International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-9 Issue-2, December, 2019

Race Condition Detection Algorithms

B. Sai Manvitha Reddy, A. Hari Kishore, P. V. S. Krishna Manmayi, Mahadev A. Gawas

Abstract: ***A data race is similar to any other bugs in software application. Data race will result in the execution of the program unpredictable. There are 46 documented races in Linux kernel. OpenMP is an Application programming interface for shared programming model. It is a construct based model which works on fork join parallelism. OpenMP achieved node level parallelism and can manage data in single instruction multiple data and single program multiple data parallelism by executing different constructs like work sharing and parallel constructs. In any shared programming model, variables are shared by multiple threads in the program to execute different tasks by different threads. OpenMP is used to achieve parallelism by creating shared variable environment but there are chances to have data races in OpenMP programs. In this paper we discuss different algorithms to detect data races in OpenMP programs.***

Key words: OpenMP , data race detection ,OMPT, shared programming model.

I. INTRODUCTION

In a multiprocessor if each and every processor has equal amounts of time to access a common global address space which is being shared among all the different multi processors then such kind of architecture is referred as symmetric multiprocessor. Shared instructions multiple data parallelism can be implemented by a shared memory model in OpenMP. In a heap of processes that are trying to use the resources, OpenMP uses multiple threads to execute the parallel processes where the multiple threads communicate through the shared address space. OpenMP is a shared memory programming model in which a task is implemented by a group of threads to achieve parallelism. OpenMP is a directive or construct based model where the constructs in OpenMP falls into four major categories, which are parallel, task, synchronization and work sharing constructs. OpenMP has several important features like the tasks will be executed in a shared address space and the variables will be shared unintendedly to achieve a disciplined access of the data.
Memory access events like read or write will have some notable changes in memory content and when the changes are not properly recorded then it will be considered as violation of concurrency control rule.

Revised Manuscript Received on December 30, 2019.
* Correspondence Author

B. Sai Manvitha Reddy, Pursuing Under Graduation Stream of Computer Science Engineering, Vellore Institute of Technology, Tamil Nadu, India.
A. Hari Kishore, Pursuing Under Graduation Stream of Computer Science Engineering, Vellore Institute of Technology, Tamil Nadu, India.
P.V. S. Krishna Manmayi, Pursuing Under Graduation Stream of Computer Science Engineering, Vellore Institute of Technology, Tamil Nadu, India.
Dr. Mahadev A. Gawas, Associate Professor, Department of Computer Science and Engineering, VIT Vellore India.

When two memory events try to access the same memory location or same variable to change or update it without proper synchronization that condition is referred as data race. Though there are multiple accesses to a variable one must ensure that they are not conflicting as the resulting situation will lead to a data race. The output of a program with multiple accesses of the data will be unpredictable as the write access will conflict the read access on the same variable. These conflicts can be avoided by paying close attention to the shared environment. One of the famous debugger to know the data conflicts is to add print statements in the code and this is not feasible when we have large code and especially in the parallel programs this cannot be identified. As a result the need of tools to detect the data race is growing. A data race detector should have the ability to understand the program and the all the memory access events on a variable. The data race detector should be able to identify difference between the mutual exclusion and concurrency, as data race occurs when there are two different memory access events on the same variable. When two events exist in a relation known as happens-before, the order of the execution is defined. Even though after establishing a happens-before relation between two events and still they are not in order then they are concurrent. SPMD (Single Program Multiple Data) parallelism keeps on being one of the most well known parallel execution models being used today, as exemplified by OpenMP for multicore frameworks and CUDA and OpenCL for quickening agent frameworks. The fundamental thought behind the SPMD model is that all legitimate processors (specialist strings) execute a similar program, with successive code executed repetitively and parallel code (work sharing builds, obstructions, and so on.) executed helpfully. In this paper, we center on OpenMP as a model of SPMD parallelism. The OpenMP parallel develop shows the production of a fixed number of parallel laborer strings to execute a SPMD parallel area. The number of strings can be indicated in the code, or in a domain variable (OMP_NUM_THREADS), or by means of a runtime work, set_omp_num_threads() that is called before the parallel locale begins execution. The OpenMP obstruction build indicates a boundary activity among all strings in the present parallel area. Every powerful case of a similar boundary activity must be experienced by all strings, e.g., it isn't allowed for a hindrance in a then-condition of a if proclamation executed by (state) string 0 to be coordinated with a hindrance in an else-statement of the equivalent if proclamation executed by string 1. For build demonstrates that the quickly following circle can be parallelized and executed in a work-sharing mode by every one of the strings in the parallel SPMD area. A certain obstruction is performed following a for circle, while the nowait statement handicaps this certain obstruction. Further, a hindrance isn't permitted to be utilized inside a for circle. When the schedule (kind, chunk_size) proviso is connected to a for develop,

