LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-599993

# Symbolic Analysis of Concurrency Errors in OpenMP Programs

H. Ma, L. Wang, C. Liao, D. Quinlan, Z. Yang

November 9, 2012

Symbolic Analysis of Concurrency Errors in OpenMP Programs
Boston, MA, United States
May 20, 2013 through May 24, 2013

## Disclaimer

# Symbolic Analysis of Concurrency Errors in OpenMP Programs

Hongyi Ma and Liqiang Wang
Department of Computer Science
University of Wyoming
{hma3,lwang7}@uwyo.edu

Chunhua Liao and Daniel Quinlan
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
{liao6, dquinlan}@llnl.gov

Zijiang Yang
Department of Computer Science
Western Michigan University
{zijiang.yang}@wmich.edu

***Abstract*—As one of the popular parallel programming models, OpenMP has been widely used in scientific applications in recent years to facilitate shared-memory parallelism. The increasing availability of multicore devices has permitted more and more sequential programs to be parallelized using OpenMP. Unfortunately, it is a daunting task to develop correct OpenMP programs. Concurrency errors, such as data races and deadlocks, are tricky to detect using traditional testing techniques. This paper presents an OpenMP Analysis Toolkit (OAT) to detect data races and deadlocks using SMT-solver based symbolic analysis, which can approximately simulate the real execution of an OpenMP program. Hence, our symbolic analysis is more accurate than traditional static analysis, and more efficient and scalable than runtime analysis tools. We conducted experiments on real-world OpenMP benchmarks and university student homework assignments by comparing our OAT tool with two commercial runtime tools for checking multithreaded programs (Intel Thread Checker and Sun Thread Analyzer). Our experiments show that our symbolic analysis tool is more efficient and scalable than the two commercial tools. Our tool OAT is as accurate as Sun Thread Analyzer, and both are more precise than Intel Thread Checker for checking concurrency errors in OpenMP programs.**

***Keywords*-OpenMP; Data Race; Deadlock; Symbolic Analysis; Static Analysis; Dynamic Analysis; SMT Solver;**

## I. Introduction

OpenMP has been widely used in scientific applications to facilitate shared-memory parallelism. In recent years, it has become even more popular due to the ubiquity of multicore computers. OpenMP is a portable parallel programming model used to create parallel C/C++ and Fortran multithreaded programs on shared-memory computing platform. It has been implemented in major compilers such as GCC, MS Visual Studio, and Intel Compiler. In OpenMP, parallelism is explicitly specified by programmers using directives inserted into existing programs. The execution of OpenMP programs is based on OpenMP runtime libraries, most of which rely on the Pthreads API to create and manipulate threads.

Developing OpenMP programs is prone to concurrency errors. A common concurrency error in OpenMP programs is data race. A data race occurs when two or more threads perform conflicting data accesses (i.e., accesses to the same variables and at least one access is a write) without using an explicit mechanism to prevent the accesses from happening simultaneously. Consider the Example-1 in Figure 1, the parallelization is indicated by OpenMP directive `#pragma omp parallel for`, which distributes all loop iterations nearly equally to each OpenMP thread. By default, all variables in a parallel region are shared except for the loop iteration variable (*i.e.*, `i` in the current example). Although `i` is a private variable, some `x[i]` may be read and written by two different threads simultaneously, which incurs a data race. Another more tricky data race is shown in the Example-2 of Figure 1, where `shared` and `private` clauses define shared and private variables for the parallel section, and `nowait` indicates that OpenMP threads do not synchronize at the end of the parallel loop, thus the thread finishing `for` iterations may execute the next assignment statement right away. In this example, there is no obvious data race if only considering the `for` loop. However, some threads that finish iterations early will execute `errors = dt[9]+1` while other threads may still simultaneously execute `for` worksharing region by reading and writing `d[i]`, which may cause a data race.

Compared to data races, deadlocks are less common in OpenMP. A deadlock in OpenMP is usually introduced by improper use of the `omp barrier` directive or the lock routines in OpenMP runtime library. The `omp barrier` directive forces a thread to wait at a barrier until other threads have reached the same barrier. The example in Figure 2 shows a deadlock scenario from [1]. By default, the two `#pragma omp section` are executed by two different threads. Since every `#pragma omp section` construct contains a `barrier` directive in the function call `print_results()`, each thread would execute a different barrier directive. Thus, each thread would wait for the other to reach its own barrier, which will never happen. Hence, a deadlock occurs.

Traditional data race and deadlock detections are usually either dynamic (*e.g.*[16], [25]) or static (*e.g.*[15]). Static analysis is able to consider all possible behaviors without actually executing a program. However, it produces false positives due to the fact that some aspects of the program's behavior, such as aliases and pointers, are impossible to

```
/* Example-1 */
#pragma omp parallel for
for(i = 1; i<1000; i++){
   x[i] = x [i]+ x[i-1];
}

/* Example-2 */
#pragma omp parallel shared(b) private(errors)
{
#pragma omp for nowait
for(i = 0; i < 10; i++)
    dt[i] = b + dt[i]*5;
errors =  dt[9] +1;
...
}
```

Figure 1. Examples of Race Condition In OpenMP Programs.

```
void print_results(float array[N], int section)
{
 #pragma omp critical {
 int tid =omp_get_thread_num();
 printf("The results are in section
          %d\n", section);
 for (i=0; i<N; i++)
        printf("%e  ",array[i]);
 } /*** end of critical ***/
 #pragma omp barrier
 printf("Thread %d done and synchronized.
          \n", tid);
}
#pragma omp sections  {
   #pragma omp section {
          print_results(c, 1);
   }
   #pragma omp section {
          print_results(c, 2);
   }
   /*** Use barrier for clean output ***/
   #pragma omp barrier
 }  /* end of parallel section */
```

Figure 2. Example of deadlock in OpenMP programs.

obtain precisely. Furthermore, static analysis usually cannot report witnesses in term of an execution leading to detected errors. Therefore, significant manual efforts are required to confirm each detected error. Dynamic analysis, on the other hand, can miss errors because not all possible program behaviors can be observed during executions. In addition, the approaches are not appropriate for large-scale applications since the overhead is usually more expensive.

