Symbolic Analysis of Concurrency Errors in OpenMP Programs

Hongyi Ma[1], Steve R. Diersen[1], Liqiang Wang[1], Chunhua Liao[2], Daniel Quinlan[2], and Zijiang Yang[3]

[1]Department of Computer Science, University of Wyoming. {hma3, sdiersen, lwang7}@uwyo.edu
[2]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory. {liao6, dquinlan}@llnl.gov
[3]Department of Computer Science, Western Michigan University. {zijiang.yang}@wmich.edu

***Abstract*—In this paper we present the OpenMP Analysis Toolkit (OAT), which uses Satisfiability Modulo Theories (SMT) solver based symbolic analysis to detect data races and deadlocks in OpenMP codes. Our approach approximately simulates real executions of an OpenMP program through schedule permutation. We conducted experiments on real-world OpenMP benchmarks and student homework assignments by comparing our OAT tool with two commercial dynamic analysis tools: Intel Thread Checker and Sun Thread Analyzer, and one commercial static analysis tool: Viva64 PVS Studio. The experiments show that our symbolic analysis approach is more accurate than static analysis and more efficient and scalable than dynamic analysis tools with less false positives and negatives.**

I. Introduction

OpenMP is a portable parallel programming model used to create parallel C/C++ and Fortran multithreaded programs on shared-memory computing platforms. Developing OpenMP programs is prone to concurrency errors, such as data race and deadlock. 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. Figure 1 is an example of a data race in OpenMP. In this example, there is no data race if only considering the `for` loop. However, some threads that finish iterations early will execute `errors = dt[9]+1` while another thread may still be simultaneously executing the `for` worksharing region by writing to `d[9]`, which may cause a data race.

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 all 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. This is a deadlock because each thread would wait for the other to reach its own barrier, which will never happen.

This work was supported in part by NSF under Grant 1118059 and the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

```
#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 is done.\n", tid);
}
#pragma omp sections  {
   #pragma omp section
      print_results(c, 1);
   #pragma omp section
      print_results(c, 2);
}  /* end of parallel section */
```

Figure 2. Example of deadlock in OpenMP programs.

Traditionally, data races and deadlocks are detected using either dynamic analysis (*e.g.*[7], [15]) or static analysis (*e.g.*[6]). Static analysis is able to consider all possible behaviors without actually executing a program. However, it may produce false positives due to dynamic behaviors, such as aliases and pointers, which are impossible to obtain precisely. Furthermore, static analysis usually cannot report witnesses in terms of a trace leading to detected errors. Therefore, significant manual effort is 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 inappropriate for large-scale applications since the over-

head is usually prohibitively high.

Symbolic execution [5] attempts to explore all program paths under symbolic values. By encoding the current execution into first-order logic formula, predicative analysis [14] is able to predict errors accurately even under correct executions. The common method among these symbolic approaches is program encoding, followed by Satisfiability Modulo Theories (SMT) based solving, such as [8]. In order to achieve scalability, encoding must be carefully designed and optimized based on domain knowledge. In [10], we presented the preliminary results on integrating symbolic analysis and dynamic analysis for the detection of concurrency errors in OpenMP programs. In this paper, we propose a symbolic approach to detect race conditions and deadlocks in OpenMP programs; a more thorough and accurate approach of symbolic execution without requiring dynamic analysis. Our tool, called OpenMP Analysis Toolkit (OAT), is able to automatically detect data races and deadlocks accurately and efficiently. 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 [3].
- By interpreting the solution reported by Yices, we are able to reproduce a feasible execution trace that reveals the errors. This evidence-based approach not only improves the accuracy of error detection, but is also very useful in debugging programs and fixing problems.

Satisfiability Modulo Theories (SMT) solvers are logical deciders for problems ranging from basic boolean satisfiability problems to problems involving uninterpreted functions. SMT solvers utilize first-order logic and conjunctive normal form for the determination of the satisfiability of specific constraint satisfaction problems. OAT encodes OpenMP code regions into first-order logic constructs. These constructs are then simulated symbolically with Yices [3] to prove their satisfiability with regard to data race and deadlock errors.