its parallel emphases are gathered into clusters of piece size emphases, which are planned on the specialist strings as per the arrangement determined by kind. In this paper, we confine our regard for OpenMP parallel circles with kind = dynamic and piece size = 1, which suggests that every cycle can be executed by any string in the parallel district.
In this survey we will compare the different techniques in three aspects. Accuracy,Firmness and ease of use.If a techniques is completely accurate then it is said to be complete tool. A complete tool will never give false positives. The lesser number of false positives we have, the more accurate the tool. By firmness we mean the level of promise will be given by the technique.We may say that the tool is firm when we have some unimportant situations which will give more false negatives. Ease of use generally means how difficult or easy the tool is to convert it into a development process. Since the research tools are often known for not scaling to larger. In our paper we will also consider scaling as a factor.

II. STATIC ANALYSIS

There are two approaches for dealing with any data race. One is static and the other is dynamic. Without executing the set of instructions or a program code will be analyzed using dependency graphs and etc., to write the lines of codes serially in order to examine them, this is known to be a static approach. There are many race detection tools which follow static approach. Archer is one of the popular static data race detectors. ARCHER maintains a list of two separate code blocks one is race free and the other having dependencies. Since dependencies may cause a data race the section of the code having the dependencies will be later examined by an LLVM inbuilt run time library to eliminate the dependency as a result race will be eliminated. ARCHER performs dependency analysis on the tasks of an OpenMP program. This is done by an existing LLVM, Clang suite tool Polly. The input of an ARCHER tool will be an OpenMP program and output will be the intermediate representation of LLVM by Clang front end. LLVM passes will be analyzed by a LLVM pass analyzer and LLVM IR will be analyzed to identify the code regions that are race free and can be executed sequentially.
The information about race free and possibly dependency free regions, gathered by the dependence examination, are taken care of in an once-over in wording of line numbers in the source code. The summaries containing the line amounts of race free areas are viewed as blacklists since every one of the heaps/stores recorded can be neglected during the dynamic examination performed by Thread Sanitizer. Right when the static assessment passes are done, the Thread Sanitizer instrumentation pass instruments the IR code to insert the limits required for getting data races at runtime. Our changed Thread Sanitizer instrumentation pass acknowledges the blacklists as its commitment to keep away from instrumenting the sans race regions as perceived by Polly.
Though the static analysis has advantages it has lot more disadvantages. While the program is getting executed the tasks which are not dependent may become dependent depending upon the constructs of the OpenMP and then the static analysis fails.
When the data races are detected at the compile time then the approach is static but when the races are detected at run time that is dynamic. We prefer dynamic race detectors over static race detectors as the number of false positives will be more in the static approach. There are two different approaches to detect the data races dynamically that are post-mortem or on-the-fly analysis. Post mortem analysis will check the data races when the execution of the program terminates whereas, on the fly race detectors will look after the data races during the execution of the program itself.
In static approach the code is being analyzed to detect unserialized access to the shared data but this approach has some disadvantages. When a data or shared variable is allocated dynamically on the heap then static analysis could not distinguish the data allocated on heap, as a result number of false positives will be increased