In recent years, symbolic approaches have become a promising trend to error detections because of less false positives and false negatives. Symbolic execution [14], [11], [27], [3], [4] attempts to explore all program paths under symbolic values. By encoding the current execution into first order logic formula, predicative analysis [28], [24] is able to predict errors accurately even under correct executions. Conconlic approaches [5] perform symbolic analysis during dynamic executions so paths can be explored systematically without redundancy. The common theme among these symbolic approaches is encoding of program models, followed by SMT-based solving. In order to achieve scalability, encoding must be carefully designed and optimized based on domain knowledge. Satisfiability Modulo Theories (SMT) problem is a decision problem for a logical formulas with respect to combinations of special background constraints in first-order logic and verify the statements whether satisfies with established constraints.

In this paper, we propose a symbolic approach to detect race conditions and deadlocks in OpenMP programs. Our tool, called OpenMP Analysis Toolkit (OAT), is able to automatically detect data races and deadlocks accurately and efficiently. To the best of our knowledge, OAT is the first tool using symbolic analysis to check OpenMP programs for concurrency errors. Specifically, our paper makes the following contributions:

- We present a novel encoding algorithm specialized for OpenMP programs. Although there exist algorithms that encode various systems, including parallel programs, none of them can be directly applied to OpenMP programs. In particular, we encode every parallel code region of an OpenMP program into formulae suitable for off-the-shelf SMT-solvers such as Yices [2]. By interpreting the solution we able to produce a feasible execution that reproduces the error.
- Our approach is fully automated and does not require any manual intervention or source code annotation by the programmer. We build our tool using the ROSE compiler infrastructure [23] to parse OpenMP programs, translate them to Yices formulae, then instrument the source code with fault injection technique to confirm reported errors.
- Finally, our approach is scalable to analyze large-scale OpenMP code. Our experimental evaluation, presented in Section V, shows that our approach is accurate and efficient at detecting race conditions and deadlocks.

The remainder of this paper is organized as follows: Section II describes the encoding algorithms. Section III introduces the approaches to detect data races and deadlocks. Section IV introduces the implementation of our tool. Section V presents our experimental evaluation of our tool OAT over a set of benchmarks and real-world applications. Section VI reviews the latest literature and discusses how the tool OAT differs from them. Section VII concludes this paper and provides direction for the future work.

## II. Symbolic Encoding Algorithms

Figure 3 gives a simplified grammar of OpenMP C programs. The notation "$var$," indicates a list of variables, "[$term$]" means an optional term, and "$type*$" and "$type[]$" denotes pointer and array types, respectively. Expression $exp$ represents usual C expressions including assignments. The notation $newline$ indicates that the following block or statement should start with a new line.

| | | |
|---|---|---|
| *program* | ::= | *var_decl* \| *fun_decl* |
| *var_decl* | ::= | *type var*[= *exp*] |
| *fun_decl* | ::= | *type fun_name*([<*type var*>, ]) *block* |
| *block* | ::= | {*stmt*;} |
| *stmt* | ::= | *openmp_construct*<br>\| `if` *exp block* [`else` *block*]<br>\| `for` (*exp*;*exp*;*exp*) *block*<br>\| *block*<br>\| *var_decl*<br>\| *lock_routines*<br>\| *fun_name* ([ *exp*,]) |
| *openmp_construct* | ::= | *par_construct*<br>\| *workshare_construct*<br>\| *sync_construct* |
| *par_construct* | ::= | `#pragma omp parallel` [*data_clause*]<br>*newline block* |
| *workshare_construct* | ::= | *for_construct* \| *section_construct*<br>\| *single_construct* |
| *for_construct* | ::= | `#pragma omp for` [*data_clause*]<br>*newline for_statement* |
| *section_construct* | ::= | `#pragma omp section`<br>*newline block* |
| *single_construct* | ::= | `#pragma omp single` [*data_clause*]<br>*newline block* |
| *sync_construct* | ::= | `#pragma omp barrier` \|<br>`critical` \| `atomic` \| `ordered` \|<br>`master` \| [*newline statement*] |
| *data_clause* | ::= | `private` (*var*,) \|<br>`firstprivate` (*var*,) \|<br>[`lastprivate` (*var*,)] \|<br>[`default (shared \| none)`] \|<br>[`shared` (*var*,)] \|<br>[`reduction`(*reduction_op*:*var*,)] |
| *lock_routines* | ::= | *omp_init_lock*(*lk_var*) \|<br>*omp_set_lock*(*lk_var*) \|<br>*omp_unset_lock*(*lk_var*) \|<br>*omp_test_lock*(*lk_var*) |
| *type* | ::= | `int` \| `double` \| `float` \| *type*$*$ \| *type*[] |

Figure 3. Simplified Grammar of OpenMP C Programs.

### A. Encoding OpenMP Constructs

The encoding of OpenMP programs mainly translates parallel constructs, including parallel regions, worksharing, synchronization directives, data environment clauses, as well as sequential statements within parallel constructs. We use the ROSE compiler to parse OpenMP C programs and generate abstract syntax trees (AST) , then traverse ASTs and translate each construct into SMT formulae.

