due to its nature by adding high-level directives without considering many low-level details. As a result, programmers should fully understand OpenACC instructions; otherwise, they will cause some errors when parallelizing their codes.

Testing parallel programs is a difficult task that is even harder if using programming models, especially if they have been badly programmed. If so, it will be challenging to detect their runtime errors as well as their causes as to whether the error comes from the user source code or from the programming model directives. Even when these errors are detected and the source code modified, we cannot guarantee that the errors have been corrected or if they are still hidden.

Many studies have investigated several programming models for identifying and detecting their related errors. However, OpenACC has not been targeted clearly in any testing tool or previous study, even though OpenACC has many benefits and features that could lead to increasingly achieving parallel

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Alawneh.

systems with less effort. As a part of our previous work [4]–[7], we proposed and created a parallel hybrid-testing tool named ACC_TEST that targeted programs built in a heterogeneous architecture and covering different errors. In addition, we aim to achieve hybrid-testing techniques for detecting errors in the dual-programming models MPI + OpenACC at the end of our project. In this paper, we enhance ACC_TEST to test OpenACC-based programs and detect runtime errors by using hybrid-testing techniques. We also focus on coverage of errors that occurs in OpenACC as well as the enhancement of the execution overheads and testing times.

Our paper has been structured as the following: Sections 2 will discuss the literature review, including OpenACC overview and related work. Our techniques for testing OpenACC-based programs are explained in Section 3. In Sections 4 and 5 we discuss the implementation, testing, and evaluation of the results of our experiments. Finally, our conclusions and future work will be discussed in Section 6.

II. LITERATURE REVIEW

This literature review tries to investigate different available testing tools and techniques that detect runtime errors in parallel applications that use programming models, including homogeneous and heterogeneous systems. Also, it gives a brief overview of the OpenACC programming model. We will review and study different tools and techniques and classify them depending on various factors and characteristics. This review will investigate what still needs to be done in testing parallel systems and directions for future research in this field. Also, we only study deeply some of the reviewed testing tools and techniques based on their relation to our subject.

A. OVERVIEW OF OpenACC PROGRAMMING MODEL

OpenACC is an abbreviation for open accelerators, which have been released to accelerate codes built by C, C++, and Fortran using high-level directives for heterogeneous CPU/GPU systems. The newest version of OpenACC 3.0 [3] was released in November 2019, adding several features and advantages for supporting parallelism, including portability, compatibility, flexibility, and less programming effort and time. OpenACC directives are always set up in the following format (in C and C++) [8], [9]:

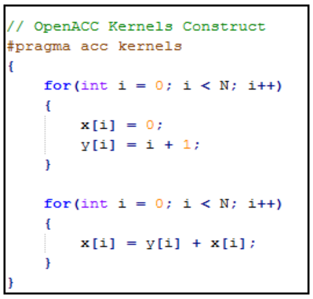
$$\#pragma\ acc\ <directive>\ [clause[[,]clause]\ldots]new-line$$

Here it is in Fortran:

$$!\$acc\ <directive>\ [clause[[,]clause]\ldots]$$

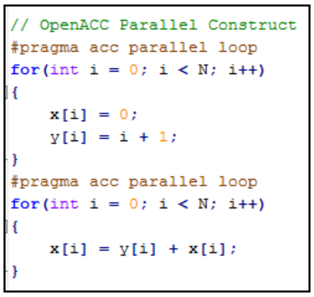
The #pragma is the keyword for a compiler directive, which is followed by the directive type that is "acc" for OpenACC directives. Then, the first word after the "acc" label is called the directive, which is an "instruction" to the compiler to do something about the following code block. OpenACC has three types of directive, including:

1) Compute directives or compute region, which mark a block of code that can be accelerated by distributing the work to multiple threads and working in parallel. These directives are parallel, kernels, routine, and loop. The main two compute directives are parallel and kernels, and we will explain the main differences between them. The kernels and parallel constructs both identify the region of code that will be parallelized; the main difference is that the kernel relies on the automatic parallelization capabilities of the compiler to analyze the region, identify which loops are safe to parallelize, and then accelerate those loops. A parallel construct relies on programmers to determine the regions, and the programmer determines whether the affected loop can be safely parallelized and allows the compiler to select how to schedule loop iterations on the target accelerator. The code in Figure 1 demonstrates the use of kernels, and the code in Figure 2 demonstrates the use of the parallel loop combined directive.



```
// OpenACC Kernels Construct
#pragma acc kernels
{
    for(int i = 0; i < N; i++)
    {
        x[i] = 0;
        y[i] = i + 1;
    }

    for(int i = 0; i < N; i++)
    {
        x[i] = y[i] + x[i];
    }
}
```

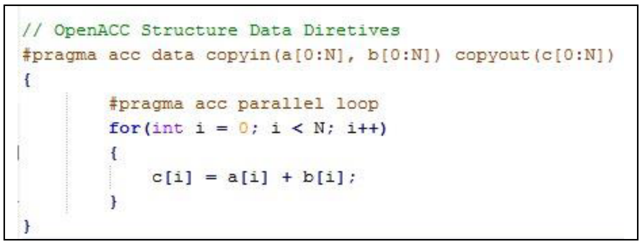
FIGURE 1. OpenACC kernels construct.



```
// OpenACC Parallel Construct
#pragma acc parallel loop
for(int i = 0; i < N; i++)
{
    x[i] = 0;
    y[i] = i + 1;
}
#pragma acc parallel loop
for(int i = 0; i < N; i++)
{
    x[i] = y[i] + x[i];
}
```

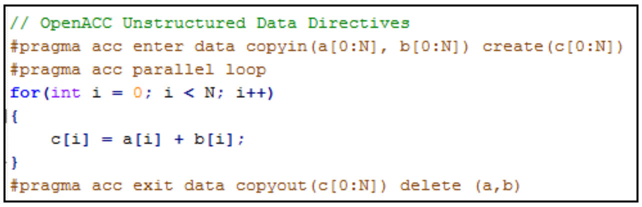
FIGURE 2. OpenACC parallel construct.

2) Data management directives or data region is used by OpenACC to avoid any unnecessary data movement between memory locations that can occur when using compute directives. Programmers can specify data lifetimes on the accelerator by using one of two types of data region directives, including structured and unstructured data directives.
The structured data directive, which defines within a single lexical scope when a data lifetime both begins and ends, is essentially owned by the accelerator. Additionally, in the structured data directives, the directive



```
// OpenACC Structure Data Diretives
#pragma acc data copyin(a[0:N], b[0:N]) copyout(c[0:N])
{
        #pragma acc parallel loop
        for(int i = 0; i < N; i++)
        {
            c[i] = a[i] + b[i];
        }
}
```

FIGURE 3. Structure OpenACC data directive.



```
// OpenACC Unstructured Data 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]) delete (a,b)
```

FIGURE 4. Unstructured OpenACC data directive.

must have explicit starting and ending points, as shown in the code in Figure 3. The data lifetime for the variable in the data clause begins at the opening curly brace (or curly bracket) and ends at the closing curly brace.
The unstructured data directive is handled differently from the previous type. An unstructured data region is delimited with a beginning-ending pair that need not be within the same lexical scope. There are two data directives that control the unstructured data directives, including "enter data" directives, which handle device memory allocation, and "exit data" directives, which handle device memory deallocation. Figure 4 demonstrates the use of unstructured data directives in C++. Six data clauses can be used on compute constructs as well as data constructs, along with one data clause unique to the exit data construct. Data clauses specify a certain data handling for the named variables and arrays. Table 1 displays the different OpenACC data clauses and their usages.