III. ON -THE -FLY ANALYSIS

On-the-fly race disclosure techniques rely upon program assessment. Thus, on-the-fly assessments work at run time, visit simply possible ways, and have exact points of view on the estimations of shared data and of other resource state. These techniques does not rely on assumptions because we analyze the code when during actually running it. Regardless, due to their dynamic nature, they power a staggering computational overhead, making it time consuming to run tests and tremendous on tasks that have serious arranging essentials. Hence when compared to the static approaches dynamic approaches require lot of computational power and cost high. The term high overhead infers that, while, on a fundamental level, on-the-fly mechanical assemblies can figure correct information, according to the dynamic approach.
Basically they are compelled to what can be enlisted capably both truly. Moreover, it is problematic or for sure, even hard to move race conditions by on-the-fly frameworks, as a result of the non-determinism displayed by schedulers. Besides, their reliance on instrumentation normally hinders their usage on low-level code, for instance, OS pieces, contraption drivers and complex embedded systems. Finally, on-the-fly contraptions can find bungles just on executed ways, which depend upon commitment to the system. This not simply makes dynamic examination irksome yet also sometimes unfathomable. Along these lines, it is appealing to have an area segment that can find races on a particular commitment with a single program execution, i.e., have the Single Input, Single Execution (SISE) property.
Everything considered, the SISE property can be manhandled for ventures that have inside non-determinism. Thus, the absolute preliminary of such system is generally not possible. Tragically, the amount of potential ways can grow exponentially with the size of code. This infers, before long, testing can simply practice a little part of each pragmatic way, leaving huge systems with a development of botches that could take significant lots of execution to appear. In specific structures it is unmistakably increasingly abhorrent, i.e., in a working structure some code may never run. Most of such code lives in contraption drivers, furthermore, only a little division of these drivers can be attempted at a customary site, since there are normally few presented devices.

IV. DYNAMIC APPROACH

Multithreaded programs have high chances of errors and data races if the mutual exclusion disciplines of a shared variable by multiple threads are not well structured. The shared memory location will have to be analyzed during the execution of the program and every shared memory reference will be monitored to understand a dependency. In dynamic approach the data race will be detected at the run time. There are several approaches in the dynamic data race detection and also different tools for race detection like cilkscreen for cilk programs, happens before analysis and Eraser etc. Two events can have a happens-before relation if there exists a synchronization between the events.

Definition 1. Let €i and €j be two events (e.g., a read, write, or synchronization operation) in a concurrent program. Let → denote the happens-before relation between two events. €i → €j if: i) €i and €j occur in the same thread and €i precedes €j in program order, or ii) there exists a directed synchronization from €i to €j , or iii) there exists an event k such that €i → €k and €k → €j (transitivity) .In happens before relations events that occur before will have less time stamp than the events that occur later. The events having happens-before relation can be established with transitivity. The main disadvantage of happens before analysis is, In happens before analysis one has to note all the memory access events that occurs on a single variable.

In all the existing data race detection algorithm, the algorithm works on thread level scheme to find data concurrency or data races. In such approaches when the memory access events try to access a shared variable then the events are mapped to the same threads but they can be considered as concurrent events then there are chances of missing a race, In other cases for example if the task that can lead to a data race are running on different number of threads then also there are chances to miss a data race so a data race detector should have the capability of understanding the construct of OPENMP and shared variables in the program while the program is executing.

ROMP supports, on-the-fly race detection in parallel program execution. Many data race detection tools employee hybrid approach or a hybrid algorithm that combines two or more approaches to detect a data race and the approch will be choosen based on data or switches the algorithm while the algorithm proceeds,like happens before ordering and lockset analysis for data race detection.

On the fly analysis is something which analyses the code during the execution of the program. On the fly analysis uses the standard happens-before relation. In ROMP happens before relation is not used instead happens before serial relation is considered. For example if the events in the history , current and future are €' , € and € '' then if there exists a happens before relation between current and history events and happens before relation between history and future events and if € || €'' then there exists happens before relation r between current and history events but not happens before serially.

For example there exists two concurrent events that can be executed in parallel say a, a' and the access history of memory location (l) contains a', before executing a, any future access all that is concurrent to a is also concurrent to a' , if the events can be related by happen before relation then they can be transitive . But parallel events cannot be transitive. This is a contradiction. All sets analysis establishes a pseudo transitivity between parallel events which can miss a data race. But in ROMP, pseudo transitivity of parallel events is not considered, maintaining access history of memory location is an important aspect of ROMP. By maintaining an access history of events races cannot be missed. For example, if two events a and a' are concurrent or parallelly executed by the tasks t and t', and if t' is still executing a' without having a to the access history would miss a race and is a false positive .If t' performs write operation that would conflict a.