**Static Single Assignment:** There are neither control flow nor variable scoping in first order logic formulae. Therefore the variables involved in multiple assignments in an OpenMP program must be differentiated. We use Static Single Assignment (SSA) form to track variable updating. In SSA, variable at each assignment is renamed by adding different subscripts. Figure 4 depicts how to encode basic statements into logical formulae. We assume that the subscript of every variable starts with 0. The formula $i.op = W$ denotes that the associated operation with $i$ is write. Without explicit indication, the default operation is read.

```
int k,i = 0;
int a[2] ={0,0};
a[0] = i*k;
i++;
```
$\rightarrow$
$k_0 = i_0 = 0 \wedge$
$a_0[0] = a_0[1] = 0 \wedge$
$a_1[0] = i_0 \times k_0$
$\wedge i_1 = i_0 + 1$

Figure 4. Encoding of OpenMP basic statement

With SSA the data flow can be explicitly encoded. However, problem may arise when a variable is defined in a branch, as we need to merge the definitions after the branch. As shown in Figure 5, variables $j$ and $k$ are assigned in one branch while $i$ is assigned in the other. In such case, we consider that $i$ is assigned in `then` branch as well and the assignment is to keep the value of $i$. That is, we are adding an assignment `i=i` in the branch where the value has not been assigned originally. It is the same for $j$ and $k$. By inserting assignments, the subscripts for an assigned variable become the same, so any read to the variable after the branch can refer to the same variable name.

```
if (i>0){
  j = i*10;
  k = j - i;
}
else{
  i = j+k;
}
```
$\rightarrow$
$(i_0 > 0$
$\wedge j_1 = i_0 \times 10$
$\wedge k_1 = j_1 - i_0$
$\wedge i_1 = i_0)$
$\vee (i_0 <= 0 \wedge$
$\wedge i_1 = j_0 + k_0$
$\wedge j_1 = j_0 \wedge k_1 = k_0)$

Figure 5. Encoding of OpenMP Branches

**OpenMP Parallel Region:** Data races and deadlocks can only happen in parallel regions, as OpenMP code outside of the parallel regions run serially. Parallel regions can include both iterative and non-iterative segments of OpenMP program code. Since the number of threads cannot be determined statically, in this paper, we assume that there are two threads in the OpenMP parallel execution for the efficiency of our analysis. In real execution, by default, the number of forked threads is determined by the number of processor or CPU scores. Although OpenMP provides functions to fork a given number of threads, however, which is rarely used. OpenMP usually runs in the SPMD (single program, multiple data) way, *i.e.*, every thread runs the same code but different dataset. Hence, two threads are usually enough to detect race conditions. Encoding OpenMP

programs using more threads will significantly affect the efficiency and scalability of analysis.

Figure 6 illustrates the encoding of OpenMP parallel constructs, where a special superscript is used to indicate the thread id. The expression $omp.par = T$ states that the current region is in parallel region. Suppose $v$ is a shared variable, then our approach has an approach to detect the read and write order to simulate execution order, that is the value of read equals to the value of latest write.

```
100 #pragam omp parallel
101 {v = v + 1;}
```

$\rightarrow$

$$
\begin{aligned}
&omp.parbegin = 100 \\
&\wedge omp.par = T \\
&\wedge v[v_1^{t_1}] = v[v_0^{t_1}] + 1 \\
&\wedge v[v_1^{t_2}] = v[v_0^{t_2}] + 1 \\
&\wedge v[v_0^{t_1}] = v[v_0^{t_1} - 1] \vee v[v_0^{t_2}] = v[v_0^{t_2} - 1] \\
&\wedge omp.parend = 101
\end{aligned}
$$

Figure 6. Encoding of OpenMP Parallel Constructs

**Worksharing Construct:** Within an OpenMP `for` loop and `section` regions, an access by one thread to a shared variable may conflict with accesses from other threads to the same variable. Individual shared variables can be handled in the same way as the aforementioned parallel regions. We need to handle array specially. Although different loop scheduling policies are allowed, we assume that the iterations of `omp for` in OpenMP will roughly be equally partitioned and distributed to all threads, for simplicity. Similarly, as before, we use a superscript to identify the accessing thread. `Single` is another worksharing directive, which indicates that the enclosed code is executed by only one thread during the execution of the parallel region. The examples are shown in Figures 7, 8, and 9.

For `omp for` construct, OAT first normalizes the loops, then uses SMT-solver to check the constraints rather than executes all the iterations in the loop. To handle multiple-dimension array, we translate it to be one-dimension array. For example, in Figure 8, there is a two-dimension array[1D][2D], each element array[i][j] is translated to be array[i*2D +j]. Instead of considering every array element individually, we generate constraints based on array index, which is one kind of abstraction for array.

```
100 #pragma omp section {
101   array[i+1]=array[i]+1;
102 }
```

$\rightarrow$

$$
\begin{aligned}
&omp.sectionbegin = 100 \wedge omp.par = T \\
&\wedge array[i^{t_1} + 1] = array[i^{t_1}] + 1 \\
&\wedge omp.sectionend = 102
\end{aligned}
$$

Figure 7. Encoding of OpenMP `section` Construct

**Data Clauses:** Data environment clauses are used to define

```
99 int array[1D][2D];
100 #pragma omp for
101 for(int i=lb;i<ub;i++) {
102   for(int j = lb; j < ub; j++)
103    array[i+1][j]=array[i][j]+1;
104 }
```

