Dynamic Data Race Detection for OpenMP Programs

Yizi Gu
Computer Science Department
Rice University
Houston, USA
yizi.gu@rice.edu

John Mellor-Crummey
Computer Science Department
Rice University
Houston, USA
johnmc@rice.edu

***Abstract*—Two concurrent accesses to a shared variable that are unordered by synchronization are said to be a data race if at least one access is a write. Data races cause shared memory parallel programs to behave unpredictably. This paper describes ROMP – a tool for detecting data races in executions of scalable parallel applications that employ OpenMP for node-level parallelism. The complexity of OpenMP, which includes primitives for managing data environments, SPMD and SIMD parallelism, work sharing, tasking, mutual exclusion, and ordering, presents a formidable challenge for data race detection. ROMP is a hybrid data race detector that tracks accesses, access orderings and mutual exclusion. Unlike other OpenMP race detectors, ROMP detects races with respect to concurrency rather than implementation threads. Experiments show that ROMP yields precise race reports for a broader set of OpenMP constructs than prior state-of-the-art race detectors.**

***Index Terms*—Data race detection, OpenMP, OMPT, DynInst**

I. INTRODUCTION

OpenMP [1] is a sophisticated, directive-based model for writing shared-memory parallel programs. It provides a wide range of constructs including parallel regions that define a team of implicit tasks with SPMD parallelism, data environments that include both shared and private variables, parallel loops for homogeneous threaded and SIMD parallelism, parallel sections for heterogeneous parallelism, explicit tasks, and synchronization primitives that enforce ordering and/or mutual exclusion. Today, there is intense interest in augmenting scalable parallel applications written using MPI with OpenMP directives to exploit node-level parallelism. In fact, OpenMP is recommended as the node-level programming model for US DOE supercomputers, e.g., [2].

When adding shared-memory parallelism to node programs, application developers must avoid introducing *data races*—two or more *conflicting*, concurrent accesses to a variable that are unordered by synchronization; two accesses conflict if at least one is a write. Programs with data races may behave unpredictably because the value read from a variable may differ if it occurs before or after a racing write. Manually detecting races with print statements or a debugger is difficult because both strategies can affect access interleavings. For that reason, programmers need tools to help detect data races. Given the growing use of OpenMP, tools for detecting data races in OpenMP node programs are an urgent need.

To avoid overlooking races, static analysis of parallel programs to pinpoint data races must be conservative. As a result, for all but simple programs, static analysis to detect data races can produce many *false positives*—access pairs that it cannot prove are ordered or reference different data despite the fact that they cannot race in any program execution. For that reason, dynamic data race detection tools that monitor an execution of a parallel program to detect observed races are preferred by application developers.

Dynamic data race detectors use either post-mortem or on-the-fly analysis. Post-mortem race detection tools trace synchronization and memory accesses as a program executes and analyze the trace for races after the execution terminates. In contrast, on-the-fly race detectors check memory accesses for races as a program executes. Ideally, on-the-fly race detectors should minimize the extra state needed at runtime to check for races, minimize the cost of checking if an access is involved in a race, avoid false positives, and avoid *false negatives*— failing to report actual races. Prior work defines data race detectors that do not generate false positives as *precise* race detectors. Building a precise race detector requires maintaining detailed information about concurrency and synchronization. Since different parallel programming models have different primitives for concurrency and synchronization, detailed knowledge about a programming model's primitives is necessary to develop a precise data race detector for it.

Since a data race is caused by the absence of ordering between a pair of accesses to the same variable, a precise race detector must be aware of how a programming model's primitives impose ordering or mutual exclusion. The happens-before relation defines the order between two events in an execution. If two events are unordered by happens-before, they are concurrent. Reasoning about orderings and mutual exclusion requires different approaches. Reasoning about event orderings requires assigning a label to each task, updating that label at synchronization points, and comparing task labels either directly, e.g., [3], [4], or with the aid of auxiliary data structures, e.g., [5], [6]. Reasoning about mutual exclusion requires determining if two conflicting accesses were performed while holding one or more common locks. Prior work has shown that applying these methods separately causes race detectors to generate false positives. To avoid this shortcoming,

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

hybrid race detectors maintain happens-before orderings, lock sets, and an access history for each memory location [**?**]. The best strategy for maintaining information to reason about races depends upon a programming model's primitives.

Prior state-of-the-art OpenMP-aware race detectors [7], [8] employ thread-level schemes for labeling concurrency and checking for data races. Such strategies cannot avoid false negatives, which occur when conflicting accesses to shared variables are logically concurrent but are mapped to the same implementation thread. Such strategies can overlook races that would occur with a different partitioning of work, i.e., if the program was run on a different number of threads.

To address this shortcoming of existing OpenMP race detectors, we developed ROMP—an on-the-fly race detector for OpenMP programs that tracks logically concurrent OpenMP task intervals instead of thread-level concurrency. ROMP reasons about a broad range of OpenMP constructs including parallel regions, work-sharing constructs, tasks, data environments, barriers, critical sections (ordered and unordered), and locks. ROMP employs a hybrid race detection algorithm that combines happens-before analysis and lock set analysis. To reason about happens-before orderings between memory accesses, ROMP employs two strategies. For structured parallelism, ROMP extends offset-span labeling [3] to represent orderings that arise from work-sharing constructs and explicit tasking in addition to those from nested fork-join parallelism. For unstructured synchronization (i.e., task dependences between explicit tasks), ROMP performs graph reachability queries to evaluate the happens-before relation.

This paper describes the following contributions of ROMP:

- a hybrid algorithm that supports data race detection in parallel executions,
- novel mechanisms for tracking and reasoning about the concurrency of accesses performed by OpenMP tasks,
- mechanisms for tracking OpenMP data environments,
- extensions to the OMPT API needed to build a precise race detector for *abelian* OpenMP programs (parallel programs whose critical sections commute [9]), and
- an experimental evaluation that compares ROMP's capabilities to those of a prior state-of-the-art race detector.