3) Synchronization directives. OpenACC also supports some task parallelism, which allows multiple constructs to be executed concurrently. To explicitly wait for one or all concurrent tasks, use the wait directive.

In terms of OpenACC parallelism, there are three levels supported by OpenACC to give programmers the ability to determine the level of parallelism in their codes. The three levels are specified by three OpenACC clauses, including vector, gang, and worker, as shown in Figure 5.

The finest granularity of parallelism is the OpenACC vector, which is defined as an individual instruction operating on multiple data. OpenACC gang represents coarse-grained parallelism, and OpenACC worker is situated between the vector and gang levels. The programmers can use these three OpenACC levels to map the parallelism in their code to any device, but it is not required. If the programmers do not use

TABLE 1. OpenACC data clauses.

Data Clause	Usage	Used In: Compute Directive	Used In: Structured Data Directive	Used In: Unstructured Data Directive
copy	Allocates memory to GPU and when entering the region copies data from CPU to GPU, and when exiting region copies data back from GPU to CPU	√	√	
copyin	Allocates memory to GPU and when entering region copies data from CPU to GPU	√	√	√
copyout	Allocates memory to GPU and copies data to CPU when exiting data region	√	√	√
create	Allocates memory to GPU without data transfer on entering data region	√	√	√
present	The variable in the present clause indicates that the variable is present in the current GPU memory	√	√	
delete	Deallocates memory from GPU without data transfer on exiting data			√

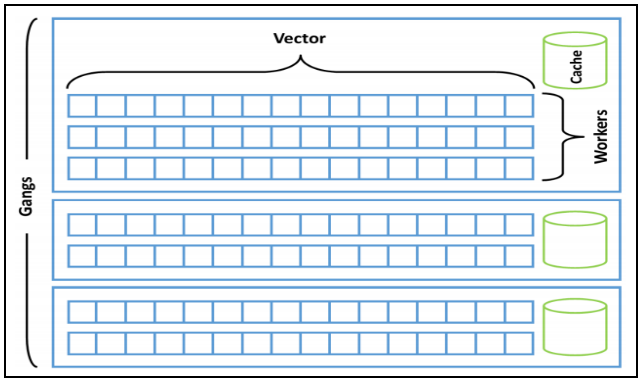


FIGURE 5. OpenACC's three levels of parallelism [10].

these three parallelism levels, the compiler will perform this mapping implicitly using the available information about the target device. Because the same code might be mapped to any number of target devices, OpenACC is considered to be highly portable.

B. RELATED WORK

High-Performance Computing (HPC) is currently a part of all scientific and manufacturing sectors driven by the improvement of HPC machines, especially with the growing attention to Exascale supercomputers, which has been projected to be feasible by 2022 in different studies [11]. This continuous improvement poses the challenging task of building massively parallel systems that can be used in these supermachines. Parallel applications have to be error-free to satisfy the application's requirements and benefits of the programming model's abilities and features. It is very difficult to test such applications because of their huge size, changeable behavior, and the integration of different programming models in the same application.

Our related work tries to investigate different available testing tools and techniques that detect runtime errors in parallel applications using programming models, including homogeneous and heterogeneous systems. We reviewed and studied different tools and techniques and classified them depending on different factors and characteristics. This review investigated what still needs to be done in testing parallel systems and directions for future research in this field. Additionally, we only study deeply some of the reviewed testing tools and techniques based on their relation to our subject. We studied more than 50 testing tools [12] and classified them according to the used testing techniques, the targeted programming models, and the runtime errors. We tried to discover the limitations and open areas for the researchers in testing parallel systems, which can yield the opportunity to focus on those areas.

Many studies investigated and targeted parallel program errors, including errors that occur in programming models, including MPI [13], OpenMP [14], CUDA [15], and OpenCL [16]. However, OpenACC has not been investigated enough or identified as thoroughly as the other programming models. OpenMP-related errors have been identified and classified in [17], where OpenMP errors were divided into defects and failures with explanations and some examples. Regarding MPI errors, a deadlock that occurs in MPI communications has been investigated and identified in [18], while in [19], the error types that apply to MPI's non-blocking collectives have been introduced. CUDA run-time errors were identified, and ways to avoid these errors were published in [20]. Finally, we identify OpenACC related errors in [6], where we classified them into different categories, including data clause-related errors, race condition, deadlock, and livelock.

Many studies have investigated several programming models for identifying and detecting related errors, such as testing tools for MPI [21], [22], OpenMP [23], [24], CUDA [25] and OpenCL [26]. However, we noted that OpenACC had not been targeted clearly in any testing tool or previous study, even though OpenACC has many benefits and features. In terms of OpenACC-related studies, there are some publications related to compilers' evaluation such as the evaluation of OpenACC 2.0 in [7], OpenACC 2.5 was evaluated in [8], and a comparison study for evaluating different compilers including CAPS, PGI, and CRAY compilers was conducted [9]. Finally, PGI has released PGI Compiler Assisted Software Testing (PCAST) [27], which is useful for detecting when results diverge between CPU and GPU versions of code, and also between the same code run on different processor architectures, but cannot detect run-time errors including data clause errors, race conditions, and deadlock in OpenACC codes.

Table 2 displays the number of testing techniques used for each programming model in the reviewed testing tools.

TABLE 2. Summary of our related work's analysis.

Testing Techniques	Programming Models					
	MPI	OpenACC	CUDA	OpenCL	OpenMP	Hybrid MPI & OpenMP
Static	1	0	2	2	4	0
Dynamic	14	0	4	1	10	2
Symbolic	0	0	4	1	1	0
Hybrid Static/Dynamic	0	0	1	0	2	2
Hybrid Static/Symbolic	0	0	2	1	0	0
Debugging	5	0	1	0	2	0

We notice that mostly dynamic techniques have been used to test MPI and OpenMP for detecting runtime errors. Symbolic testing has been used to detect runtime errors in CUDA. However, OpenACC has not been targeted to be tested by any reviewed testing tools.

Despite efforts to test parallel applications built by using programming models, there is still more work to be done with respect to high-level programming models used in heterogeneous systems.

III. OUR TECHNIQUES FOR TESTING OpenACC-BASED PROGRAMS

We proposed a hybrid approach by integrating static and dynamic testing techniques to check the actual and potential runtime errors of OpenACC programs to ensure correctness with low overhead. In our approach, the static testing would be helpful to reduce unnecessary code instrumentation during runtime detection. We classify the targeted source code into several classifications, including OpenACC data regions, OpenACC compute regions, and Non-OpenACC regions. Our main focus will be on the first two classifications and further classifications for them in the static testing phase. However, the last classification might be used for monitoring some variables used in related OpenACC clauses or directives.

In our solution, different OpenACC directives and clauses will be checked and examined for identifying actual and potential errors. Because of the wide range of errors and directives to be covered, our testing approach will be classified into several classes, including data clauses, race conditions, and deadlock checking. The targeted OpenACC source code is classified into potential error region code, which refers to the regions with actual and potential errors, while free region code refers to regions without errors.

OpenACC data clause-related errors, reduction clause, and asynchronous operations, as well as data dependency analysis and infinite loop detections, will be checked by our static approach. The tested source code will be understood, and their syntax and semantics will be analyzed to ensure its correctness.