$\rightarrow$

$$
\begin{aligned}
&omp.forbegin = 100 \wedge omp.par = T \\
&\wedge i^{t_1} \in [lb, \lceil (ub - lb)/2 \rceil + lb] \wedge i^{t_2} \in [\lceil (ub - lb)/2 \rceil + 1, ub] \\
&\wedge array[(i^{t_1} + 1) * 2D + j^{t_1}] = array[i^{t_1} * 2D + j^{t_1}] + 1 \\
&\wedge array[(i^{t_2} + 1) * 2D + j^{t_2}] = array[i^{t_2} * 2D + j^{t_2}] + 1 \\
&\wedge omp.forend = 104
\end{aligned}
$$

Figure 8. Encoding of OpenMP `for` Construct

```
100 #pragma omp parallel
101 {
102 # pragma omp single
103   j = j + 1;
104 }
```

$\rightarrow$

$$
\begin{aligned}
&omp.par = T \wedge omp.parbegin = 100 \wedge omp.singlebegin = 102 \\
&\wedge j_1 = j_0 + 1 \wedge omp.singleend = 103 \\
&\wedge omp.parend = 104
\end{aligned}
$$

Figure 9. Encoding of OpenMP `single` Constructs.

the properties of data-sharing and other specific operations. The data environment clauses handled by OAT include `private`, `firstprivate`, `lastprivate`, `shared`, `default`, and `reduction`. For example, `reduction` specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region. We have extended ROSE to have an OpenMP normalization phase that makes all data-sharing attributes explicit so that the encoding process can easily distinguish between shared and private variables. As shown in Figure 10, properties are added to each variable, corresponding to the data clauses such as `shared` and `reduction`. The `default(none)` is analyzed and removed by the normalization phase of ROSE, so it is not encoded into SMT formulae. By default, variables are shared, so we do not need to encode `shared(n,x)` in SMT formulae.

**Synchronization Directives** are used to control threads. OAT handles the synchronization directives related to race conditions and deadlocks, including `master`, `critical`, `barrier`, `atomic`, and `ordered`. The synchronization directives manage the execution order of each thread. Given an OpenMP construct $blk$, we use $omp.master$, $omp.critical$, $omp.atomic$, $omp.ordered$ to indicate that $block$ is within a parallel region enforced by the synchronization directives, `master`, `critical`, `atomic`, and `ordered`, respectively. The beginning and ending of these synchronization directives determine what kinds of threads

```
100 int sum = 0;
101 #pragma omp parallel shared(n,x) {
102 #pragma omp for private(i) reduction(+:sum)
103 for(i = 0; i <n; i++)
104   sum = sum + x[i];
105 }
```

$\rightarrow$

$sum_0 = 0 \wedge omp.parbegin = 103 \wedge omp.par = T$
$\wedge omp.forbegin = 102 \wedge i.shared = F$
$\wedge i^{t_1} \in [0, \lceil n/2 \rceil] \wedge i^{t_2} \in [\lceil n/2 \rceil + 1, n] \wedge sum.redc = ADD$
$\wedge sum_1^{t_1} = sum_0^{t_1} + x_0[i^{t_1}]$
$\wedge sum_1^{t_2} = sum_0^{t_2} + x_0[i^{t_2}]$
$\wedge omp.forend = 104 \wedge omp.parend = 105$

Figure 10. Encoding of Data Clauses.

```
100 #pragma omp parallel
101 #pragma omp critical
102   block
103 #pragma omp master
104   block
105 #pragma omp atomic
106   block
```

$\rightarrow$

$omp.parbegin = 100 \wedge omp.par = T \wedge$
$omp.criticalbegin = 101 \wedge \mathcal{F}(block)$
$\wedge omp.criticalend = 102 \wedge omp.masterbegin = 103$
$\wedge \mathcal{F}(block) \wedge omp.masterend = 104 \wedge omp.barrier = T$
$\wedge omp.atomicbegin = 105 \wedge \mathcal{F}(block)$
$\wedge omp.atomicend = 106 \wedge omp.parend = 106$

Figure 11. Encoding of Synchronization Directives.

will execute the code in the $block$. Figure 11 shows an example of encoding synchronization directives, where $\mathcal{F}(block)$ indicates the entire formulae of $block$.

**Synchronization using** `omp lock` **and** `barrier`**:** The encoding of `lock` and `barrier` is similar to other synchronization directives. A thread is granted ownership of a lock when it becomes available.

```
100 omp_lock_t lockA;
101 omp_init_lock(&lockA);
102 #pragma omp parallel
103 {
104   omp_set_lock(&lockA);
105   x = x + 1;
106   omp_unset_lock(&lockA);
107   #pragma omp barrier
108 }
109 omp_destroy_lock(&lockA);
```

$\rightarrow$

$omp.par = T \wedge omp.parbegin = 102 \wedge lockA.begin = 104$
$omp.barrier.numSetLocks = 1 \wedge$
$\wedge x_1^{t_1} = x_0^{t_1} + 1 \wedge x_0.lock = T \wedge x_1.lock = T$
$\wedge omp.barrier.numUnsetLocks = 1$
$lockA.end = 106 \wedge omp.barrier = T$
$\wedge omp.parend = 107$

Figure 12. Encoding of `Lock`.