The rest of the paper is organized as follows. Section II provides some background about data races and OpenMP constructs. Section III discusses related work on race detection. Section IV describes ROMP's hybrid data race detection algorithm and its novel strategy for maintaining and reasoning about orderings of accesses. Section V provides a high-level description of ROMP's implementation. Section VI presents an evaluation of ROMP with a standard set of data race detection benchmarks for OpenMP. Section VII presents our conclusions and future plans.

II. BACKGROUND

Data races have been formally defined in the literature [10], [11], [12]. We formally define a data race and some notation in this section for the paper to be self-contained. We begin by defining a happens-before relation [13] between two events in a concurrent program.

Definition 1. *Let ϵ_i and ϵ_j be two events (e.g., a read, write, or synchronization operation) in a concurrent program. Let $\rightarrow$ denote the happens-before relation between two events. $\epsilon_i \rightarrow \epsilon_j$ if: i) ϵ_i and ϵ_j occur in the same thread and ϵ_i precedes ϵ_j in program order, or ii) there exists a directed synchronization from ϵ_i to ϵ_j, or iii) there exists an event ϵ_k such that $\epsilon_i \rightarrow \epsilon_k$ and $\epsilon_k \rightarrow \epsilon_j$ (transitivity)*

Definition 2. *Let $\Rightarrow$ denote the following relation: ϵ_i and ϵ_j occur in the same thread, ϵ_i precedes ϵ_j in program order, and no fork/join or synchronization event occurs between them in the thread. Note that $\epsilon_i \Rightarrow \epsilon_j$ implies $\epsilon_i \rightarrow \epsilon_j$. But $\epsilon_i \rightarrow \epsilon_j$ does not imply $\epsilon_i \Rightarrow \epsilon_j$. We call this relation happens-before-serially.*

We use the happens-before relation to define the concurrent relation between two events:

Definition 3. *Let $\parallel$ denote the concurrent relation between two events. $\epsilon_i \parallel \epsilon_j$ iff $\neg(\epsilon_i \rightarrow \epsilon_j \vee \epsilon_j \rightarrow \epsilon_i)$.*

Definition 4. *A lock set is the set of implicit or explicit locks held when a memory access is performed.*

Definition 5. *There exists a data race between two memory access events ϵ_i and ϵ_j that access the same memory location if the following three conditions are all true: i) $\epsilon_i \parallel \epsilon_j$, and ii) ϵ_i's lock set does not share a common element with ϵ_j's lock set, and iii) at least one of ϵ_i and ϵ_j performs a write.*

OpenMP supports nested fork-join parallelism. Nested fork-join parallelism is realized by nesting OpenMP parallel regions. An OpenMP parallel region consists of a team of worker threads (implicit tasks). Each worker thread can create a nested parallel region. OpenMP constructs fall into four main categories: i) parallel construct, 2) task construct, 3) synchronization constructs 4) work-sharing constructs. Upon reaching a parallel construct, a parallel region is created and the code surrounded by the parallel construct is executed by all worker threads in that parallel region. Upon reaching a task construct, the encountering task spawns an explicit task. The spawned explicit task is concurrent with other tasks in the parallel region unless ordering is enforced by synchronization. Explicit tasks differ from implicit tasks, whose creation, execution and destruction correspond to structured fork-join parallelism. The happens-before relation is more difficult to analyze for explicit tasking because of its unstructured nature. Upon entering a work-sharing construct, work to be performed is partitioned and distributed to a team of worker threads either statically at compile time or dynamically at run time.

OpenMP includes synchronization directives related to mutual exclusion (e.g., `critical`) or barriers (e.g., `ordered`) and data sharing clauses that specify whether a variable is task private or shared. A dynamic OpenMP data race detector should be able to analyze the effects of OpenMP constructs and data sharing attributes in programs as they execute.

III. RELATED WORK

Race detectors use a variety of representations for happens-before relationships. Offset-span labeling, which represents happens-before relationships in series-parallel graphs [3], labels each task with a vector of tuples proportional to the task's nesting depth. The Cilk project's Nondeterminator [5] and Nondeterminator-2 [9] maintain happens-before orderings using a series-parallel bags (SP-bags) representation that employs path compression on balanced trees [14] to reduce task label comparisons to nearly constant time. This approach supports a fully-strict concurrency model where a task signals completion to its parent. Raman et al. [15] extended this approach to Habanero Java's terminally strict async-finish synchronization model, where a task may signal completion to any single ancestor. While SP-bags approaches are efficient and increase space by only a small constant factor, they only express happens-before orderings that arise from fork and join operations and maintenance of this representation is serial.

To support on-the-fly race detection in parallel executions, Raman et al. [6] use a Dynamic Program Structure Tree (DPST) to model relationships between tasks in Habanero Java. Their SPD3 race detector infers ordering between two tasks by searching a DPST for their least common ancestor. A recent paper by Agrawal et al. [16] proposes a DAG reachability based race detection algorithm for detecting races in a program with arbitrary synchronization constraints in addition to series-parallel constructs. While Agrawal et al.'s algorithm requires serial execution of programs, ROMP supports on-the-fly race detection in parallel program executions.

Many tools employ hybrid algorithms that consider both happens-before ordering and lock sets when checking for data races, e.g. [9], [**?**], [17], [11].