II. Symbolic Encoding Algorithms

Static Single Assignment (SSA) is used to track variable updating by renaming variables with an increasing subscript for each write operation on the variable. Figure 3 shows the encoding of basic statements into logical formulae. We assume that the subscript of every SSA variable starts at 0.

```
int k, i = 0;          →   k0 = i0 = 0
int a[2] = {0,0};      →   ∧a0[0] = a0[1] = 0
a[0] = i * k;          →   ∧a1[0] = i0 × k0
i++;                   →   ∧i1 = i0 + 1
```

Figure 3. Encoding of OpenMP basic statements

Branch statements require the possible merging of variable definitions after the branch. Figure 4 shows a typical conditional statement with two possible execution paths. ite is a keyword in Yices, which denotes an `if-then-else` formula. Within the `then` branch variables j and k are modified, and variable i is modified in the `else` branch. To ensure the same SSA value for each variable following the `if-then-else` statement, instead of using Φ function, we will create dummy operations in each branch for those variables that were not originally coded into the branch. For example, we add the assignment $i_1 = i_0$ in the `then` branch and $j_1 = j_0 \wedge k_1 = k_0$ to the `else` branch. By inserting these assignments, the subscripts for an assigned variable after the branch will not depend upon the branch taken. This means any reads of the variables following the `if-then-else` statement will refer to the same SSA value. In the following paragraphs we will describe the intuition behind the other construct encodings.

```
if (i > 0){                ite(i0 > 0
  j = i * 10;              ∧j1 = i0 × 10
  k = j - i;          →    ∧k1 = j1 − i0
}                          ∧i1 = i0)
else{                      ∨(i0 <= 0
  i = j + k;               ∧i1 = j0 + k0
}                          ∧j1 = j0 ∧ k1 = k0)
```

Figure 4. Encoding of OpenMP branches.

OpenMP Parallel Region: Parallel regions can include both iterative and non-iterative segments. By default, the number of forked threads is determined by the number of processor or CPU cores. Since OpenMP provides functions to fork a given number of threads, our approach is general and can handle any number of threads. OpenMP usually runs in the SPMD (single program, multiple data) way, *i.e.*, every thread runs the same code but different dataset.

Worksharing Construct: Within 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. SSA encoding handles most variable situations; however, arrays are a special case. Although different loop scheduling policies are allowed, we assume the static scheduling policy for `omp for` loops. As shown in Figures 5 and 6, we use a superscript to identify the accessing thread ID. The line numbers are encoded into formulae in order to locate the reported problems in the source code. Note that we intentionally assume that there is

only one thread in Figures 3 and 4 and omit thread ID for simplicity.

Multi-dimensional arrays need to be translated into one-dimensional arrays. For example, in Figure 6, there is a two-dimension $array[1D][2D]$, each element $array[i][j]$ is translated to be $array[i*2D+j]$. We use $i[i_m^{t_n}]$ and $j[j_k^{t_n}]$ to represent the value of loop index i and j, where $i_m^{t_n}$ and $j_k^{t_n}$ mean the timing order of i and j in the loop, respectively. N denotes the total number of execution threads and n denotes the thread ID number.

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

$\rightarrow$

$omp.sectionsbegin = 99$
$\wedge omp.sectionbegin = 100 \wedge omp.par = T$
$\wedge array[i[i_1^{t_n}]+1][array_1^{t_n}] = array[i[i_0^{t_n}]][array_0^{t_n}]+1$
$\wedge omp.sectionend = 102 \wedge t_n = n \wedge n \in [0,...,N-1]$
$\wedge i_1^{t_n} > i_0^{t_n} \wedge i_1^{t_n} = array_1^{t_n}$
$\wedge i_0^{t_n} = array_0^{t_n} \wedge omp.sectionsend = 102$

Figure 5. Encoding of OpenMP `section` construct, where $i_0^{t_n}$, $i_1^{t_n}$, $array_0^{t_n}$, and $array_1^{t_n}$ indicate the timing orders of i and $array$. Only one thread executes the `section`.

```
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   }
105 }
```