Our static analyzer will extract some information from the source code to be stored in a log file for further debugging. Finally, our static approach will help to determine the code's parts that need to be instrumented for further testing during our dynamic approach. This instrumentation will include deadlock and race detections as well as some data clauses that need to be tested during runtime. We tried to minimize the need for inserting instrumentation statements to improve our approach performance and reduce the overhead resulting from the dynamic testing. We started to detect any OpenACC error by using our static phase if possible and then completed the detection by using our dynamic phase.

Our dynamic approach will use our static analysis insertion statements to execute the dynamic testing during runtime and detect any runtime errors that cannot be detected during the static approach, including deadlock, race condition, and present clause-related errors. Our dynamic approach will use the annotation from our static approach to instrument the insertion statements in the user source code, and our dynamic approach will detect these errors during runtime. These instrumentation statements will be executed during runtime to check or verify some potential errors determined by the static phase and to instrument extra inserted statements during the runtime for some regions that have been marked and inserted by our static approach.

Also, our dynamic approach will be used to collect some information related to the memory address, thread executions, and variable values to be used for extra testing or for further debugging. The resulting code from our dynamic testing approach will include the user code and the instrumented inserted test code.

Finally, some historical information will be collected as well during our dynamic phase and stored in a log file to be used in debugging and tracking the changes in the user codes. Errors resulting from our dynamic phase will be shown in the dynamic analysis error report, while the errors resulting from our static phase will be shown in the static analysis report list. By the end of our testing for each file, four different files will result from our testing, including static error report, source code with inserted testing code, dynamic error report, and historical log file.

In our previous paper [6], we explained how our static approach detects different OpenACC errors with the explanation of several algorithms. Where we have different algorithms dedicated to detect OpenACC static errors from the targeted source code. 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.

IV. IMPLEMENTATION AND TESTING

Implementing and testing our testing tool is required to verify and validate our proposed techniques. We conducted several experiments that cover several scenarios and test suites for testing our proposed solution and ensure our tool's ability to detect errors in OpenACC. The technical information of our experimental environment includes an Intel(R) Core(TM) i7-7700HQ CPU (2.80GHz), 16 GB main memory, with an NVIDIA GeForce GTX 1050 Mobile GPU, which has 768 NVIDIA CUDA cores, 4 GB GDDR5 RAM, and memory speed 7 Gbps.

In terms of OpenACC testing, five different benchmark suites including 50 benchmarks have been used for evaluating the ACC_TEST and measuring our hybrid approach error coverage and testing performance. These benchmark suites include SHOC [28], PolyBench-ACC[29], EPCC [30], NAS [31], and TORMENT OpenACC2016 [32]. Some statistics of the chosen benchmarks are shown in Table 3, including the benchmarks that will be used for evaluating ACC_TEST. These benchmarks have been chosen to evaluate our testing tool performance and the ability to deal with various OpenACC data clauses, parallel and kernel construct.

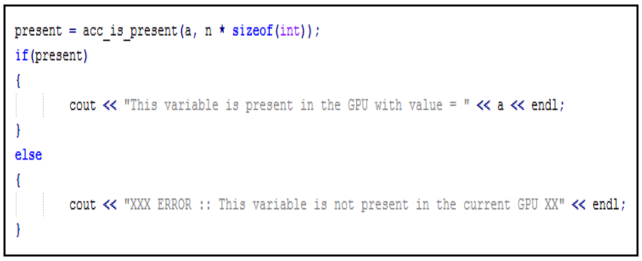
TABLE 3. OpenACC statistics from the chosen benchmarks.

Benchmarks *	No. of OpenACC Data Clause	No. of Parallel Construct	No. of Kernels Construct	No. of OpenACC Directives
TORMENT				
StringMatch	**2**	**1**	**0**	**2**
3dstencil	**2**	**1**	**0**	**4**
EPCC				
27stencil	**3**	**2**	**0**	**8**
himeno	**3**	**2**	**0**	**4**
PolyBench-ACC				
atax	**5**	**2**	**0**	**7**
bicg	**4**	**2**	**0**	**7**
NAS				
BT	**1**	**0**	**0**	**2**
SP	**2**	**0**	**0**	**7**
SHOC				
Reduction	**2**	**0**	**2**	**4**
Stencil	**8**	**4**	**0**	**6**

In the following three categories, we will discuss our tool's ability to detect errors in OpenACC-related programs based on our error classifications.

A. OpenACC DATA CLAUSE DETECTION

We test the ability of ACC_TEST to detect the actual and potential runtime errors of OpenACC. We noticed that the majority of OpenACC data clause-related errors could be detected during our static testing, but the OpenACC present data clause cannot be detected because of its nature. When the user code and the inserted test code move to the instrumenter phase, our instrumenter will remove any comment character followed by our inserted label "ASSERT" to keep the insert test code to be compiled, as shown in Figure 6, including an



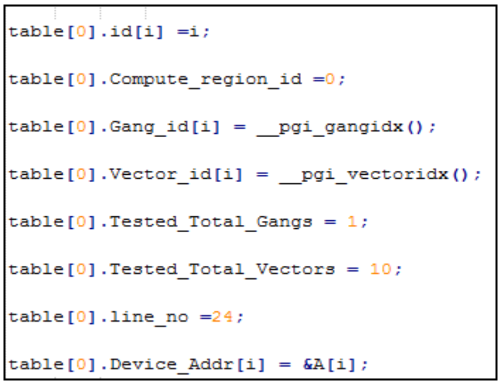
```
present = acc_is_present(a, n * sizeof(int));
if(present)
{
        cout << "This variable is present in the GPU with value = " << a << endl;
}
else
{
        cout << "XXX ERROR :: This variable is not present in the current GPU XX" << endl;
}
```

FIGURE 6. The instrumented inserted statements for detecting OpenACC present clause errors.

example of the actual instrumented code to be used during our dynamic testing phase.

B. OpenACC RACE CONDITION DETECTION

In terms of race condition detection, especially loop parallelization race detection, ACC_TEST will have the ability to test the thread's generation, including OpenACC gang and vectors. Based on the user source code analysis, the ACC_TEST static approach will generate threads for each compute region for comparing them with the actual number of threads during runtime when executing the user source code. The ACC_TEST static approach will annotate the user source code, and some statements will be inserted into the user code for detecting the actual number of threads. Figure 7 demonstrates the instrumented code to be used by our dynamic tester.

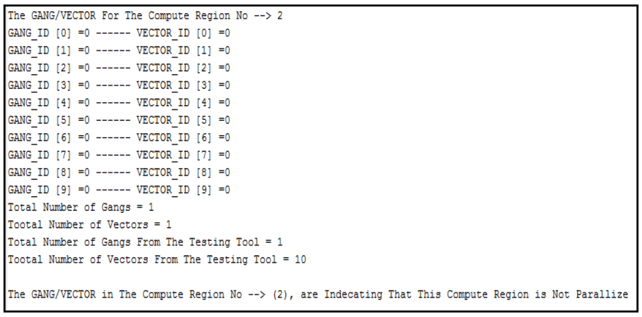


```
table[0].id[i] =i;

table[0].Compute_region_id =0;

table[0].Gang_id[i] = __pgi_gangidx();

table[0].Vector_id[i] = __pgi_vectoridx();

table[0].Tested_Total_Gangs = 1;

table[0].Tested_Total_Vectors = 10;

table[0].line_no =24;

table[0].Device_Addr[i] = &A[i];
```

FIGURE 7. Example of collecting information related to the compute region parallelism.