**Pointers, Aliases, and Function Calls:** We design a basic intraprocedural alias/pointer analysis to find out all aliases for each variable in ROSE. Thus we avoid encoding alias, which can improve both accuracy and efficiency of our analysis. In following example, variables `*p` and `*q` are alias of `a`, and `c` is alias of `b`. Hence we do not need to encode `*p`, `*q`, and `c`. For function calls within OpenMP constructs, we do a basic inline operation instead of employing more complex interprocedural pointer analysis, which is enough for our experiments on real-world OpenMP benchmarks of Section V.

```
int *p, *q;
int a = 0; p = &a; q = &a;
*p = *p + 2; *q = *q + 1;
int c;
int &b =c;
```

Figure 13 shows how to encode pointers, aliases, and function calls. In this example, there are two functions involved. According to the call graph, the scope of the function call is 1, and the scope of the function definition is 2. Thus, ${}^2array_0$ indicates the first array variable in the scope 2.

scope 2:
```
100 void print_results(float array[], int t,
      int N){
101   int i;
102   int *p = array;
103   for(i = 0; i < N; i++){
104     printf("%e %d", *p, t);
105     p++;
106   }
107 }
```

scope 1:
```
108 if(t==0)
109   print_result(array,t,N);
```

$\rightarrow$

$\neg(t_0 == 0) \vee (t_0 == 0) \wedge {}^2array_0 = {}^1array_0$
$\wedge {}^2t_0 = {}^1t_0 \wedge {}^2N_0 = {}^1N_0 \wedge fun.begin = 100$
$\wedge p.pointer = {}^2array_0 \wedge i_0 \in [0, N-1] \wedge printf({}^2array_0[i_k])$
$\wedge printf({}^2t_0) \wedge p.pointer = {}^2array_0[i_k + 1]$
$\wedge k \in [0, N-1] \wedge fun.end = 107$

Figure 13. Encoding of Pointer, Alias, and Function Call.

## III. Detecting Concurrency Errors

### A. Data Race

After encoding OpenMP constructs into SMT formulae, we detect data races using Yices [2]. Given an OpenMP parallel construct $c$, let $\mathcal{F}(c)$ denote the corresponding SMT formulae. Based on OpenMP memory model [8], we design symbolic execution for detecting data races. The

OpenMP memory model is based on weak ordering, which can prohibit overlapping a synchronization operating with any other shared memory operations of different threads, while synchronization operations of different threads are sequentially consistent. Thus, OpenMP memory model can guarantee the following ordering among synchronization operations and accesses to shared variables: $S \rightarrow W, S \rightarrow R, R \rightarrow S, W \rightarrow S, S \rightarrow S$, where $S$, $W$, $R$ denote synchronization, write, and read, respectively. Based on SSA notation, we can encode all ordering relationships between different threads.

According to OpenMP Programming Model, an OpenMP program is partitioned into segments that are a sequence of instructions ending with a synchronization instruction. In our symbolic analysis, we use an event $e$ to represent a write or a read instruction on a variable. Let $\pi(s) = \{e_1, ..., e_n\}$ be a concrete execution order for a segment $s$. Each variable has an SSA form. Then we define that the variable value read by an event is always the value written by the most recent write in $\pi(s)$. The SMT-solver checks whether there exists a variable $v$ in $\pi(s)$ that can cause a non-deterministic and unexpected results. A solution to these SMT constraints indicates that a race condition is revealed in feasible symbolic execution.

### B. Deadlock

Barrier synchronization is a common cause for deadlocks in OpenMP programs. The semantics of OpenMP requires that all threads involved in a parallel region execute the same `barrier` point; otherwise, a deadlock will occur. In addition, lock/unlock can also incur deadlocks. Figure 14 illustrates a deadlock occurring for unreleased lock variable $\&lock\_a$, where only one thread executes lock region without releasing lock variable. One technique implemented in OAT based on ROSE is called path feasibility [6], to determine whether a code block can be reachable by all threads.

```
100 int x=A, y=B; // assuming A > B
101 if(x> y){ // (A > B) evaluates to be true
101    omp_set_lock(&lock_a);
102   x =x + y; //x = A + B;
103   y = x -y; //y = A;
104   x = x -y; //x = B;
105 }
105 if(x > y){ //(B > A) evaluates to be false
106    omp_unset_lock(&lock_a );
107 } // no path feasibility in SMT solver
```

Figure 14. Example of Deadlock Detection.

After encoding OpenMP constructs into SMT formulae, we use Yices [2] to solve the formulae in order to detect deadlocks. OAT uses dataflow and control flow analyses to establish constraints for $\mathcal{F}(c)$ to check whether all threads may access barrier directive $br$ and lock region. If $br$ is not shown on some paths, then some threads may not call $br$ during execution, which indicates a deadlock. In addition, OAT can detect whether some lock variables are still held when some other threads try to access them, which also indicates a deadlock.

**OAT Detecting Algorithm**
Given OpenMP program $p$
MakeDataSharingExplicit($p$)
**if**(Error Injection is needed)
  ErrorInjection($p$)
$AST_p$ = ROSE_Parser($p$)
$\mathcal{F}(p) = \emptyset$
$\forall$ constructs $c \in AST_p\{$
  $\mathcal{F}(p) = \mathcal{F}(p) \cup \mathcal{F}(c)$ // encoding
}
Report = SMT_Solver($\mathcal{F}(p)$)

Figure 15. The Workflow of OAT.

## IV. Implementation

Our symbolic encoding is built on top of the ROSE [23], which is an open source compiler infrastructure for building source-to-source translators, and supports instrumentation in source code level. ROSE can parse the source code of an OpenMP program and generate an intermediate representation in the form of Abstract Syntax Tree (AST). ROSE currently supports OpenMP 3.0 and has dedicated AST nodes for all OpenMP directives [18]. By traversing the AST nodes using ROSE, OAT produces quantifier free first order logic formulae for Yices [2]. OAT translates all OpenMP regions into SMT constraints.

Figure 15 shows the analysis procedure of OAT. Variables used in OpenMP parallel regions have default data-sharing attributes (*e.g.*, `private` or `shared`) based on their definition places if not explicitly specified. We extended ROSE to automatically identify the data-sharing attributes of all variables, which is called `MakeDataSharingExplicit()`.