$\rightarrow$

$omp.forbegin = 100 \wedge omp.par = T$
$\wedge i[i_m^{t_n}] \in [lb + \lceil (ub-lb)/N * n, lb + \lceil (ub-lb)/N * (n+1))$
$\wedge j[j_k^{t_n}] \in [lb, ub)$
$\wedge array[(i[i_1^{t_n}]+1) * 2D + j[j_1^{t_n}]][array_1^{t_n}] =$
$array[i[i_0^{t_n}] * 2D + j[j_0^{t_n}]][array_0^{t_n}] + 1$
$\wedge t_n \in [0,1,2,...,N-1] \wedge omp.forend = 105$

Figure 6. Encoding of OpenMP `for` construct.

Without Loop Bound: OAT can analyze loops in two ways: without loop bound or with loop bound. Analyzing without loop bound requires loop abstraction to avoid loop unrolling, as in [17]. We use the range of the index value to construct constraints for two types of data race: over-write conflict and write-read conflict. Without-loop-bound uses the index range to determine if it is possible for different threads to gain accesses to the same array element. In this way, we do not need to consider the timing order. The only constraint we need to construct is the index range of the array for each thread.

The following constraints together with the encoding in Figure 6 are used to check the write-read conflict in $array[i+1][j] = array[i][j] + 1$ (line 103). A solution reported by the SMT-solver indicates a data race in a feasible execution trace. For example, if there is no barrier to synchronize threads, one thread may read an array element after another thread updates it, which incurs a non-deterministic result. A data race is detected if there exists an overlap between the range of $i+1$ in $array[i+1][j]$ and the range of i in $array[i][j]$. The overlap of $i^{t_1}+1$ range and i^{t_2} range can be easily determined by solving the constraints.

$i^{t_1} \in [lb, (lb+ub)/2) \wedge i^{t_2} \in [(lb+ub)/2, ub)$
$\wedge j^{t_1} \in [lb, ub) \wedge j^{t_2} \in [lb, ub)$
$\wedge (i^{t_1}+1) * 2D + j^{t_1} = i^{t_2} * 2D + j^{t_2}$

With Loop Bound: is used to detect data races that depend upon the number of iterations in a `for` loop. This is accomplished through symbolic execution of a loop, as in [11], and tracking the timing order of variables within the loop. For a loop m, let Max_m denote the maximum number of iterations, and $LoopBound_m$ denote the loop bound during symbolic execution, then $LoopBound_m \leq Max_m$. If the loop bound value cannot be obtained by analyzing loop structures (*i.e.*, there is a symbolic value in the loop condition), then OAT uses a default value for the loop bound.

Tracking the timing order requires more constraints. For example, in Figure 6, when the nested loop index $j[j_k^{t_n}]$ is beyond the bound (*i.e.*, `j >= ub`), the index value $i[i_m^{t_n}]$ would be incremented by `i++`, and $j[j_k^{t_n}]$ would be set to the initial value in its loop.

Another worksharing directive, `single`, dictates that only one thread may enter and execute the enclosed parallel region. Access to a `single` region is encoded by $t_n = n \wedge n \in [0,1,2,...,N-1]$. This ensures that n is a known thread ID and that only thread t_n can simulate this code section.

Data Clauses: are used to define the properties of data-sharing and other specific operations. OAT can encode: `private`, `firstprivate`, `lastprivate`, `shared`, `default`, and `reduction`. As shown in Figure 7, 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. Assume we have the data-sharing attribute variable `reduction`(sum). We use $sum = \sum sum[sum_1^{t_n}]$ to represent the `reduction` directive. Yices does not support $\sum$ directly; we use the ROSE [9] compiler to call the Yices C API to implement $\sum$ with a loop. Without loop bound uses the range of the loop index i; with loop bound uses the timing order $i_k^{t_n}$ and $i[i_k^{t_n}] = i[{i_{k-1}}^{t_n}] + 1$ is encoded for the increment `i++`.