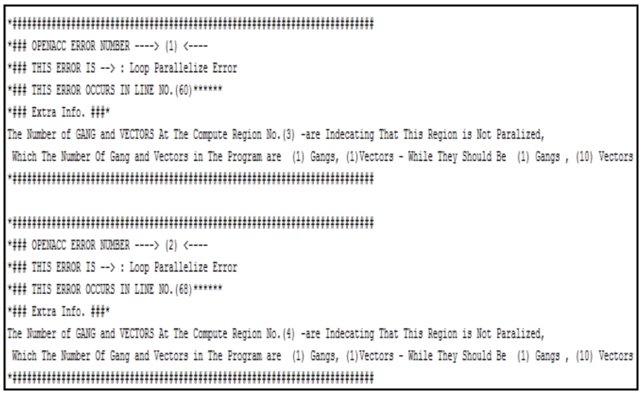
After collecting all information from all threads in the compute region, the inserted code will be used for testing the actual parallelism situation in the compute region and report any error to the developers. It shows the process of inserting and comparing the threads gang and vectors between the actual number of gang and vectors from our dynamic approach and the generated gang and vectors by our static approach, and the respective collected information will appear in the dynamic analysis log file, as shown in Figure 8.

Any differences between them will be reported to the developers, indicating that the targeted compute region is not parallelized. This will help the developers to ensure that any compute region they want to parallelize will be detected and therefore create their code parallelism. The related error message indicating a compute region that is not parallelized is shown in Figure 9, which has been detected by our dynamic analysis.



```
The GANG/VECTOR For The Compute Region No --> 2
GANG_ID [0] =0 ------ VECTOR_ID [0] =0
GANG_ID [1] =0 ------ VECTOR_ID [1] =0
GANG_ID [2] =0 ------ VECTOR_ID [2] =0
GANG_ID [3] =0 ------ VECTOR_ID [3] =0
GANG_ID [4] =0 ------ VECTOR_ID [4] =0
GANG_ID [5] =0 ------ VECTOR_ID [5] =0
GANG_ID [6] =0 ------ VECTOR_ID [6] =0
GANG_ID [7] =0 ------ VECTOR_ID [7] =0
GANG_ID [8] =0 ------ VECTOR_ID [8] =0
GANG_ID [9] =0 ------ VECTOR_ID [9] =0
Total Number of Gangs = 1
Tootal Number of Vectors = 1
Total Number of Gangs From The Testing Tool = 1
Tootal Number of Vectors From The Testing Tool = 10

The GANG/VECTOR in The Compute Region No --> (2), are Indecating That This Compute Region is Not Parallize
```

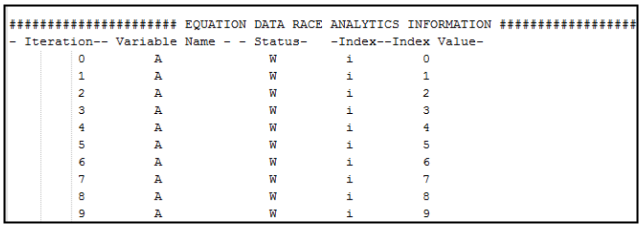
FIGURE 8. Information collected from our dynamic analyzer indicating the compute region parallelism.



```
*##################################################################
*### OPENACC ERROR NUMBER ----> (1) <----
*### THIS ERROR IS --> : Loop Parallelize Error
*### THIS ERROR OCCURS IN LINE NO.(60)******
*### Extra Info. ###*
The Number of GANG and VECTORS At The Compute Region No.(3) -are Indecating That This Region is Not Paralized,
 Which The Number Of Gang and Vectors in The Program are  (1) Gangs, (1)Vectors - While They Should Be  (1) Gangs , (10) Vectors
*##################################################################

*##################################################################
*### OPENACC ERROR NUMBER ----> (2) <----
*### THIS ERROR IS --> : Loop Parallelize Error
*### THIS ERROR OCCURS IN LINE NO.(68)******
*### Extra Info. ###*
The Number of GANG and VECTORS At The Compute Region No.(4) -are Indecating That This Region is Not Paralized,
 Which The Number Of Gang and Vectors in The Program are  (1) Gangs, (1)Vectors - While They Should Be  (1) Gangs , (10) Vectors
*##################################################################
```

FIGURE 9. Error messages resulting from our dynamic tester indicating loop parallelize error.

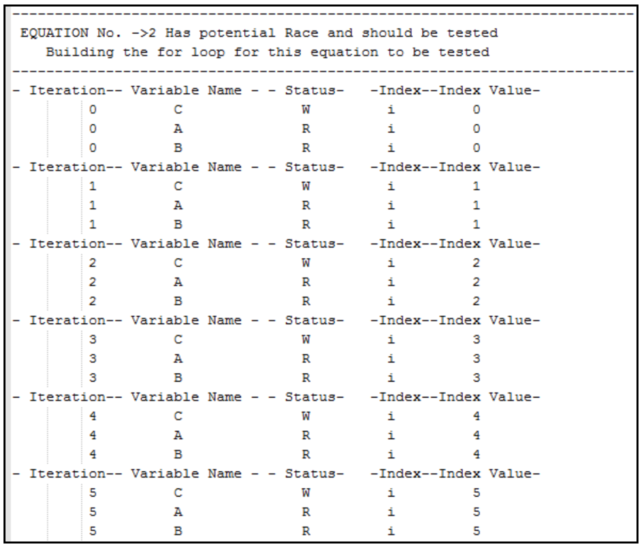
In another case, ACC_TEST will build a table for each equation for detecting read/write race condition. This table will store data for each equation that has more than one array variable because of the possibility of data dependency. ACC_TEST will compare the table's values to detect any reading and writing to the same variable. The equation data race analytic information will be stored in the data structure shown in Figure 10:



```
####################### EQUATION DATA RACE ANALYTICS INFORMATION ##################
- Iteration-- Variable Name - - Status-   -Index--Index Value-
            0         A             W         i        0
            1         A             W         i        1
            2         A             W         i        2
            3         A             W         i        3
            4         A             W         i        4
            5         A             W         i        5
            6         A             W         i        6
            7         A             W         i        7
            8         A             W         i        8
            9         A             W         i        9
```

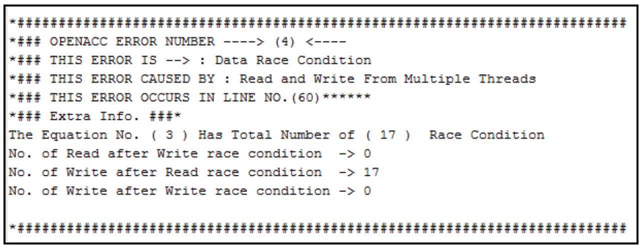
FIGURE 10. Example from ACC_TEST for stored data structure for each equation.