The Archer data race detector for OpenMP [7] uses static analysis to create a black of source lines (e.g., those in serial code) guaranteed to be race free. Archer's black list and synchronization orderings gathered by an annotated runtime are passed as annotations to a dynamic analysis module based on Google's ThreadSanitizer, known as TSan [17]. TSan uses a thread-centric, vector-clock scheme for reasoning about orderings. Because of TSan's thread-centric design, Archer cannot avoid false negatives when conflicting, logically concurrent accesses are both performed by the same thread.

Atzeni et al. [8] describe SWORD—an OpenMP data race detector that reduces memory overhead by having each thread record a history of its accesses into a log file. In a post-mortem phase, SWORD analyzes log files for races using an algorithm based on offset-span labeling [3]. Offset-span labeling lacks support for reasoning about inter-task synchronization—a challenge that we address in ROMP.

IV. APPROACH

Here, we describe ROMP's approach for detecting data races on-the-fly as an OpenMP program executes. Section IV-A describes ROMP's hybrid data race detection algorithm. Sections IV-B, IV-C and IV-D describe mechanisms for tracking and reasoning about the concurrency of accesses performed by OpenMP tasks. Section IV-F describes mechanisms for tracking OpenMP data environments.

Algorithm 1 A hybrid data race detection algorithm

```
1: procedure CHECKDATARACE(l, [ϵ,a,h])
2:     skip_current ← FALSE
3:     for [ϵ′,a′,h′]∈ History[l] do
4:         if ϵ ‖ ϵ′ ∧ h ∩ h′ = ∅ ∧ (a = w ∨ a′ = w) then
5:             report a data race
6:         if ((a′ = w ∧ a = w) ∨ a′ = r) ∧ h′ ⊇ h ∧ ϵ′ → ϵ then
7:             History[l]← History[l] - [ϵ′,a′,h′]
8:             continue
9:         if ((a′ = r ∧ a = r) ∨ a′ = w) ∧ h ⊇ h′ ∧ ϵ′ ⇒ ϵ then
10:            skip_current← TRUE
11:    if skip_current is False then
12:        History[l]← History[l] ∪ [ϵ,a,h]
```

A. Hybrid Data Race Detection Algorithm

All-Sets [9] is a powerful hybrid data race detection algorithm designed for detecting data races in a *depth-first serial* execution of a Cilk program. ROMP employs a variant of All-Sets shown in Algorithm 1 adapted to support data race detection during a *parallel* execution of an OpenMP program. The key differences between ROMP's algorithm and All-Sets are the criteria for pruning an access history (lines 6 and 9 in Algorithm 1). ROMP cannot use the pruning criteria employed by All-Sets because All-Sets assumes pseudo transitivity of the parallel relation between memory access events (i.e., if $\epsilon_1 \parallel \epsilon_2$ and $\epsilon_2 \parallel \epsilon_3$, $\epsilon_1 \parallel \epsilon_3$); pseudo-transitivity does not hold for programs that execute in parallel. This difference can be explained without considering lock sets. All-Sets does not record information about an access a in the access history for a memory location l if l's access history already contains a concurrent access a' with the knowledge that *in its depth-first serial execution*, any future access a'' to l that is concurrent with a is also concurrent with a'. For programs executing in parallel, pruning based on pseudo transitivity can cause races to be missed. For example, the task t' that performed a' may still be executing while task t performs a. Not adding a to the access history would cause a race to be missed if t' subsequently performs a write w' that would conflict with a but is not concurrent with a' since $a' \Rightarrow w'$. With that clarification out of the way, we now explain ROMP's hybrid race detection algorithm in its entirety.

In line 1, l stands for the memory location being accessed. $[\epsilon,a,h]$ stands for an access record, where ϵ is a memory access event; a is the type of access, either a read (r) or a write (w); h is the set of mutual exclusion entities being held on the memory location at the time of access. 'History' in line 3 is the access history for each memory location maintained by the algorithm. Access history pruning is viable because the goal of a data race detector is to report at least one data race on any memory location that contains data races. Multiple data races on a memory location are likely to be correlated so reporting one representative data race on a memory location is efficient and practical. Access histories may be pruned in two ways:

1) Line 6 checks if an access record in the history could be discarded while adding the current access record to access history; 2) Line 9 checks if the current access record need not be added to the access history. When the predicate in line 6 is true, any future access racing with history access A' whose access record is [ϵ',a',h'] will also race with the current access A whose access record is [ϵ,a,h], so A' may be pruned. When the predicate in line 9 is true, any future access racing with current access A will also race with the history access A'. In this case, A need not be recorded in the history. Note that in line 9, we use relation *happens-before-serially* instead of *happens-before*. By definition, relation *happens-before-serially* eliminates the possibility of the following scenario: Let events of history, current, future accesses be $\epsilon', \epsilon, \epsilon''$ respectively. Then $(\epsilon' \rightarrow \epsilon) \wedge (\epsilon' \rightarrow \epsilon'') \wedge (\epsilon \parallel \epsilon'')$ satisfies $\epsilon' \rightarrow \epsilon$, but does not satisfy $\epsilon' \Rightarrow \epsilon$. If the access record for current access not recorded and if at least one of the memory access events is a write, an OpenMP data race detector will miss detecting the data race.

B. OpenMP Task Graph Model

Parallel program execution forms a directed acyclic graph (DAG) [16], [3]. Prior work on OpenMP race detection reasons about a thread-based DAG that models structured fork-join parallelism in OpenMP programs [18], [19], [8]. A thread-based DAG is limited in its ability to model a rich set of OpenMP constructs. Because each physical thread can execute several tasks during its lifetime, a thread-based model cannot accurately represent logical concurrency between OpenMP tasks. We use an OpenMP task graph model that is an extension to the fork-join graph [3] to model logical concurrency and happens-before ordering in an OpenMP program execution.