Synchronization Directives are used to synchronize

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

$\rightarrow$

$sum_0 = 0 \wedge omp.parbegin = 103 \wedge omp.par = T$
$\wedge omp.forbegin = 101$
$\wedge i[i_k^{t_n}] \in [0 + \lceil ub/N \rceil * (t_n), 0 + \lceil ub/N \rceil * (t_n + 1))$
$\wedge sum[sum_1^{t_n}] = sum[sum_0^{t_n}] + x[i[i_0^{t_n}]][x_0^{t_n}]$
$\wedge sum = \sum sum[sum_1^{t_n}] \wedge sum_1^{t_n} > sum_0^{t_n} \wedge x_0^{t_n} = i_0^{t_n}$
$\wedge sum_0^{t_n} > x_0^{t_n} \wedge omp.forend = 104 \wedge omp.parend = 105$
$\wedge t_n \in [0, 1, 2, ..., N-1]$
$\wedge \{sum_1^{t_n}, sum_0^{t_n}, x_0^{t_n}, i_0^{t_n}\} \in [0, 1, 2, ..., 3N-1]$

Figure 7. Encoding of data clauses (without using loop bound).

```
100 #pragma omp parallel {
101 #pragma omp critical
102   block1
103 #pragma omp master
104   block2
105 #pragma omp atomic
106   block3
107 #pragma omp flush (data)
108   block4
109 }
```

$\rightarrow$

$omp.parbegin = 100 \wedge omp.par = T$
$\wedge omp.criticalbegin = 101 \wedge \mathcal{F}(block_1) \wedge omp.criticalend = 102$
$\wedge omp.masterbegin = 103 \wedge \mathcal{F}(block_2) \wedge omp.masterend = 104$
$\wedge omp.barrier = T \wedge omp.atomicbegin = 105 \wedge \mathcal{F}(block_3)$
$\wedge omp.atomicend = 106 \wedge omp.flushbegin = 107$
$\wedge data[data_k^{t_n}] = data[data_k^{t_n} - 1] \wedge \mathcal{F}(block_4)$
$\wedge omp.flushend = 108 \wedge omp.parend = 109$

Figure 8. Encoding of synchronization directives.

threads. OAT handles the synchronization directives related to race conditions and deadlocks, including `master`, `critical`, `barrier`, `atomic`, and `ordered`. The synchronization directives manage the timing order of each thread. Given an OpenMP construct *block*, 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 will execute the code in the *block*. Figure 8 shows an example of encoding synchronization directives, where $\mathcal{F}(block)$ indicates the entire formulae of *block*. In Figure 8, the translation of $data[data_k^{t_n}] = data[data_k^{t_n} - 1]$ shows that the variable $data$ in different threads at execution point $data_k^{t_n}$ would be its latest updated value. If one variable is updated in memory by some threads, but the remaining threads symbolically do not receive that update, it can cause non-deterministic results and an error will be reported.

Pointers, Aliases, and Function Calls: An alias for a variable can be created through pointers or references. For example, formal parameters for functions are considered aliases for actual variables when those parameters are pointers or references. We use intra-procedural alias and pointer analysis to determine all aliases for each variable. Then we encode each alias as its corresponding variable. In Figure 9, variables `*p` and `*q` are aliases for `a`, and `b` is an alias for `c`. After the initial setup, all references to `*p` and `*q` are encoded as `a` with the appropriate SSA value; `b` is treated the same, for `c`.

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

$a_0 = 0 \wedge p.pointer = a_{0_{address}} \wedge q.pointer = a_{0_{address}}$
$\wedge a_1 = a_0 + 2 \wedge a_2 = a_1 + 1$
$\wedge b_{0_{address}} = c_{0_{address}} \wedge a_3 = c_0 + 1$

Figure 9. Encoding of alias and pointers.

For function calls in OpenMP constructs, we do a basic inlining operation. When multiple scopes are involved, variables with the same name, but in different scopes, need to be distinguishable. To accomplish this we prepend a scope number to the variable t: ${}^n t$ indicates that variable t is in scope n. Our tool cannot fully handle recursion now. However recursive functions can also be inlined and processed by setting a recursion bound.

III. Detecting Concurrency Errors

A. Data Race

According to the OpenMP programming model, an OpenMP program is partitioned into segments that are a sequence of instructions ending with a synchronization instruction. We use event e to represent a write or a read instruction on a variable. Let $\pi(s) = \{e_1, ..., e_n\}$ be a concrete timing order for a segment s, where each variable has an SSA form. We define the variable value read by an event as 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 result. A solution to these SMT constraints reveals a race condition.