Before having an access history there should be a note on mutual exclusion entries that should be held on a memory location. A data race will be found by considering four factors. They are 1.memory location being accessed[1] 2. An access record of l which contains {memory access event, type of the event whether read or write and mutual exclusion entities on a memory location}[€, a, h]. Apart from these history will be maintained for each memory location .The access history can be pruned after successful revision whether they have any use in detecting race and can be decided whether it can be pruned or not because on basis of the access history, if we can find a data race our goal is completed. Multiple data races on a memory location are efficient and practically feasible. Data races are detected by pruning the access history.

That is for every current memory access record [€, a, h] if their concurrent record [€' , a' ,h'] exists in the access history of l then there are high chances of a data race , if access event € is parallel to €' , lock sets of each record h and h' does not have any intersection either one of the events are write then it is a data race. If both of the events are write and the event parallel to the previous event is read and h is super set of h' there exists happens before relation between € and €'. Then the record [€' , a' ,h'] can be pruned from the history[l] .In another case if both the events are read and other parallel events is write then h is superset or equal to h', then no change in the access history. In none of the above cases, current record [€, a, h] can be added to the access history[l].

Every parallel program can be expressed in the form of directed acyclic graph to represent the dependencies. OpenMP works on fork join parallelism. OpenMP has several constructs which is a thread based model can not explain the logical concurrency between the events. Open task graph can be visualized and analyzed using Intel flow graph analyzer. In OpenMP task graph every vertex is labelled with certain procedure. In ROMP, OpenMP task graphs are labelled with a methodical approach. A task dependency graph gives the relation between different regions of the code and in the OpenMP programs as the code executes and different constructs are executing the dependency between each thread should be understood for detecting data race.

#pragma omp task depend (type: list items) is the construct used to establish the dependencies between different list items of different type. This can be understood by the following example:

```
void task_dependency_example
{
int a, b, c;
#pragma omp task depend(out : a, b)
a=b=1;
#pragma omp task depend(int : a) depend (out :c)
c=a;
#pragma omp task depend(in : b)
depend (out : d)
d=b;
#pragma omp task depend(in : c, d)
Computation( c, d);
}
```

For the above example the task graph will be:

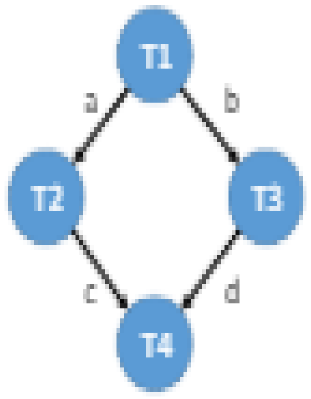

Fig1.task graph of a sample openmp program.

Dynamic data race deectors (including ROMP tool) cannot be able to support the race detection for OpenMP Single Instruction Multiple Data develops with the present compilers.

A compiler either or not replaces scaler version of code with vector code,Scalar code which is marked with an OpenMP Single Instruction Multiple Data directive. Without the first scalar code as a guide a dynamic race detector cannot decide whether the vectorizer changes the program semantics by overlooking an data race related with data reliance.

V. ROMP TASK LABELLING

To understand the synchronization between the tasks in the task graph of the OpenMP model each task will be assigned with a label. Each order of the event can be understood by the access history of the location. The serial execution of the multiple parallel programs is feasible keeping logical concurrency into consideration. For each shared variable in the shared address space the access history is maintained. The label of a task depends on the history ,therefore when a new task is created the labelling depends upon the nesting level of the OpenMP tasks. Task label segments have different fields and will be updated accordingly.They are {offset,span,iteration id,taskwait count,task create count,loop count, phase , task waited, task group info ,segment type}. The offset will be the relative id of the worker thread in a team of threads forked when parallel construct is invoked. Span will be the total number of threads forked other than the master thread for each fork join loop. Iteration id will be the relative id of the work share construct if any exixts in the program. Task wait count will be the number of taskwait encountered in the current task construct. Task create count will be the enumber of explicit tasks . loop count will be the number of work share loops ended by the current task in execution. Phase will the number of time the task under execution is entering or leaving the critical .task waited will be a boolean which will be set to true if the task is waited by the parent.

Task group info will the information of the orderings of the tasks. Segment type will be the type of the task , it can explicit ,implicit and logical. For an implicit task offset and span will be 0 and team size respectively. For an explicit task offset and span will be 0 and 1 respectively. Each current label will be the nested label of its parents . That is for every current task its label will be appended to the parent tasks label. There are a notable advantages of this labelling scheme. Two queries can be compared by comparing their labels. Multiple queries can be executed in parallel.