Definition 6. *An OpenMP task graph $G = (V, E, v_{src}, v_{snk})$ is a directed acyclic graph such that*

- *Each vertex v is either an implicit task vertex of type T_{imp}, an explicit task vertex of type T_{exp}, a logical task vertex of type T_{lgc}, or a synchronization vertex of type T_{syn}. Vertices with type T_{imp}, T_{exp}, T_{lgc} are called task vertices.*
- *A directed edge e_{ij} exists between two task vertices v_i and v_j if v_i is parent task of v_j and the following predicate is true: $(v_i.type = T_{imp} \wedge v_j.type = T_{imp}) \vee (v_i.type = T_{imp} \wedge v_j.type = T_{lgc}) \vee (v_i.type = T_{lgc} \wedge v_j.type = T_{imp}) \vee (v_i.type = T_{exp} \wedge v_j.type = T_{imp})$.*
- *A directed edge e_{ij} exists between task vertex v_i and synchronization vertex v_j, if v_i performs synchronization operation that creates vertex v_j.*
- *A directed edge e_{ij} exists between synchronization vertex v_i and task vertex v_j, if $\epsilon_i \rightarrow \epsilon_j$, where ϵ_i and ϵ_j denote the events associated with v_i and v_j, respectively.*
- *A directed edge e_{ij} exists between two synchronization vertices v_i and v_j, if $\epsilon_i \Rightarrow \epsilon_j$, where ϵ_i and ϵ_j denote the events associated with v_i and v_j, respectively.*

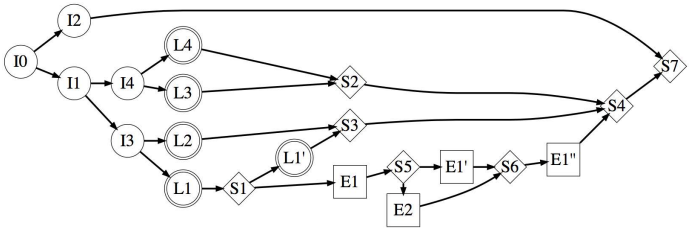


Fig. 1: OpenMP task graph for program in listing 1

Listing 1: An OpenMP program example

```
#pragma omp parallel
#pragma omp single
#pragma omp parallel for
for (int i = 0; i < 4; i ++) {
    if (i == 0) {
        #pragma omp task
        {
            #pragma omp task
            {
            }
            #pragma omp taskwait
        }
    }
}
```

Let ϵ be the event associated with vertex v in a task graph of an OpenMP program. It is obvious that $\epsilon_i \rightarrow \epsilon_j$ if there exists a directed path connecting v_i and v_j in the task graph.

Figure 1 shows an example of OpenMP task graph for the program in Listing 1. For simplicity of the graph, we assume that each parallel region has two worker threads.

In Figure 1, circles represent implicit tasks, double circles represent logical tasks, squares represent explicit tasks, and diamonds are vertices that represent synchronization. In Figure 1, vertex E_1 corresponds to the outer explicit task in Listing 1. At some point in its execution, represented by synchronization node S_5, E_1 creates an an inner explicit task E_2. Following synchronization point S_5, the continuation of E_1 is represented as E_1'. Events in vertex E_1 happen before events vertex E_2 because there exists a directed path between them. In contrast, events in vertex E_2 and events in vertex E_1' are in parallel because no directed path connects the two vertices. Synchronization vertex S_6 represents the `taskwait` directive and task vertex E_1'' corresponds to the outer task in Listing 1 after executing the `taskwait` directive. Vertex E_1'' is reachable from vertex E_2, so events in E_2 happen before events in E_1''. This is consistent with the semantics of `#pragma omp taskwait`.

Each work-sharing loop iteration is a logical task vertex. In Figure 1, implicit task vertices I_3 and I_4 are each assigned two iterations. While each implicit task executes its two iterations sequentially, concurrency still exists among all iterations of a work-sharing loop. By assigning each iteration a logical task vertex, concurrency in a work-sharing loop is exposed in our task graph model. This enables ROMP to represent and reason about concurrency among iterations. One can derive the corresponding task graph from the semantics of OpenMP

synchronization constructs. Vertices S_2 and S_3 correspond to the end of the work-sharing loop for each implicit task. while vertex S_4 and S_7 represent the implicit barriers at the end of the inner and outer parallel regions, respectively. The novelty of our OpenMP task graph model is its ability to fully depict logical concurrency among tasks in OpenMP programs with arbitrary valid combinations of OpenMP constructs.

If an `ordered section` were present, it would cause logical tasks in the graph to be split into pieces that represent before, during, and after the ordered section with synchronization nodes interspersed. An edge from the synchronization node representing the end of the ordered section on one logical task would connect to the synchronization node prior to the ordered section in the next logical task.

C. ROMP Task Labeling

The data race detection algorithm discussed in Section IV-A requires mechanisms for reasoning about the concurrency of accesses performed by OpenMP tasks. Here we describe a scheme for assigning a *task label* to each node in our task graphs to solve the problem of analyzing the concurrency of memory accesses performed by OpenMP tasks.

Our task labeling scheme is inspired by offset-span thread labeling [3]. Offset-span labeling solved the problem of reasoning about concurrency of memory accesses by a program with nested fork-join parallelism. If an OpenMP program only contains parallel constructs (i.e., nested fork-join parallelism), offset-span labeling suffices. However, extensions to offset-span labeling are needed to represent the full spectrum of concurrency and synchronization possible with OpenMP.