The values of each equation in a given compute region will be computed and compared for testing the case of reading and writing in the same place; this case will be detected as a read/write race condition. Additionally, if there are two threads writing in the same variable, this case will be detected as write/write race, but if these two threads are reading from the same place, no error is indicated. If our static approach detects a potential race condition, it will create a data structure and collect the related information as shown in Figure 10. In the case of detecting race conditions that resulted from reading and writing from multiple threads, the data structure in Figure 11 will be used.



```
---------------------------------------------------------------------------
 EQUATION No. ->2 Has potential Race and should be tested
    Building the for loop for this equation to be tested
---------------------------------------------------------------------------
- Iteration-- Variable Name - - Status-   -Index--Index Value-
         0         C            W           i         0
         0         A            R           i         0
         0         B            R           i         0
- Iteration-- Variable Name - - Status-   -Index--Index Value-
         1         C            W           i         1
         1         A            R           i         1
         1         B            R           i         1
- Iteration-- Variable Name - - Status-   -Index--Index Value-
         2         C            W           i         2
         2         A            R           i         2
         2         B            R           i         2
- Iteration-- Variable Name - - Status-   -Index--Index Value-
         3         C            W           i         3
         3         A            R           i         3
         3         B            R           i         3
- Iteration-- Variable Name - - Status-   -Index--Index Value-
         4         C            W           i         4
         4         A            R           i         4
         4         B            R           i         4
- Iteration-- Variable Name - - Status-   -Index--Index Value-
         5         C            W           i         5
         5         A            R           i         5
         5         B            R           i         5
```

FIGURE 11. Our static phase detecting potential race condition and collecting more information for further testing.

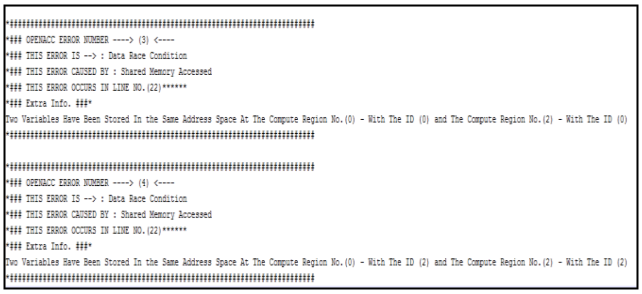


```
*##########################################################################
*### OPENACC ERROR NUMBER ----> (4) <----
*### THIS ERROR IS --> : Data Race Condition
*### THIS ERROR CAUSED BY : Read and Write From Multiple Threads
*### THIS ERROR OCCURS IN LINE NO.(60)******
*### Extra Info. ###*
The Equation No. ( 3 ) Has Total Number of ( 17 )  Race Condition
No. of Read after Write race condition  -> 0
No. of Write after Read race condition  -> 17
No. of Write after Write race condition -> 0

*##########################################################################
```

FIGURE 12. Detected race condition caused by reading and writing from multiple threads.

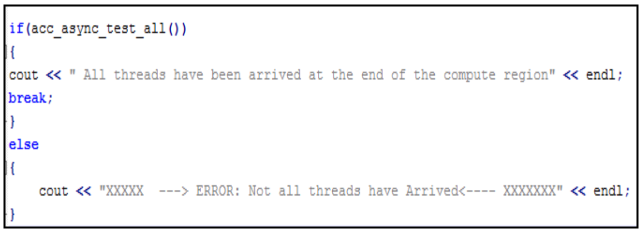
Figure 12 shows the error message that appears to the developer in the case of a race condition resulting from reading and writing to the same address space by different threads. In addition, Figure 13 shows an error message that resulted from our dynamic analysis, indicating a race condition.

In our dynamic phase, OpenACC asynchronous directives will be tested by detecting any errors using the OpenACC API function. In more detail, the API function that will be instrumented at our static phase and used during runtime are (acc_async_test) and (acc_async_test_all) to test the completion of specific or all asynchronous directives. The code in Figures 14 and 15 show our instrumented insertion statements if our static tester found an OpenACC asynchronous directive and marked them for our insertion mechanism to insert the test code and move them to the



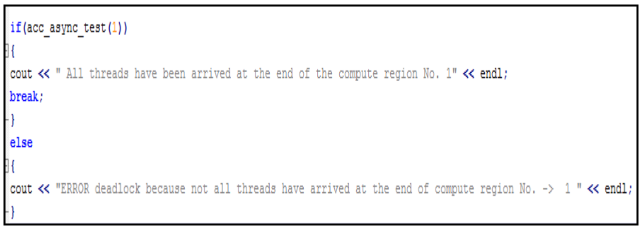
```
*####################################################################
*### OPENACC ERROR NUMBER ----> (3) <----
*### THIS ERROR IS --> : Data Race Condition
*### THIS ERROR CAUSED BY : Shared Memory Accessed
*### THIS ERROR OCCURS IN LINE NO.(22)******
*### Extra Info. ###*
Two Variables Have Been Stored In the Same Address Space At The Compute Region No.(0) - With The ID (0) and The Compute Region No.(2) - With The ID (0)
*####################################################################

*####################################################################
*### OPENACC ERROR NUMBER ----> (4) <----
*### THIS ERROR IS --> : Data Race Condition
*### THIS ERROR CAUSED BY : Shared Memory Accessed
*### THIS ERROR OCCURS IN LINE NO.(22)******
*### Extra Info. ###*
Two Variables Have Been Stored In the Same Address Space At The Compute Region No.(0) - With The ID (2) and The Compute Region No.(2) - With The ID (2)
*####################################################################
```

FIGURE 13. Error race condition caused by witting and reading from the same address space detected by our dynamic tester.



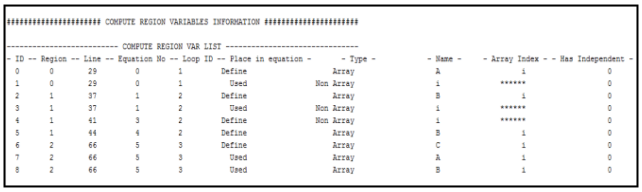
```
if(acc_async_test_all())
{
cout << " All threads have been arrived at the end of the compute region" << endl;
break;
}
else
{
    cout << "XXXXX  ---> ERROR: Not all threads have Arrived<---- XXXXXXX" << endl;
}
```

FIGURE 14. Analysis dynamic testing to test the completion of all OpenACC asynchronous directives.



```
if(acc_async_test(1))
{
cout << " All threads have been arrived at the end of the compute region No. 1" << endl;
break;
}
else
{
cout << "ERROR deadlock because not all threads have arrived at the end of compute region No. ->  1 " << endl;
}
```

FIGURE 15. Dynamic testing to test the completion of specific OpenACC asynchronous directives.



```
###################### COMPUTE REGION VARIABLES INFORMATION ######################

------------------------- COMPUTE REGION VAR LIST ------------------------------
- ID -- Region -- Line -- Equation No -- Loop ID -- Place in equation -      - Type -          - Name -     - Array Index - - Has Independent -
  0       0        29        0             1        Define                  Array              A              i                  0
  1       0        29        0             1         Used               Non Array              i           ******                0
  2       1        37        1             2        Define                  Array              B              i                  0
  3       1        37        1             2         Used               Non Array              i           ******                0
  4       1        41        3             2        Define              Non Array              i           ******                0
  5       1        44        4             2        Define                  Array              B              i                  0
  6       2        66        5             3        Define                  Array              C              i                  0
  7       2        66        5             3         Used                   Array              A              i                  0
  8       2        66        5             3         Used                   Array              B              i                  0
```

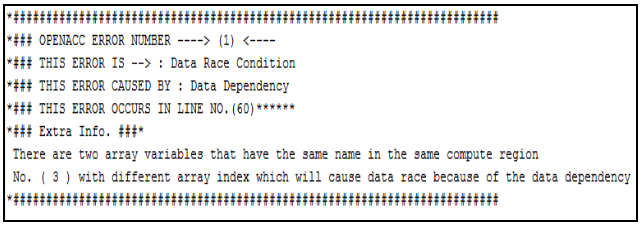
FIGURE 16. Sender compute region variable information resulting from our static approach.

dynamic tester to complete the dynamic testing during runtime after instrumentation.