Our experiments are conducted on real-world OpenMP benchmarks. However, they are mainly for testing performance, hence are mostly free of data races and deadlocks. To our best knowledge, there is a lack of good benchmarks supporting OpenMP error detection. Hence, to test the accuracy and scalability of our tool, we automatically inject errors into OpenMP programs, which is discussed in details below.

### A. Error Injection

There are two major software error injection methods based on when the errors are inserted: compile-time or runtime. OAT utilizes compile-time error injection, which injects errors into the source code of the target program. OAT employs two ways to inject data races. The first way is to flip the data-sharing attributes of variables, as shown in the Case-1 of Figure 16. Specifically, flipping attributes includes exchanging `private/shared` and `firstprivate/lastprivate`, and removing variables from reduction clause. Our algorithm randomly chooses variables from data-sharing attribute variable list and flips its attributes. The other way is to add random write-read statements as shown in the Case-1 of Figure 16. To be simple, OAT randomly chooses a variable $u$ in data-sharing attribute variable list to execute $u = u * 2$. A dummy write $u = u$ will not work here because the value of $u$ is not really changed. Our race detection is based on checking nondeterministic values of shared variables. Note that both error injection methods may change the semantics of the programs, however, our purpose is to inject errors and test our tool.

The Case-2 in Figure 16 shows how to inject errors regarding deadlocks. If there exists a lock variable, OAT creates the same lock variable without releasing operating on this lock variable. Another error injection is to insert `#pragma omp barrier` to synchronization or worksharing constructs such as `omp critical`, `omp single`, `omp atomic`, `omp section`, `omp for`, and `omp master` to synchronize threads, which may prevent other threads from accessing the above regions .

```
//Ccase-1: Original code
#pragma omp parallel private( a) shared (b,c){
   a = a * 2;
   b = b * a;
}
// After data race injection:
#pragma omp parallel  shared (a, b,c)
{// private a is flipped to be shared.
   a = a * 2;
   b = b * 3;
   b = b  * 2;// statement of b are inserted.
}
// Case-2: Origin code
#pragma omp parallel shared (a, b,c) {
   a = a + b;
   c  = c * a
   #pragma omp critical{
   c = c +1;
   }
}
// After deadlock injection:
#pragma omp parallel shared (a, b, c){
   omp_set_lock(lock_t1)
   a = a + b;
   c  = c * a
   #pragma omp critical{
       c = c +1;
      #pragma omp barrier;
   }
}
```

Figure 16. Example of Error Injection In OpenMP Program.

## V. Experiments

This section discusses our experimental evaluation of OAT. The experiment was done on a workstation with Intel Xeon Quad-core E5160 3.0GHz, 16GB memory, and GCC v. 4.4.1. We conduct experiments on the real-world NAS Parallel OpenMP Benchmarks [21] in NPB2.3 (C version, with Class A as input), OpenMP Source Code Repository [22], and as well as student homework assignments on the course of High Performance Computing in the University of Wyoming. We use the approach introduced in Section IV-A to inject errors into the benchmarks. Three student homework assignments were chosen with typical race conditions. Since problems already exist in the student code, OAT analyzes the code directly without error injection.

We compare our tool OAT with two commercial tools, Intel Thread Checker 3.1 [9] and Sun Thread Analyzer in Oracle Studio 12.0 [26]. Both are runtime tools to check multi-threaded programs. According their manuals, Sun Thread Analyzer has incorporated some OpenMP features, but Intel Thread Checker lacks of such functionality.

### A. Data Race Detection

In Table I, "CG", "BT", "EP", "FT", "LU", "IS", "MG", and "SP" are from the NAS Parallel OpenMP benchmark package [21], "c_*" benchmarks are from OpenMP Source Code Repository [22]. "Stu.*" are student homework assignments. "LOC" is short for Lines of Code. "Base_Time" is the real execution time of the programs using two threads without any error checking. The "Number of Error Injection" represents the number of errors injected for testing. The last three columns give the number of data races reported by the three tools, followed by the total execution time in seconds.

We use a small dataset (*i.e.*, Class A) for all NAS Parallel Computing benchmarks. We found that the large input data size, such as Class S, will incur a very high overhead for the two dynamic commercial tools, which cannot terminate even after several hours of execution.

As shown in Table I, the OAT tool utilizes symbolic execution, which significantly reduces memory locations

| Code | LOC | Base_Time | # of Injected data races | OAT | Intel Thread Checker | Sun Thread Analyzer |
|---|---|---|---|---|---|---|
| CG class A | 922 | 3.69s | 2 | 2 (2.30s) | 2 (4.01s) | 2 (6.18s) |
| BT class A | 3617 | 243.27s | 1 | 1 (10.97s) | 2 (578.89s) | 1 (920.11s) |
| EP class A | 269 | 42.50s | 2 | 2 (5.45s) | 2 (121.10 s) | 2 (131.22s) |
| FT class A | 1143 | 8.47s | 0 | 0 (3.90s) | 0 (30.21s) | 0 (40.71s) |
| LU class A | 3482 | 47.86s | 0 | 0 (14.43s) | 0 (515.5s) | 0 (719.23s) |
| IS class A | 707 | 1.76s | 5 | 5 (4.01s) | 6 (52.09s) | 5 (64.99s) |
| MG class A | 1255 | 3.86s | 2 | 2 (5.09s ) | 2 (26.00s) | 2 (34.98s) |
| SP class A | 2986 | 150.82s | 3 | 3 (20.12s) | 2 (2501.12s) | 3 (3202.10s) |
| c_fft | 258 | 2.13s | 1 | 1 (2.09s) | 2 (20.19s) | 1 (25.43s ) |
| c_pi | 83 | 0.99s | 1 | 1 (1.28s) | 1 (10.91s ) | 1 (14.46s) |
| c_Jacobi | 295 | 2.98s | 1 | 1 (1.56s) | 1 (31.09s) | 1 (43.09s) |
| c_quicksort | 168 | 1.01s | 2 | 2 (2.99s) | 2 (3.01s) | 2 (5.09s) |
| c_mandel.c | 142 | 0.91s | 1 | 1 (2.01s) | 1 (3.09s) | 1 (4.98s) |
| Stu.1 | 98 | 0.99s | 3 | 3 (3.09s) | 3 (10.11s) | 3 (11.09s) |
| Stu.2 | 109 | 1.08s | 1 | 1 (2.09s) | 1 (10.90s) | 1 (10.80s) |
| Stu.3 | 123 | 1.07s | 2 | 2 (2.15s) | 2 (11.03s) | 2 (14.72s) |