Figure 10 illustrates the encoding of a parallel variable update using two threads. The expression $omp.par = T$ indicates the current region is a parallel code region, which is enclosed by the encodings $parbegin$ and $parend$. Suppose v is a shared variable; we use the timing order for reads and writes to simulate operation orders from multiple threads. A timing order ID is defined for each variable in SSA form

```
100 #pragma omp parallel
101 {v = v + 1;} →
```
$omp.parbegin = 100 \wedge omp.par = T$
$\wedge v[v_1^{t_1}] = v[v_0^{t_1}] + 1 \wedge v[v_1^{t_2}] = v[v_0^{t_2}] + 1$
$\wedge v[v_0^{t_1}] = v[v_0^{t_1} - 1] \wedge v[v_0^{t_2}] = v[v_0^{t_2} - 1]$
$\wedge v_1^{t_1} > v_0^{t_1} \wedge v_1^{t_2} > v_0^{t_2} \wedge \{v_0^{t_1}, v_1^{t_1}, v_0^{t_2}, v_1^{t_2}\} \in [0, 1, 2, 3]$
$\wedge omp.parend = 101$

Figure 10. Encoding of shared variable update based on two threads.

for each thread. The encoding $v_m^{t_k}$ describes the timing order, where v is the variable, m is the SSA subscript, and t_k is the thread. The range of $v_m^{t_k}$ is [0,1,2,...] and $v[v_m^{t_k}]$ represents the value of v at SSA index m in thread t_k.

The assignment $v = v+1$ is modeled by $v[v_1^{t_1}] = v[v_0^{t_1}]+1$ and $v[v_1^{t_2}] = v[v_0^{t_2}] + 1$ for threads 1 and 2, respectively. The encoded timing orderings, $v_1^{t_1} > v_0^{t_1}$ and $v_1^{t_2} > v_0^{t_2}$, ensure that the L-value of v in the current thread is only updated after the R-value of v for each thread. To represent the dependence of the R-value on the last write of v we write $v[v_0^{t_1}] = v[v_0^{t_1} - 1]$ and $v[v_0^{t_2}] = v[v_0^{t_2} - 1]$. The last write on v could be from the current thread or another thread. Using Figure 10, suppose the SMT-solver generates $v_0^{t_1} = 0$, $v_1^{t_1} = 3$, $v_0^{t_2} = 1$, and $v_1^{t_2} = 2$. We then have $v[3] = v[0] + 1 \wedge v[2] = v[1] + 1 \wedge v[1] = v[0]$, which indicates v is incremented by 1. Now, if the SMT-solver generates $v_0^{t_1} = 0$, $v_1^{t_1} = 1$, $v_0^{t_2} = 2$, and $v_1^{t_2} = 3$, then we have $v[1] = v[0] + 1 \wedge v[3] = v[2] + 1 \wedge v[2] = v[1]$, which indicates v is incremented by 2. A non-deterministic result on shared variable v is detected and OAT will report a data race on v.

B. Deadlock

Barrier synchronization is a common cause of deadlock in OpenMP programs. The semantics of OpenMP require that all threads involved in a parallel region execute the same `barrier`; otherwise, a deadlock will occur. In addition, missing lock/unlock can also create a deadlock. Figure 11 illustrates a potential deadlock with regard to $lock\_a$. Assuming A is initially greater than B, then $lock\_a$ is acquired in the first if statement at line 101. The body of the if statement then swaps the values of x and y. The if statement at line 106 would evaluate to false, which means the body of the if statement does not get executed, $lock\_a$ is never released, and a deadlock is created.