Definition 7. *A label L for a vertex in an OpenMP task graph consists of a sequence of label segments: $L = S_1S_2...S_k$. Each label segment is marked as implicit, explicit or logical, mapping to a type of task vertex. The type of the last label segment in L reflects the type of task vertex labeled by L.*

A label segment encodes information about OpenMP constructs related with current task. It is composed of the following fields: {offset, span, iter-id, task-wait-count, task-create-count, loop-count, phase, task-waited, task-group-info, segment-type}. Each field holds information about different OpenMP constructs. The original offset-span labeling mechanism only contains two fields: offset and span; we use several additional fields to represent the full range of OpenMP ordering concerns. During the execution of an OpenMP program, the state of an OpenMP task changes as it encounters different OpenMP constructs and synchronization events. The task label for each task also changes accordingly.

As an OpenMP program runs, it creates tasks. To create a task's label, ROMP copies the label of its parent task and appends a new label segment. Let the label of parent task be $L_p = S_1S_2...S_k$. Then the label for the newly created task is $L_n = S_1S_2...S_kS_{k+1} = L_pS_{k+1}$. Besides creating the new label, the last segment of the parent label will be modified accordingly to record the effect of the task creation event. This label creation rule has two implications: First, by inheriting the label of parent task, the history of task creation dating back to the creation of initial task is available for the current task. This makes reasoning about concurrency of accesses by comparing task labels possible. Second, the number of label segments of a task label is equal to the number of task vertices along the path of task creation in the OpenMP task graph. The following paragraphs briefly describe how the fields of the label segment encode information of various OpenMP constructs.

1) Parallel Construct: Upon encountering a parallel construct, a team of implicit tasks is created. Each new implicit task gets a task label by appending a new label segment to the label of the encountering task. In the new label segment, offset is set to the relative id of the implicit task within the team and span is set to the size of the team.

2) Work-sharing Loop Construct: Each work-sharing loop iteration is treated as a logical task. A task label is created as an iteration is dispatched. For the appended label segment, the iter-id is set to the relative id of the iteration.

3) Task Construct: An explicit task is created upon encountering the task construct. The task label creation procedure is the same as the one for implicit task. However, the (offset,span) pair is set to (0,1) because we regard an explicit task as a team with only one task (itself). After the label is created, the task-create-count field in the last segment of the parent label is incremented by one to record the change of state due to the creation of the explicit task.

4) Implicit/Explicit Barrier: We treat a barrier as a fork immediately after a join. We label a task leaving a barrier by updating the second-to-last segment of its task label prior to the barrier by adding its span to its offset.

5) Ordered Section: Upon entering or exiting an ordered section, increment the phase field in the last segment of the task label. The phase field is for reasoning about happens-before relation with tasks that encounter ordered section.

6) Taskwait / Taskgroup: Upon encountering a `taskwait`, increment the task-wait-count field in the last segment of the task label. Upon encountering a `taskgroup` construct, modify the task-group-info field which can encode nesting task-group information.

ROMP uses our task labeling scheme to reason about the concurrency of memory accesses by comparing segments of the labels of tasks performing the accesses. ROMP compares label segments of two task labels left to right to find the first pair of segments that differ in the segment fields. Then, ROMP uses information stored in the fields to reason about the concurrency of associated memory accesses.

D. Other Synchronization Constraints

Although structured OpenMP constructs enable direct encoding of ordering information using task labels, some OpenMP synchronization constraints cannot be directly encoded in task labels. In such cases, ROMP maintains additional information about these constraints, as described below.

1) Task Dependencies: OpenMP task dependences are expressed using dependence variables. ROMP maintains a dependency graph built from dependency variables passed by

OMPT callbacks as described later in Section V-B. ROMP directly searches the graph to check for dependencies between explicit tasks. ROMP only checks task dependences when two accesses are judged concurrent by their task labels and both accesses are performed by explicit tasks with one or more dependence variables specified for each task.

2) Reductions: OpenMP's `reduce` clause directs worker threads in a parallel region to combine their private partial results into a final result. Without considering the thread synchronization performed during a reduction, access to a peer's thread-private variable would appear to be a data race. ROMP employs a flag to indicate whether a task is performing a reduction in a parallel region. While performing a reduction, ROMP treats memory accesses to variables written by another implicit task labeled as a peer as safe. If an OpenMP runtime library is correctly implemented, variable accesses during a reduction region should not race with accesses by peers.

3) OpenMP critical or atomic sections: For OpenMP critical or atomic sections, ROMP utilizes OMPT callbacks to get notified upon entering and exiting the sections. A unique id is assigned to each section by the OMPT callback, ROMP treats entering and exiting the same critical or atomic section as locking and unlocking a lock with the unique id.

E. OpenMP SIMD Directive

Dynamic data race detectors (including ROMP) cannot support race detection for OpenMP SIMD constructs with today's compilers. A compiler either replaces scalar code marked with an OpenMP SIMD directive with a vector version or not. Without the original scalar code as a guide, a dynamic race detector cannot determine if the vectorizer changed the program semantics by ignoring a data race associated with a data dependence.

F. Data Environment Tracking

In OpenMP, each variable instance has an associated data scoping attribute. A variable may be shared or private. Shared variables may be accessed by tasks other than the binding task. Private variables are meant to only be accessed by the binding task. ROMP must track data scoping attributes for variables to accurately identify data races. Consider the following. During execution, a thread may execute an explicit task T1, using space on its stack to hold T1's local variables. When T1 finishes, the thread may execute a logically concurrent explicit task T2 using the same stack space. Without considering the stack space used by each task as private, it would appear that these two logically concurrent tasks were racing by using the same stack space. Algorithm 2 shows how ROMP manages state of memory locations. The algorithm assumes the stack grows downward. In the algorithm, mem_addr is the address of the memory location being accessed. Line 2 to 3 checks if the memory access is not on the local thread stack. If the access is on the local thread stack, line 6 checks if the access is above the current task's base runtime frame. If so, it means that current task is accessing another task's data, (i.e., shared access). Line 5 updates the lowest stack memory address accessed so far by the task. This serves as the lower bound of the local stack access. When a task finishes or suspends, stack local memory locations are marked as deallocated, as shown in line 11 and 12. When memory is marked as deallocated, its access history is lazily discarded. Note that if a task is explicit, the runtime library may allocate task private data on the heap. Line 16 and 17 reset the boundary of the stack local memory locations.