Table I
COMPARISON OF DATA RACE DETECTION FOR OAT, SUN THREAD ANALYZER, AND INTEL THREAD CHECKER.

and synchronizations to monitor during runtime execution. Hence the OAT tool has a much lower overhead compared to Sun Thread Analyzer and Intel Thread Checker, which have an average 950% times and 890% times overhead, respectively. our OAT tool can successfully detect all injected data races and accurately report the race condition locations and scenarios. Our OAT tool is as accurate as Sun Thread Analyzer tool, where both report the same number of race conditions. However, the accuracy of Intel Thread Checker is less than the other two tools. Three false positives and one false negative are reported by the Intel Thread Checker. Specifically, Intel Thread Checker cannot fully support `#pragma omp critical`, some false positives are reported. The false negative occurs because some data dependencies in `#pragma omp sections` cannot be determined. The experiment on student homework assignments shows that all of three tools can detect the data races, which are mainly incurred by the indices in the nested loop of `#pragma omp for` worksharing region. Students ignore the data-sharing attributes of indices in loop initialization statement.

### B. Deadlock Detection

In OpenMP, deadlock is not so common as race condition. There are no existing benchmarks with deadlock. The error injection of OAT introduced several different kinds of deadlock errors by inserting `barrier` into parallel regions using `critical`, `atomic`, `master`, `ordered`, `section`. In addition, some lock variables are inserted using `omp_set_lock()` into `if-then-else` statements. OAT can successfully detect all of injected deadlocks and accurately report the deadlock locations and scenarios.

| Code | LOC | # of Injected Deadlocks | OAT | Intel | Sun |
|---|---|---|---|---|---|
| BT Class A | 3617 | 2 | 2 | 2 | 2 |
| CG Class A | 922 | 1 | 1 | 1 | 1 |
| LU Class A | 3482 | 2 | 2 | 2 | 2 |
| c_fft | 258 | 1 | 1 | 1 | 1 |
| c_pi | 83 | 1 | 1 | 1 | 1 |
| c_quicksort | 168 | 2 | 2 | 2 | 2 |

Table II
COMPARISON OF DEADLOCK DETECTION OF THE THREE TOOLS.

Due to the specific feature of OpenMP, deadlocks in OpenMP usually do not depend on scheduling. Hence deadlock in OpenMP is relatively easy to detect and fix. Table II compares the number of injected errors and reported results, all the three tools can report all deadlocks. However, unlike the other two runtime tools, our OAT tool can detect deadlock without running the program.

### C. False Positive and False Negative Analysis

The symbolic execution in our OAT tool is an approximation to real execution. Hence, there is potential to report false positives and false negatives. One reason is that the values in real execution are unavailable to our analysis. To keep the OAT tool efficient, our analysis takes approximation to avoid expensive simulations. For example, if a value of variable depends on the number of iterations in real execution, it is possible that our constraints cannot make the correct solution, therefore some false positives and negatives may be reported. Another reason is due to SMT solver itself. False positives may exist due to the facts that SMT solver is not able to precisely model all software artifacts. For example, Yices cannot solve nonlinear logic, hence our OAT

tool handles the nonlinear operating using bitvector type in Yices, it may cause false positives. However, all the above possibilities appear very rarely in practice. According to our experiments, the OAT tool can detect all injected data races and deadlocks.

## VI. Related Work

There are a few prior research tools and papers addressing error detection issues in OpenMP programs. We [20] presented the preliminary results on integrating symbolic analysis and dynamic analysis to detect concurrency errors in OpenMP programs. This paper introduces a more thorough and accurate approach of symbolic execution without dynamic analysis. One of the earliest is Intel Thread Checker [9], which rewrites program binary code with additional intercepting instructions to monitor a program's execution and infer possible parallel traces of the program. However, it treats OpenMP programs as general multi-threaded programs and does not consider the particularities of OpenMP programs, which makes the tool report false positives. Kim *et al.* [13] designed a practical tool called RaceStand that utilizes an on-the-fly dynamic monitoring approach to detect data races in OpenMP programs. RaceStand also includes a user-friendly web interface for ease of use and error reporting. Kang *et al.* [10] present a tool that focuses on the detection of first data races that are conflicting accesses with no explicit happen-before order in OpenMP programs. They first execute the instrumented program to obtain the conflicting accesses, then rerun the program with happened-before analysis to refine the conflicting accesses to those involved in the first data races. [7] focuses on race verification for debugging programs with OpenMP directives. Their approach is enhanced using a static analysis that identifies thread escape objects and inlines the instrumentation for better performance. Our approach differs from theirs in that OAT conducts a symbolic execution, where approximately simulates parallel program executions.