In the case of data race caused by an independent clause, ACC_TEST static analyser detects any independent clause, determining the clause's place and the related compute region. After that, our static approach will analyze the source code to detect any dependency in each equation in the independent compute region to ensure there is no dependency. In the case of a dependency situation, the race condition will be detected by ACC_TEST and reported to the user, as shown in Figure 17. Figure 16 shows our static approach information collection for all compute regions variables.

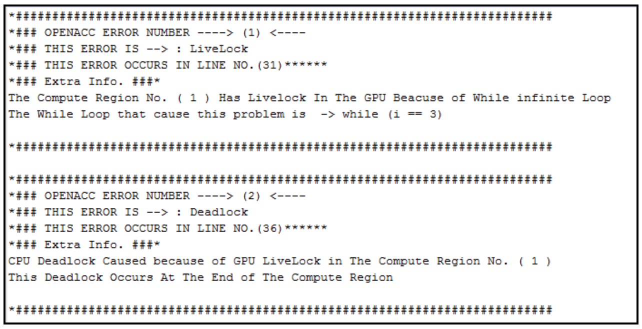
C. OpenACC DEADLOCK DETECTION

Each compute region's end will considered as a potential error point that might occur or not, including deadlock. Therefore, the ACC_TEST static analyzer annotates each



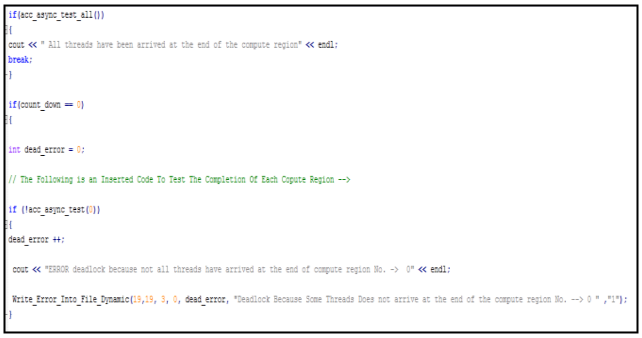
```
*###################################################################
*### OPENACC ERROR NUMBER ----> (1) <----
*### THIS ERROR IS --> : Data Race Condition
*### THIS ERROR CAUSED BY : Data Dependency
*### THIS ERROR OCCURS IN LINE NO.(60)******
*### Extra Info. ###*
 There are two array variables that have the same name in the same compute region
 No. ( 3 ) with different array index which will cause data race because of the data dependency
*###################################################################
```

FIGURE 17. Error message indicating race condition caused by data dependency.



```
*##########################################################################
*### OPENACC ERROR NUMBER ----> (1) <----
*### THIS ERROR IS --> : LiveLock
*### THIS ERROR OCCURS IN LINE NO.(31)******
*### Extra Info. ###*
The Compute Region No. ( 1 ) Has Livelock In The GPU Beacuse of While infinite Loop
The While Loop that cause this problem is  -> while (i == 3)

*##########################################################################

*##########################################################################
*### OPENACC ERROR NUMBER ----> (2) <----
*### THIS ERROR IS --> : Deadlock
*### THIS ERROR OCCURS IN LINE NO.(36)******
*### Extra Info. ###*
CPU Deadlock Caused because of GPU LiveLock in The Compute Region No. ( 1 )
This Deadlock Occurs At The End of The Compute Region

*##########################################################################
```

FIGURE 18. Error message for developers indicating deadlock and livelock errors.

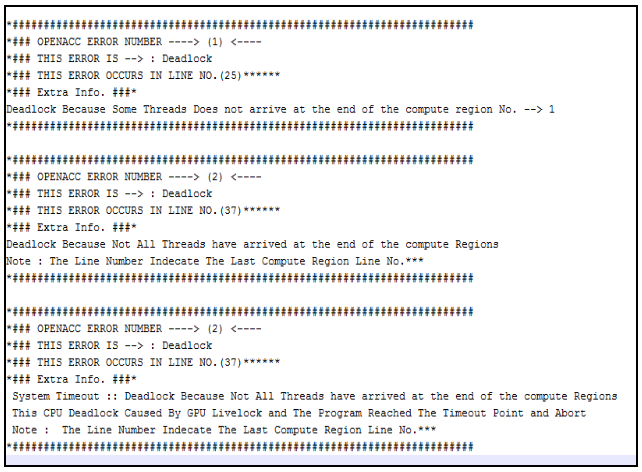


```
if(acc_async_test_all())
{
cout << " All threads have been arrived at the end of the compute region" << endl;
break;
}

if(count_down == 0)
{

int dead_error = 0;

// The Following is an Inserted Code To Test The Completion Of Each Copute Region -->

if (!acc_async_test(0))
{
dead_error ++;

 cout << "ERROR deadlock because not all threads have arrived at the end of compute region No. ->  0" << endl;

 Write_Error_Into_File_Dynamic(19,19, 3, 0, dead_error, "Deadlock Because Some Threads Does not arrive at the end of the compute region No. --> 0 " ,"1");
}
```

FIGURE 19. Inserted instrumented test code for detecting OpenACC deadlock in each compute region.

OpenACC compute region's end for further investigation during runtime. Our dynamic approach will be using the instrumentations annotated in our static phase to test the arrival of all threads at the end of each compute region because we assume that every parallel compute region could potentially have a deadlock. Therefore, our dynamic phase will check at every compute region the number of threads included in the region and compare them to the number of threads at the end of the region. After instrumentation of the insert test code, our dynamic tester will investigate which compute region has caused this deadlock and report to the developer with an appropriate message indicating the error type, the compute region number, and the line number. Figure 18 shows a sample error message from our static analysis phase that is displayed to the developer, and Figure 19 shows a sample of the instrumented inserted test code.

This combination of using hybrid testing (static and dynamic) will ensure the correctness of the user code and detect any potential deadlock. Figure 20 shows the error messages that resulted from our dynamic testing.



```
*##########################################################################
*### OPENACC ERROR NUMBER ----> (1) <----
*### THIS ERROR IS --> : Deadlock
*### THIS ERROR OCCURS IN LINE NO.(25)******
*### Extra Info. ###*
Deadlock Because Some Threads Does not arrive at the end of the compute region No. --> 1
*##########################################################################

*##########################################################################
*### OPENACC ERROR NUMBER ----> (2) <----
*### THIS ERROR IS --> : Deadlock
*### THIS ERROR OCCURS IN LINE NO.(37)******
*### Extra Info. ###*
Deadlock Because Not All Threads have arrived at the end of the compute Regions
Note : The Line Number Indecate The Last Compute Region Line No.***
*##########################################################################

*##########################################################################
*### OPENACC ERROR NUMBER ----> (2) <----
*### THIS ERROR IS --> : Deadlock
*### THIS ERROR OCCURS IN LINE NO.(37)******
*### Extra Info. ###*
 System Timeout :: Deadlock Because Not All Threads have arrived at the end of the compute Regions
 This CPU Deadlock Caused By GPU Livelock and The Program Reached The Timeout Point and Abort
 Note :  The Line Number Indecate The Last Compute Region Line No.***
*##########################################################################
```

FIGURE 20. Deadlock detected by our dynamic analyzer.

TABLE 4. Our hybrid testing tool error coverage for openACC.

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

* The symbols are √: Fully Detected; P: Partially Detected; X: Not Detected.