Algorithm 2 Data Sharing Attributes Management

```
1:  procedure UPONMEMORYACCESS(mem_addr)
2:      if mem_addr falls out of thread stack then
3:          mem_addr.state ← shared
4:      if mem_addr falls within thread stack then
5:          thread.lowest ← min(thread.lowest, mem_addr)
6:          if mem_addr > thread.cur_task_frame_base then
7:              mem_addr.state ← shared
8:          if mem_addr ≤ thread.cur_task_frame_base then
9:              mem_addr.state ← private
10: procedure UPONTASKSCHEDULE(old_task, new_task)
11:     for mem_addr ∈ [thread.lowest, thread.active_task_frame] do
12:         mem_addr.state ← deallocated
13:     if old_task is explicit then
14:         for mem_addr ∈ {old_data.heap_task_private_data} do
15:             mem_addr.state ← deallocated
16:     thread.active_task_frame ← new_task.task_frame_base
17:     thread.lowest ← new_task.task_frame_base
```

V. IMPLEMENTATION

ROMP is implemented as a shared library that provides all data race checking protocols and maintains shadow memory for access histories. We use DynInst [20]—an open source binary instrumentation tool—to statically rewrite an executable by inserting a call to `CheckAccess` before each memory access instruction. To make a runtime-independent race detector such as ROMP possible, we worked with the OpenMP language committee to extend the design of the OMPT tool interface for OpenMP 5.0 with sufficient hooks to meet the needs of precise race detectors. When an application executes, the OpenMP runtime system invokes OMPT callbacks implemented in the ROMP library to track concurrency using task labels and the application calls ROMP's `CheckAccess` routine to maintain access histories and check for data races.

Figure 2 shows how ROMP serves as a data race detection library.

A. Shadow Memory Management

Each memory location has an associated slot in shadow memory. A slot stores an *access history* for the associated memory address. ROMP manages shadow memory using a two-level page table, which is similar to the organization used by Valgrind [21], but is more lightweight. As described in Section IV-A, pruning is always legitimate. Multiple memory accesses that touch the same memory address query the same access history. Two threads may race to inspect and update an access history for a variable if they both access the variable at the same time. ROMP uses an MCS queueing lock [22] in each slot to guard against racing updates to an access history.

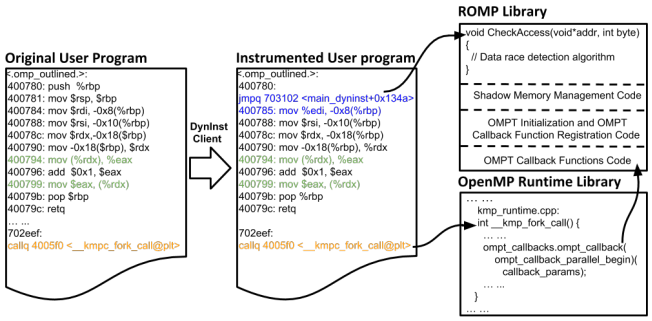


Fig. 2: Illustration of ROMP system workflow

Threads may inspect or update independent access histories concurrently. ROMP can directly infer a race condition on a memory address when a thread wanting to inspect and update an access history for a write access finds the slot already locked. Lock contention for a slot occurs if two calls to `CheckAccess` are concurrent, meaning that they are not ordered by either the happens-before relation or mutual exclusion. Lock contention is identified by a failing `try_lock()` operation on a slot's lock. In our implementation, when ROMP knows that there are races associated with a memory location due to lock contention for its access history, ROMP skips the traversal of access records and task label checks and immediately reports the race.

B. Metadata Management

As described in section IV-C, task labels are modified as appropriate at OpenMP runtime events, (e.g., task creation, barriers, etc.). ROMP relies upon the LLVM OpenMP runtime library which includes a draft implementation of the OpenMP 5.0 OMPT monitoring API [23]. The OpenMP 5.0 version of the OMPT interface enables a monitoring tool to store pointers to tool-defined metadata for parallel regions and tasks. ROMP allocates metadata for OpenMP tasks and parallel regions and uses these pointers to associate ROMP's metadata with runtime implementation of these OpenMP constructs. For precise race detection, we worked with the OpenMP language committee to add two new callback notifications to the OMPT interface for OpenMP 5.0. To detect races between two iterations of a parallel loop that are scheduled to the same physical thread, OMPT was extended to support a callback at the beginning of each loop iteration. To avoid false race reports for accesses associated with reductions, OMPT was extended to invoke a callback when a thread begins and completes a parallel reduction associated with a loop, parallel region, or task group.

C. Data Environment Management

Section IV-F describes ROMP's mechanism for tracking data environments. ROMP keeps track of states for memory locations being accessed by leveraging OMPT callbacks. ROMP calls OMPT API function `ompt_get_task_info` to get the base address of current task frame. ROMP calls `ompt_get_thread_data` to access thread local data structures that store variables `thread.*` shown in algorithm 2. ROMP registers a `ompt_callback_task_schedule` callback to receive notification when a task is scheduled. ROMP manages the state of each memory location by manipulating bits in the access history for the memory location. When a memory location is deallocated, ROMP only marks it as deallocated and does not clear access history records for this memory location until it is allocated again. This lazy destruction policy is a trade-off between memory usage and running time. Note that clearing access records is safe because future memory accesses will not race on dead variables.