Conditional statements require analyzing if each barrier is called by all threads or a proper subset of all threads (*i.e.*, well synchronized or not well synchronized). In well synchronized barriers for a conditional statement each branch executes the same number of barriers. We determine if locks are well synchronized by ensuring all lock variables are

```
99 int x = A, y = B;
100 if(x > y){
101   omp_set_lock(&lock_a);
102   x = x + y; y = x - y; x = x - y;
103 }
104 if(x > y){
105    omp_unset_lock(&lock_a);
106 }
```
→
$x[x_0^{t_n}] = A \wedge y[y_0^{t_n}] = B \wedge ite((x[x_0^{t_n}] > y[y_0^{t_n}]$
$\wedge lock\_a[0] = 1 \wedge x[x_1^{t_m}] = x[x_0^{t_m}] + y[y_0^{t_m}]$
$\wedge y[y_2^{t_m}] = x[x_2^{t_m}] - y[y_1^{t_m}] \wedge x[x_4^{t_m}] = x[x_3^{t_m}] - y[y_3^{t_m}])$
$\vee (x[x_5^{t_m}] = x[x_0^{t_m}] \wedge y[y_4^{t_m}] = y[y_0^{t_m}]$
$\wedge \neg(x[x_0^{t_n}] > y[y_0^{t_n}]))) \wedge$
$ite((x[x_6^{t_m}] > y[y_5^{t_m}] \wedge unlock\_a[0] = 1)$
$\vee \neg(x[x_6^{t_m}] > y[y_5^{t_m}]))$
$\wedge t_m = m \wedge m \in [0, 1, 2, ..., N-1]$
$\wedge \{x_k^{t_n}, y_k^{t_n}\} \in [0, 1, 2, ..., kn - 1]$
$\wedge t_n \in [0, 1, 2, ..., N-1] \wedge k \in [0, 1, 2, ...]$

Figure 11. Example of deadlock detection.

released when some threads try to obtain them. In Figure 11, our system uses arrays $lock\_a[]$ and $unlock\_a[]$ to represent the order of lock acquires and releases for $lock\_a$. For example, for the m^{th} lock acquire, $lock\_a[m-1]$ is m and $lock\_a[m]$ indicates the next lock acquire. Thus, $unlock\_a[m] = lock\_a[m+1] - 1$ indicates no deadlock, whereas $unlock\_a[m] \neq lock\_a[m+1] - 1$ indicates a deadlock because the lock variable $lock\_a$ is not released after the m^{th} lock acquire.

IV. Experiments

A. Technical Setup

Code	LOC	DR	OAT	PVS	ITC	STA	Code	LOC	DR	OAT	PVS	ITC	STA
CG	922	10	10	11	10	10	c_fft	258	1	1	1	2	1
BT	3617	1	1	1	2	1	c_pi	83	1	1	1	1	1
EP	269	2	2	2	2	2	c_Jacobi	295	1	1	1	1	1
FT	1143	0	0	1	0	0	c_quicksort	168	2	2	2	2	2
LU	3482	0	0	0	0	0	c_mandel.c	142	1	1	1	1	1
IS	707	5	5	5	6	5	Stu.1	98	3	3	3	3	3
MG	1255	2	2	2	2	2	Stu.2	109	1	1	1	1	1
SP	2986	3	3	3	2	3	Stu.3	123	2	2	2	2	2

Figure 12. Test codes used in the experiments. Code - name of test code, LOC - lines of code, DR - number of injected data races, the columns OAT, PVS, ITC, and STA list the number of data races detected for each test program.

The experiments were done on a workstation with a 2.3GHz Intel Core i5-2410M with 4GB Dual Channel DDR3 at 1333MHZ and GCC v. 4.3.1. We compared OAT with two dynamic analysis tools, Intel Thread Checker(ITC) 3.1 [4] and Sun Thread Analyzer(STA) in Oracle Studio 12.0 [16], and one static analysis tool, Viva64 PVS-Studio(PVS) [2].

Both STA and PVS incorporate OpenMP features, but Intel Thread Checker does not.

Our experiments were conducted on the NAS Parallel OpenMP Benchmarks [12] in NPB2.3 (C version, with Class A as standard test input), OpenMP Source Code Repository [13], as well as student homework assignments from the High Performance Computing course at the University of Wyoming. Errors were injected into the NAS Parallel and OpenMP Source Code Repository benchmarks. Data races were injected either by flipping the data-sharing attributes of variables or adding variable updating statements. Deadlock errors were injected by either insertion of a lock acquire or insertion of a `#pragma omp barrier` to a synchronization or work-sharing construct. The three student homework assignments chosen had typical race conditions already.