V. DISSCUSSION AND EVALUATION

In this section, we evaluate our enhancement of ACC_TEST and disscuss its ability to detect OpenACC errors based in our prevous paper [6]. The main aspects of our evaluation will include ACC_TEST capability, performance, size, and scalability as well as measuring different overheads. In terms of the error coverage, we will discuss ACC_TEST capability of testing OpenACC-related applications, as well as evaluating them by using the appropriate test suite.

Table 4 shows ACC_TEST and its ability to detect OpenACC-related errors classified by error types. We evaluate ACC_TEST's ability to detect each type of OpenACC error fully or partially, where full detection means that our used approach can detect this error, while partial detection means that ACC_TEST can detect some cases, while other cases need to be tested during runtime or investigated with more than static testing. The table shows ACC_TEST's ability to detect any error with each approach.

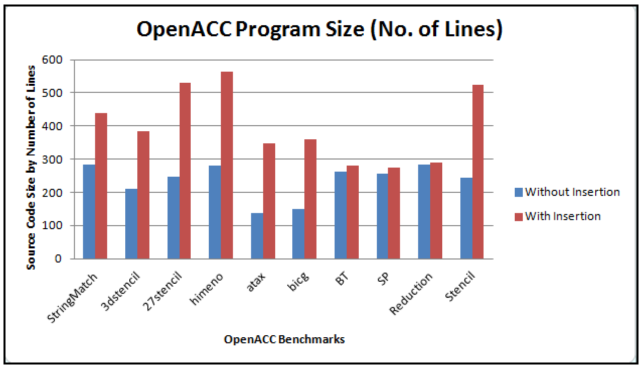


FIGURE 21. OpenACC program size overhead (by number of lines).

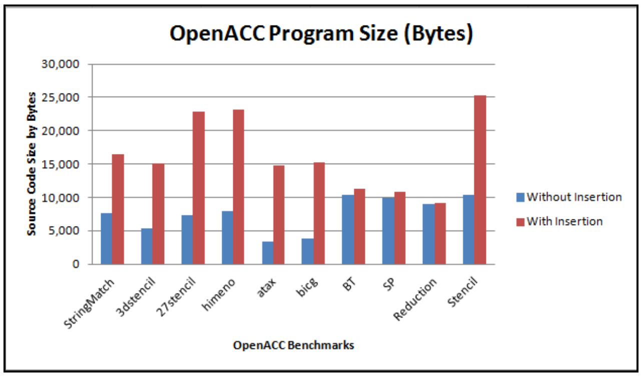


FIGURE 22. OpenACC program size overhead (by bytes).

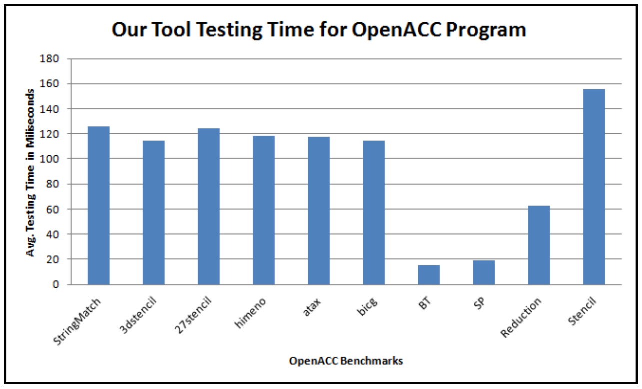


FIGURE 23. Testing time for OpenACC related program in milliseconds.

From Table 4 we note that our static approach can resolve and detect the majority of OpenACC data clause-related errors. In addition, our static analysis can partially detect race condition and deadlock and annotated for further testing during runtime. However, ACC_TEST will minimize the number of errors that need to be detected during runtime, as well as the parts that need further testing as marked by our static tool to enhance ACC_TEST's performance and minimize the overhead resulting from the dynamic testing approach. In terms of OpenACC data clause-related errors, our dynamic approach will be used to detect the present clause only due to its nature that cannot be detected from the source code with our static approach and needs investigation during runtime. As a result, ACC_TEST minimizes the execution overhead by using our static approach and resolving the majority of the data clause-related errors. Finally, ACC_TEST has successfully detected all OpenACC errors included in our classification, including data clause-related errors, data transmission errors, and memory errors, as well as race conditions and deadlock cases.

The following figures show the resulting size overhead from our insertion mechanism, as well as the testing time for each of the tested benchmarks. We measure the size overhead by using Equation 1: *Measuring the Size Overhead*, as shown at the bottom of this page.

As noted in Figure 21, the average size overhead by number of lines of our chosen benchmarks is 78% because the majority of the tested benchmarks have several OpenACC parallel constructs, which will be tested during our dynamic phase to detect any race condition and deadlock as well as ensuring the loops' parallelization. In more detail, the benchmarks BT, SP, and Reduction have an average of only 6% size overhead because they do not include any OpenACC parallel constructs, while the benchmark Stencil has 116% size overhead because it has four parallel constructs. However, these size overheads will not affect the user code execution because the inserted test code will be inserted as comment statements, as we explained in Section 4, and will be ignored by the compiler. These inserted test codes will be instrumented in the testing house for ensuring high-quality error-free codes, and the final output will be error-free user codes without test codes.

The reason behind the differences between the size overheads when measuring by number of lines or by bytes is that the number of characters in each line will affect the size in bytes, but will not affect the number of lines. Based on our results, the range of size overheads varies based on the behavior of the insertion statements.

Similarly, in Figure 23 the lowest testing time is for the benchmarks that have not included any parallel constructs and have fewer OpenACC data clauses, while the highest testing time is for the benchmark that has a big number of OpenACC parallel constructs. The average testing time for the OpenACC-related tested benchmark is 97 milliseconds.

As we discussed before, our testing tool has the ability to detect errors in OpenACC-related applications by using our hybrid testing techniques. This will help to increase reliability and ensure error-free codes to have high-quality systems. One of our contributions is to provide new techniques for detecting

$$SizeOverhead = \frac{Size\ with\ inserted\ test\ code - Size\ without\ inserted\ test\ code}{Size\ without\ inserted\ test\ code}$$

errors in OpenACC-related programs by using hybrid testing techniques.

We used OpenACC to build our testing tool, which makes it portable and hardware architecture-independent and can work with any type of GPU accelerator, hardware, platform, or operating system. ACC_TEST is also compatible with various compilers and easy to maintain with less effort because of the high maintainability of OpenACC. In addition, ACC_TEST is reliable because of the insertion techniques that avoid centralized control and the single point of failure problems. Additionally, by using our insertion techniques, we increase performance by distributing our testing tasks and avoiding centralized controlled testing. Finally, ACC_TEST will help to increase reliability and produce high-quality systems without errors. To the best of our knowledge, there is no testing tool designed to target OpenACC, and we are the first to propose and implement such a testing tool.

VI. CONCLUSION AND FUTURE WORK

Despite the fact that there are several testing tools that targeting parallel applications built by using programming models, there is more effort needed for high-level GPU-related programming models. Although OpenACC has been widely used in the past few years, there is no testing tool made especially to target OpenACC.

One of the main features of OpenACC is that it uses high-level directives without considering details to parallelize the existing sequential codes. This feature helps in attracting more non-computer science specialists to use OpenACC to parallelize their systems. As a result, the possibility of misusing OpenACC directives and clauses is higher and can lead to several runtime errors when the programmers try to parallelize their applications.