VI. EVALUATION

We evaluate the precision, recall, and accuracy of ROMP as well as its runtime and memory overhead using two suites of OpenMP microbenchmarks: DataRaceBench [24] and OmpSCR [25]. DataRaceBench was designed for systematic evaluation of OpenMP data race detection tools [24]. OmpSCR [25] contains OpenMP programs that implement common scientific computing algorithms. We performed our experiments on a single-node server equipped with four 1.9GHz 12-core AMD Opteron 6168 processors and 128GB of memory.

A. Evaluation Methodology

Liao et al. [24] use DataRaceBench to compare the quality of four different tools for detecting data races in OpenMP programs: Helgrind, ThreadSanitizer, Archer and Intel Inspector. Their results show that OpenMP-aware tools such as Intel Inspector and Archer are better than other tools according to three metrics: Precision (P), Recall (R) and Accuracy (A). These metrics are computed using counts of TP—true positives, TN—true negatives, FP—false positives, and FN—false negatives:

- P = TP/(TP+FP)
- R = TP/(TP+FN)
- A=(TP+TN)/(TP+FP+TN+FN)

We choose to compare ROMP's effectiveness with Archer [7], which was found to be the most effective OpenMP data race detector in Liao et al.'s study using DataRaceBench [24]. To fairly compare ROMP with Archer, we replicated the evaluation method and parameters used in the prior evaluation of Archer [24]. For each test case, we control the number of threads through the *OMP_NUM_THREADS* environment variable and select from a set of numbers: (3, 36, 45, 72, 90, 180, 256). For test cases that can allocate arrays with different sizes based on the command line input, we choose the array sizes from the following set (32, 64, 128, 256, 512, 1024). We run each test case five times with same input. We define TP/TN/FP/FN as follows: Suppose each benchmark program p is run multiple times with different parameters. Denote the test result of a race detector D on p as $D(p)$. Then $D(p)$ = TP if p contains race conditions and at least one race is reported in each run. $D(p)$=FP if p is data race free but at least one race is reported by D among all runs. $D(p)$=TN if p is data race free and no race is reported among all runs. $D(p)$=FN if p contains race conditions but in at least one run, no race is reported. Our definitions of TP and TN are different

ID	Microbenchmark Program	R	Archer	ROMP	ID	Microbenchmark Program	R	Archer	ROMP
1	antidep1-orig-yes.c	Y	TP	TP	2	antidep1-var-yes.c	Y	TP	TP
3	antidep2-orig-yes.c	Y	TP	TP	4	antidep2-var-yes.c	Y	TP	TP
5	indirectaccess1-orig-yes.c	Y	TP	TP	6	indirectaccess2-orig-yes.c	Y	FN	TP
7	indirectaccess3-orig-yes.c	Y	FN	TP	8	indirectaccess4-orig-yes.c	Y	FN	TP
9	lastprivatemissing-orig-yes.c	Y	TP	TP	10	lastprivatemissing-var-yes.c	Y	TP	TP
11	minusminus-orig-yes.c	Y	TP	TP	12	minusminus-var-yes.c	Y	TP	TP
13	nowait-orig-yes.c	Y	FN	TP	14	outofbounds-orig-yes.c	Y	TP	TP
15	outofbounds-var-yes.c	Y	TP	TP	16	outputdep-orig-yes.c	Y	TP	TP
17	outputdep-var-yes.c	Y	TP	TP	18	plusplus-orig-yes.c	Y	TP	TP
19	plusplus-var-yes.c	Y	TP	TP	20	privatemissing-orig-yes.c	Y	TP	TP
21	privatemissing-var-yes.c	Y	TP	TP	22	reductionmissing-orig-yes.c	Y	TP	TP
23	reductionmissing-var-yes.c	Y	TP	TP	24	sections1-orig-yes.c	Y	TP	TP
25	targetparallelfor-orig-yes.c	Y	TP	TP	26	taskdependmissing-orig-yes.c	Y	FN	TP
27	truedep1-orig-yes.c	Y	TP	TP	28	truedep1-var-yes.c	Y	TP	TP
29	truedepfirstdimension-orig-yes.c	Y	TP	TP	30	truedepfirstdimension-var-yes.c	Y	TP	TP
31	truedeplinear-orig-yes.c	Y	TP	TP	32	truedeplinear-var-yes.c	Y	TP	TP
33	truedepscalar-orig-yes.c	Y	TP	TP	34	truedepscalar-var-yes.c	Y	TP	TP
35	truedepseconddimension-orig-yes.c	Y	TP	TP	36	truedepseconddimension-var-yes.c	Y	TP	TP
37	truedepsingleelement-orig-yes.c	Y	FN	TP	38	truedepsingleelement-var-yes.c	Y	FN	TP
39	3mm-parallel-no.c	N	TN	TN	40	3mm-tile-no.c	N	TN	TN
41	adi-parallel-no.c	N	TN	TN	42	adi-tile-no.c	N	TN	TN
43	doall1-orig-no.c	N	TN	TN	44	doall2-orig-no.c	N	TN	TN
45	doallchar-orig-no.c	N	TN	TN	46	firstprivate-orig-no.c	N	TN	TN
47	fprintf-orig-no.c	N	TN	TN	48	functionparameter-orig-no.c	N	TN	TN
49	getthreadnum-orig-no.c	N	TN	TN	50	indirectaccesssharebase-orig-no.c	N	TN	TN
51	inneronly1-orig-no.c	N	TN	TN	52	inneronly2-orig-no.c	N	TN	TN
53	jacobi2d-parallel-no.c	N	TN	TN	54	jacobi2d-tile-no.c	N	TN	TN
55	jacobiinitialize-orig-no.c	N	TN	TN	56	jacobikernel-orig-no.c	N	FP	TN
57	lastprivate-orig-no.c	N	TN	TN	58	matrixmultiply-orig-no.c	N	TN	TN
59	matrixvector1-orig-no.c	N	TN	TN	60	matrixvector2-orig-no.c	N	FP	TN
61	outeronly1-orig-no.c	N	TN	TN	62	outeronly2-orig-no.c	N	TN	TN
63	pireduction-orig-no.c	N	FP	TN	64	pointernoaliasing-orig-no.c	N	TN	TN
65	restrictpointer1-orig-no.c	N	TN	TN	66	restrictpointer2-orig-no.c	N	TN	TN
67	sectionslock1-orig-no.c	N	TN	TN	68	simd1-orig-no.c	N	TN	TN
69	targetparallelfor-orig-no.c	N	TN	TN	70	taskdep1-orig-no.c	N	TN	TN
71	ordered-section-missing-yes.c	Y	FN	TP	72	taskgroup-missing-yes.c	Y	TP	TP
73	taskwait-missing-yes.c	Y	TP	TP	74	doall2-orig-yes.c	Y	FN	TP
75	fibonacci-orig-yes.c	Y	TP	TP	76	taskdep2-orig-no.c	N	FP	TN
77	taskdep3-orig-no.c	N	FP	TN	78	taskdep-chain-orig-no.c	N	FP	TN
79	taskwait-orig-no.c	N	FP	TN	80	multiple-ordered-section-no.c	N	FP	TN
81	taskgroup-taskwait-mix-no.c	N	TN	TN	82	fibonacci-orig-no.c	N	FP	TN
Summary	**Precision**	**Archer – 0.79** **ROMP – 1.00**		**Recall**	**Archer – 0.79** **ROMP – 1.00**		**Accuracy**	**Archer – 0.78** **ROMP – 1.00**	