Figure 12 describes the test codes used in our experiments. "CG", "BT", "EP", "FT", "LU", "IS", "MG", and "SP" are from the NAS Parallel OpenMP benchmark package, "c_*" benchmarks are from the OpenMP Source Code Repository, and "Stu.*" are student homework assignments, used with the permission of the students. Tests for Data Race detection were conducted using 2, 4, 8, 16, and 32 threads. Tests for Deadlock detection were conducted with 2 threads. For all NAS Parallel Computing benchmarks, we used the standard test dataset (*i.e.*, Class A). We found that large datasets (*i.e.*, Class S) prevented the dynamic analysis tools from completing execution in a timely manner (*i.e.*, several hours without completion).

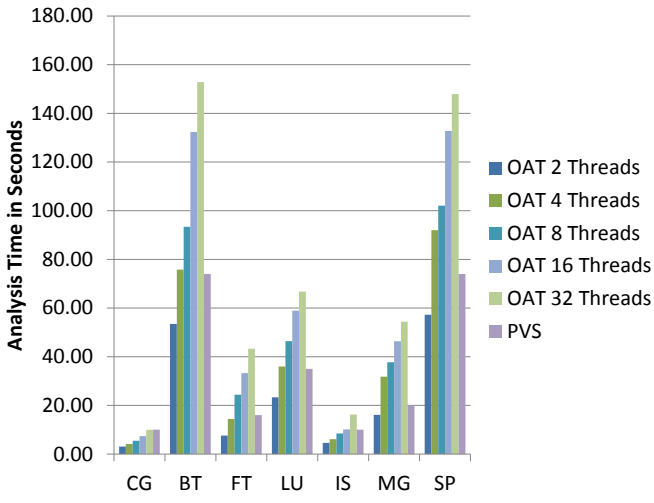

Figure 13. Analysis Time for OAT compared to PVS for benchmarks with greater than 500 lines of code.

B. Data Race Analysis

We designed our data race experiments to test for accuracy, efficiency and scalability. With regard to accuracy we were concerned with: (1) did we detect all data races (2) were any false positives generated and (3) were any false negatives generated? A false positive is reporting a data race where one does not exist and a false negative is failing to detect a data race. We found that the number of data races each analysis tool reported was independent of the number of threads used in the experiment because of the specific structures of OpenMP. Both OAT and STA found all data races in each experiment and did not produce any false positives. PVS found all data races, but also reported two false positives. One false positive was in the CG test and the other in the FT test. These false positives were reported due to updating variables outside parallel regions. In addition, PVS cannot handle pointer operations and races related to branches and specific number of iterations using static analysis, whereas our symbolic analysis is able to do these. However, we did not inject these types of errors since the current benchmarks are relatively simple and do not contain these corner-case structures in OpenMP regions. ITC found all data races with one exception, for the SP test ITC produced one false negative. The false negative is because some data dependencies in `#pragma omp sections` cannot be determined. ITC also reported three false positives. We believe this is because ITC cannot fully support `#pragma omp critical`.

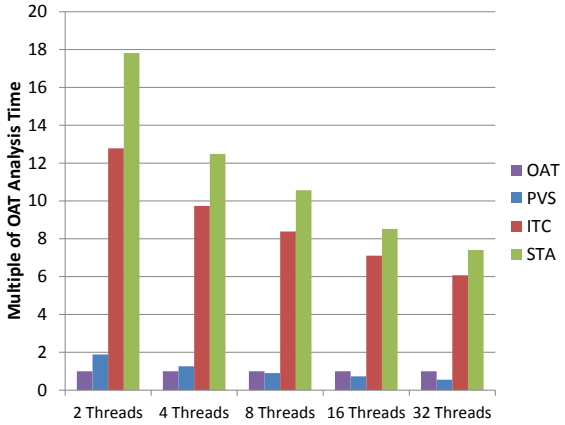

Figure 14. Comparison of OAT analysis time vs the other three analysis tools (PVS, ITC, STA) for test codes with greater than 500 LOC. All analysis times are normalized to OAT's analysis time.