Our main contribution is that we provide new techniques for detecting errors in OpenACC-related programs by using hybrid testing techniques. We have implemented these techniques in ACC_TEST to detect runtime errors for OpenACC-based systems. Our solution integrated static and dynamic testing techniques to build ACC_TEST and allowed us to benefit from these techniques advantages, which reduced overheads and enhanced system execution time. In addition, our tool is a parallel testing tool that detects runtime errors by creating testing threads based on the number of application threads. ACC_TEST also is a platform-independent testing tool that can work with any heterogeneous architecture, which will increase ACC_TEST's portability. We have implemented our solution and evaluated its ability to detect OpenACC runtime errors.

Our testing tool has successfully detected errors that occurred in OpenACC applications by using our hybrid testing techniques. This will help to increase reliability and ensure error-free codes to have high-quality systems. We also achieved the error coverage with acceptable execution overhead, and it will be used only in the testing house and will not affect the delivered user applications. Finally, to the best of our knowledge, there is no parallel testing tool built to test applications programmed by using the single-programming model OpenACC. In our future work, we will enhance the ability of ACC_TEST to cover errors in the dual-programming models MPI + OpenACC.

ACKNOWLEDGMENT

This project was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under grant No. (RG-9-611-40). The authors, therefore, gratefully acknowledge the DSR technical and financial support.

REFERENCES

[1] The U.S. Department of Energy's Oak Ridge National Laboratory. (2018). *Summit*. [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[2] S. Chandrasekaran and G. Juckeland, *OpenACC for Programmers: Concepts and Strategies*. Reading, MA, USA: Addison-Wesley, 2017.

[3] *The OpenACC Application Programming Interface Version 3.0*, OpenACC Standards Org, 2019. [Online]. Available: https://www.openacc.org/specification

[4] 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, doi: 10.25046/aj040227.

[5] 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, doi: 10.14569/IJACSA.2019.0100448.

[6] A. M. Alghamdi and F. E. Eassa, "OpenACC errors classification and static detection techniques," *IEEE Access*, vol. 7, pp. 113235–113253, 2019, doi: 10.1109/ACCESS.2019.2935498.

[7] A. M. Alghamdi and F. E. Eassa, "Proposed architecture for a parallel hybrid-testing tool for a dual-programming model," *IJCSNS Int. J. Comput. Sci. Netw. Secur.*, vol. 19, no. 3, pp. 54–61, 2019.

[8] *The OpenACC TM Application Programming Interface*, OpenACC Standards, 2013.

[9] OpenACC-standard.org. OpenACC Organization. (2017). *About OpenACC*. [Online]. Available: https://www.openacc.org/about

[10] *OpenACC Programming and Best Practices Guide*, OpenACC Organization, 2015.

[11] M. McCorkle. ORNL Launches Summit Supercomputer. The U.S. Department of Energy's Oak Ridge National Laboratory, 2018. [Online]. Available: https://www.ornl.gov/news/ornl-launches-summit-supercomputer

[12] 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.

[13] Message Passing Interface Forum. (2017). *MPI Forum*. [Online]. Available: http://mpi-forum.org/docs/

[14] B. Barney. OpenMP. Lawrence Livermore National Laboratory, 2018. [Online]. Available: https://computing.llnl.gov/tutorials/openMP/#Introduction

[15] NVIDIA Corporation. (2015). *About CUDA*. [Online]. Available: https://developer.nvidia.com/about-cuda

[16] Khronos Group. (2017). *About OpenCL*. [Online]. Available: https://www.khronos.org/opencl/

[17] J. F. Münchhalfen, T. Hilbrich, J. Protze, C. Terboven, and M. S. Müller, *Classification of Common Errors in OpenMP Applications* (Lecture Notes in Computer Science: Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 8766. Cham, Switzerland: Springer, 2014, pp. 58–72.

[18] 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, Sep. 2017, doi: 10.1145/3095075.

[19] 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, doi: 10.1145/2966884.2966906.

[20] 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, pp. 527–563.

[21] The Open MPI Organization. (2018). *Open MPI: Open Source High Performance Computing*. [Online]. Available: https://www.open-mpi.org/

[22] 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, Nov. 2014, doi: 10.1177/1094342014552204.
[23] 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, doi: 10.1109/ICPP.2013.63.
[24] 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.
[25] 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, Aug. 2015, doi: 10.1145/2813885.2737962.
[26] 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, doi: 10.1007/978-3-642-34188-5_18.
[27] *PCAST—PGI Compiler Assisted Software Testing*. Accessed: Jan. 2, 2020. [Online]. Available: https://www.pgroup.com/resources/pcast.htm
[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, doi: 10.1145/1735688.1735702.
[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)*, May 2012, pp. 1–10.
[30] EPCC. EPCC OpenACC Benchmark Suite. The University of Edinburgh, 2013. [Online]. Available: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite
[31] 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, doi: 10.13140/RG.2.2.23914.41921.
[32] 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)*, 2017, pp. 246–250, doi: 10.1109/PDP.2017.32.

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 a joint supervision with the University of Colorado, USA, in 1989. He is currently a Full Professor with the Department of Computer Science, 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.

AHMED MOHAMMED ALGHAMDI received the B.Sc. degree in computer science and the first M.Sc. degree in business administration from King Abdulaziz University, Jeddah, Saudi Arabia, in 2005 and 2010, respectively, the second master's degree in Internet computing and network security from Loughborough University, U.K., in 2013, and the Ph.D. degree in computer science from King Abdulaziz University. He is an Assistant Professor with the Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Saudi Arabia. He has also over 11 years of working experience before attending the academic carrier. His research interests include high-performance computing, big data, distributed systems, programming models, software engineering, and software testing.

SEIF HARIDI was also the Chief Scientific Advisor of RISE SICS, until December 2019. He is a Chair Professor of computer systems specialized in parallel and distributed computing and the Head of the Distributed Computing Group, KTH Royal Institute of Technology, Stockholm, Sweden. He led a European research program on cloud computing and big data with EIT-Digital, from 2010 to 2013, and is a Cofounder of a number of start-ups in the areas of distributed and cloud computing, including Hive Streaming and Logical Clocks. He is a Codesigner of SICStus Prolog, the most well-known logic programming system, and the Mozart Programming System, a high-quality open-source development platform based on the Oz multiparadigm programming language. His research is focused on the combination of systems research and theory in the areas of programming systems and distributed computing.

MAHER ALI KHEMAKHEM received the B.Sc. degree in physics from the University of Tunis, Tunisia, in 1982, the M.Sc. and Ph.D. degrees in digital electronics and computer science from the University of Paris 11, Orsay, France, in 1984 and 1987, respectively, and the Habilitation (HDR) degree in computer science from the University of Sfax, Tunisia, in 2008. He is currently a Full Professor with the Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. His research interests include distributed systems, performance analysis, network security, and pattern recognition.

ABDULLAH S. AL-MALAISE AL-GHAMDI received the B.Sc. degree in computer science from the University of Southern Mississippi, USA, in 1990, the M.Sc. degree in management information systems from the University of Illinois at Springfield, IL, USA, in 1992, and the Ph.D. degree in computer science from George Washington University, USA, in 2003. He is currently a Full Professor with the Department of Information Systems, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. His research interests include collaborative software, distributed systems, conflict measurements, workflow, information systems, and artificial intelligence.

EESA A. ALSOLAMI received the bachelor's degree in computer science from King Abdulaziz University, in 2002, and the M.Sc. degree in IT and the Ph.D. degree from the Queensland University of Technology, Australia, in 2008 and 2012, respectively. He is an Associate Professor of computer science and engineering with the University of Jeddah, where he is currently the Dean of admission and registration. His research project involves feature selection techniques for continuous biometric authentication.