VI. CONCLUSION

Static or dynamic investigations can improve each other by giving data that would somehow or another be inaccessible. Performing initial one investigation, at that point the other (and maybe repeating) is more dominant than performing it is possible that one in separation. Then again, various examinations can gather various assortments of data for which they are most appropriate. This notable collaboration has been and keeps on being misused by specialists and experts the same. As one straightforward model, profile-coordinated arrangement [1] uses indications about every now and again executed methodology or code ways, or usually watched qualities or types, to change code. The change is significance saving, and it improves execution under the watched conditions yet may debase it in divergent conditions (the right outcomes will at present be processed, just devouring additional time, memory, or power). As another model, static examination can forestall the gathering of certain data by ensuring that gathering a littler measure of data is satisfactory; this makes dynamic examination increasingly productive or exact.

ROMP is a hybrid data race detector in OpenMP programs which is dependent on several LLVM run time libraries and several binaries. ROMP has more performance when compared to other dynamic data race detectors and also static analyzers. ROMP follows the same algorithm as ARCHER but the precision is increased by taking the logical concurrency in to consideration. The numbers of false positives are decreased. ROMP has almost the capability of understanding a code semantically for race detection.

REFERENCES

1. Yizi Gu, John Mellor-Crummey ‘’Dynamic Data Race Detection for OpenMP Programs’’.
2. OpenMP Language Committee, “OpenMPApplicationProgrammingInterface,version4.5,”http://www. openmp.org/wp-content/uploads/ openmp-4.5.pdf, November 2015.
3. https://contribute.llnl.gov/tutorials/openmp/(online)
4. ‘’An Efficient Algorithm for On-the-Fly Data Race Detection Using an Epoch-Based Technique’’.
5. R. H. B. Netzer and B. P. Miller, “What are race conditions?: some issues and formalizations,” ACM Letters on Programming Languages and Systems, vol. 1, no. 1, pp. 74–88, 1992.
6. Vineet Kahlon1 , Yu Yang2 , Sriram Sankaranarayanan1 , and Aarti Gupta1 .’’ Fast and Accurate Static Data-Race Detection for Concurrent Programs’’.
7. “Eraser: A Dynamic Data Race Detector for Multithreaded Programs’’
8. A Survey of Methods for Preventing Race Conditions by Nels E. Beckman; May 10, 2006
9. ‘’RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking’’ Yuan Yu, Tom Rodeheffer, Wei Chen.
10.] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, “Scalable and precise dynamic datarace detection for structured parallelism,” in Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI ’12. New York, NY, USA: ACM, 2012, pp. 531–542.
11. R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, “Efficient data race detection for async-finish parallelism,” in Proceedings of the First International Conference on Runtime Verification, ser. RV’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 368–383. [Online]. Available: http://dl.acm.org/citation.cfm?id=1939399.1939430

12. A Review of Race Detection Mechanisms, Aoun Raza.
13. Flanagan, C., Freund, S.N.: Type-Based Race Detection for Java. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, Vancouver, British Columbia, Canada, pp. 219–232 (2000).
14. Helmbold, D.P., McDowell, C.E.: A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35 (1994).
15. Static and dynamic analysis: synergy and duality. Michael D. Ernst.

AUTHORS PROFILE

B.Sai Manvitha Reddy,She is pursuing under graduation in the stream of Computer science and Engineering at Vellore Institute of Technology, Tamil Nadu, India.

A.Hari Kishore, He is pursuing under graduation in the stream of Computer science and Engineering at Vellore Institute of Technology, Tamil Nadu, India.

P.V.S.Krishna Manmayi , She is pursuing under graduation in the stream of Computer science and Engineering at Vellore Institute of Technology, Tamil Nadu, India.

Dr. Mahadev A. Gawas, is currently working as an Associate Professor in the Department of Computer Science and Engineering, VIT Vellore India. He completed his Ph.D. from the Department of Computer Science & Information Systems, BITS Pilani, India. He has authored several research papers in refereed international conferences and journals. His research interests include wireless communications, multimedia communications, cross-layer architecture, vehicular ad hoc networks. He has received a number of awards, such as the Asia Pacific Advanced Network Fellowship, and Microsoft Research Travel Grant fellowship.