We tested the efficiency of OAT vs. the other three analysis tools. Since two of the tools are dynamic and the other two are static, we believe that analysis time is the proper metric to judge efficiency between the four analysis tools. Analysis time is strictly the time from starting the analysis of the code until the reporting of the results. Figure 14 depicts analysis time comparisons between OAT and the other three analysis tools. Figure 13 is a direct comparison with PVS for 2, 4, 8, 16, and 32 threads. With small thread counts, OAT outperforms PVS, but as the thread count grows the size

of constraints created by OAT increases the analysis time. This was expected due to the nature of symbolic analysis. According to the experiments, testing two threads is enough to give an accurate analysis of OpenMP programs, because they often run in the SPMD mode.

Scalability has two different meanings in the context of these experiments: (1) number of threads and (2) size of the tested code (LOC). Scalability by the number of threads has been addressed above. Size of the test code was partially addressed by our experiments. As Figure 14 shows, OAT outperformed PVS when fewer threads were used. The analysis time of small code is not shown because the time of I/O and parsing dominates the whole execution time.

C. Deadlock Analysis

Code	DL	OAT	PVS	Code	DL	OAT	PVS
BT	2	60s	73s	c_fft	1	4s	6s
CG	2	8s	11s	c_pi	1	2s	3s
LU	2	26s	41s	c_quicksort	1	3s	4s

Figure 15. Analysis time required to find injected deadlocks in the tested codes. All tests run using 2 threads.

Our deadlock experiments were twofold: accuracy and efficiency. All four analysis tools were able to detect all injected deadlocks, without any false positives or false negatives. The analysis times varied considerably; ranging from a minute or less for the static analysis tools to being unable to terminate normally during deadlock detection for the dynamic tools. Specifically, the time comparison between OAT and PVS is shown in Figure 15.

V. Conclusions and Future Work

In this paper, we presented the OpenMP Analysis Toolkit (OAT) for detecting data races and deadlocks in OpenMP programs. By comparing our approach to two commercial dynamic analysis tools and one commercial static analysis tool, we have shown that OAT has comparable analysis time to the commercial static analysis tool, averaging 2.4x overall and 1.28x for test codes with greater than 500 lines of code compared to PVS for 8 threads. At 2 threads, OAT was 1.24x overall and 0.6x for codes with greater than 500 lines of code. OAT has faster analysis times than the two commercial dynamic analysis tools, averaging only 0.41x ITC analysis time and 0.32x STA analysis time for all test codes at eight threads. At 32 threads, OAT averaged only 0.53x ITC analysis time and 0.42x STA analysis time.

There are a number of ways to improve our tool. One way is to enhance OAT by leveraging dependence analysis and autoscoping of ROSE. We will also optimize the analysis of conditional statements and handle more complicated constructs.

References

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

[2] PVS-Studio, Static Code Analyzer for C, C++, and C++11. http://www.viva64.com/.

[3] Yices: An SMT Solver. http://yices.csl.sri.com.

[4] Intel Thread Checker 3.1 for Linux. http://software.intel.com.

[5] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7), Jul.

[6] S. K. Lahiri, S. Qadeer, and Z. Rakamarić. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In *CAV '09*.

[7] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782.

[8] 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.

[9] 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 *Proceedings of the 6th international conference on Beyond Loop Level Parallelism in OpenMP: accelerators, Tasking and more*, IWOMP'10, pages 15–28. Springer-Verlag, Berlin, Heidelberg, 2010.

[10] 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, 2012.

[11] M. d. Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '08, pages 161–166. IEEE Computer Society, 2008.

[12] NASA Advanced Supercomputing Division. http://www.nas.nasa.gov/publications/npb.html/.

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

[14] 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.

[15] 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.

[16] Oracle Solaris Studio 12.3. http://www.oracle.com/technetwork/server-storage/solarisstudio/.

[17] Y. Zhao and S. Malik. Exact Memory Size Estimation for Array Computations without Loop Unrolling. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 811–816. ACM, New York, NY, USA, 1999.