A few research tools have been proposed for OpenMP program analysis. Kim *et al.* [12] present a thread visualization tool that can visualize the partial order relationship between the traced threads in OpenMP programs in a three-dimensional code. This tool makes it more convenient for programmers to identify and manually confirm data race reports. However, its usage is limited when the number of threads increases exponentially, since it is almost impossible to manually examine the complex and massive intertwined graphs. Wang *et al.* [29] present a static analysis approach to check whether a barrier directive has been invoked by all threads within a team. Their implementation focuses on Fortran OpenMP applications.

SMT solvers have been applied to detect various concurrency errors. Li *et al.* [17] used an SMT solver not only to detect race conditions in GPU programs, but also to detect incorrect synchronized barriers. Said *et al.* [24] present an approach that uses an SMT solver to generate the data race witnesses that can be used to deterministically replay and confirm data race errors in multithreaded programs.

## VII. Conclusions and Future Work

In this paper, we present a tool named OpenMP Analysis Toolkit (OAT) for detecting data races and deadlocks in OpenMP programs. OAT first encodes OpenMP constructs into SMT formulae, then utilizes an SMT-solver to detect race condition and deadlock errors. Our experiments on a few real-world benchmarks show that OAT can detect concurrency errors in OpenMP programs accurately and efficiently. Compared to the runtime tools, our tool can detect errors automatically without actually running the programs, hence is much more efficient and scalable. In addition, our tool does not need complete program source code for checking concurrency errors, therefore it can help programmers identify potential errors during development stage.

In the future, we plan to enhance the OAT tool by leveraging dependence analysis and autoscoping of ROSE [19]. In the current approach, we just check deadlocks caused by barrier directives and locks without nested lock variables. We will implement nested locks to check deadlock errors. For data races, we will optimize the analysis on conditional statements and handle more complex constructs.

## Acknowledgement

This work was supported in part by NSF under Grant 1001239.

Prepared by LLNL under Contract DE-AC52-07NA27344

## References

[1] OpenMP execersie. https://computing.llnl.gov/tutorials/openMP/exercise.html.

[2] Yices: An SMT solver. http://yices.csl.sri.com.

[3] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224. USENIX Association, Berkeley, CA, USA, 2008.

[4] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265–278, Mar. 2011. ISSN 0362-1340.

[5] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223. ACM, New York, NY, USA, 2005. ISBN 1-59593-056-6.

[6] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 80–94. ACM, New York, NY, USA, 1994. ISBN 0-89791-683-2.

[7] O.-K. Ha and Y.-K. Jun. Efficient thread labeling for on-the-fly race detection of programs with nested parallelism. In *FGIT-ASEA/DRBC/EL*, pages 424–436, 2011.

[8] J. P. Hoeflinger and B. R. De Supinski. The OpenMP memory model. In *Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming*, IWOMP'05/IWOMP'06, pages 167–177. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 3-540-68554-5, 978-3-540-68554-8.

[9] Intel thread checker 3.1 for linux. http://software.intel.com.

[10] M.-H. Kang, O.-K. Ha, S.-W. Jun, and Y.-K. Jun. A tool for detecting first races in OpenMP programs. In *PaCT '09: Proceedings of the 10th International Conference on Parallel Computing Technologies*, pages 299–303. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03274-5.

[11] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 553–568. Springer-Verlag, Berlin, Heidelberg, 2003. ISBN 3-540-00898-5.

[12] Y.-J. Kim, J.-S. Lim, and Y.-K. Jun. Scalable thread visualization for debugging data races in OpenMP programs. In *Proceedings of the 2nd international conference on Advances in grid and pervasive computing*, GPC'07, pages 310–321. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 978-3-540-72359-2.

[13] Y.-J. Kim, M.-Y. Park, S.-H. Park, and Y.-K. Jun. A practical tool for detecting races in OpenMP programs. In *PaCT*, pages 321–330, 2005.

[14] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782.

[15] S. K. Lahiri, S. Qadeer, and Z. Rakamarić. Static and precise detection of concurrency errors in systems code using smt solvers. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 509–524. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-02657-7.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782.

[17] G. Li and G. Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 187–196. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-791-2.

[18] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski. A rose-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, editors, *IWOMP*, volume 6132 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 2010. ISBN 978-3-642-13216-2.

[19] C. Liao, D. J. Quinlan, J. Willcock, and T. Panas. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming*, 38(5-6):361–378, 2010.

[20] H. Ma, Q. Chen, L. Wang, C. Liao, and D. Quinlan. An OpenMP analyzer for detecting concurrency errors (poster paper). In *ICPP 2012: Proceedings of the International Conference on Parallel Processing*. IEEE Computer Society, Washington, DC, USA, 2012.

[21] Nasa advanced supercomputing division. http://www.nas.nasa.gov/publications/npb.html/.

[22] OpenMP Source Code Repository. http://sourceforge.net/projects/ompscr/.

[23] The rose compiler. http://www.rosecompiler.org/.

[24] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 313–327. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-20397-8.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997. ISSN 0734-2071.

[26] Oracle solaris studio 12.3. http://www.oracle.com/technetwork/server-storage/solarisstudio/.

[27] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 97–107. ACM, New York, NY, USA, 2004. ISBN 1-58113-820-2.

[28] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 256–272. Springer-Verlag, Berlin, Heidelberg, 2009.

[29] S. Wang and C. Huang. Static detection of deadlocks in openm p fortran programs. pages 44(3):536543,2007. Journal of Computer Research and Development, 2007. ISBN 1000.1239—CN 1 1,1777l TP.