TABLE I: Data race detection report I. Column R: Y if a program contains a data race; N if it is data race free

from the definitions in [24]. In their definitions, the data race detection result on a benchmark program could be listed as TP and FN, or TN and FP at the same time (e.g., for a race free benchmark program, detector reports races in some runs, and doesn't report races in other runs, in [24], $D(p)$ is both TN and FP.) We regard our definitions being stricter and could reflect the effectiveness of a race detector more accurately.

To evaluate the efficiency of ROMP, we ran ROMP and Archer (version 0.1) on OmpSCR [25] benchmark suite. We set environment variable `OMP_NUM_THREADS` to 48, which equals to the maximum number of physical threads available on our experimental platform. Benchmark programs are compiled using clang (version 8.0.0) with compilation flags '-g -O2 -fopenmp -fpermissive -ltcmalloc'. After compilation, the benchmark program is instrumented by our DynInst client `omp_race_client`. We conducted 10 independent runs for each benchmark program and collected the elapsed real time and the resident set size of the program during its lifetime using command `/usr/bin/time -f "%E %M"`. The result is described in Section VI-C.

B. DataRaceBench Results

Table I shows results for ROMP and Archer on DataRaceBench. The type of result (i.e., TP/TN/FP/FN) is determined according to the definition in section VI-A. We do not include experiments with two microbenchmarks that employ the OpenMP SIMD construct `simdtruedep-{orig|var}-yes.c` for the reason discussed in section IV-E.

We found that the first 70 microbenchmarks (ID:1–70) used in the evaluation of Archer in prior work [24] do not cover all OpenMP constructs. For example, `ordered section`, `taskwait`, and `taskgroup` synchronizations are not tested in the 70 benchmark programs. To make a more comprehensive evaluation of the effectiveness of both tools, we added another 12 benchmark programs (ID: 71–82) that cover those OpenMP constructs. Benchmarks 74, 76 and 77 are from DataRaceBench. We developed our additional test

cases following the design guideline of DataRaceBench [24]: each test program either contains zero or one data races[1]

Our experiments show that ROMP successfully detects data races in benchmark programs `indirectaccess{2|3|4}_orig-*.c`. Archer and SWORD [8] (a new OpenMP race detector) yield false negatives on these programs due to the thread-centric nature of their data race detection algorithms. For these programs, two iterations may be scheduled to the same physical thread either statically at compile time or dynamically by the OpenMP runtime library; in such cases, thread-based data race detection algorithms do not report a data race. ROMP's algorithm is based on tasking and does not suffer from this problem. We list the effectiveness metrics at the bottom of Table I. The result demonstrates the high effectiveness of ROMP due to its novel task labeling mechanism which analyzes logical concurrency between tasks.

We enforce dynamic scheduling of work-share loop iterations in benchmark programs 5–8 by specifying `schedule(dynamic)`. An OMPT dispatch callback notifies ROMP as each iteration of the loop begins execution. With static scheduling, ROMP will report false negatives due to an incomplete logical task label. This shows that our task labeling scheme reports races caused by loop-carried dependences.

C. OmpSCR Results

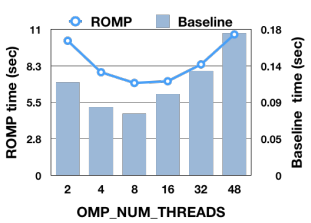
Table II shows the average runtime and memory overhead of ROMP and Archer on our platform. Labels in the note column of Table II have the following meaning: ST means that the program is test with '-test' argument, with default input size and using maximum number of threads on the system. MT means that the program is tested with the input size similar to the one set by '-test' argument using customized number of threads[2]. CR means that the input parameter is customized because ROMP's run time exceeds our limit with the default '-test' argument. CA means that the input parameter is customized because Archer's run time exceeds our limit with the default '-test' argument. Arguments for programs labeled with MT, CR and CA are listed in the Table II.

Table II also shows data race detection results of ROMP and Archer. For benchmark programs reported by ROMP to contain a data race, we checked the source code and verified the race[3] For example, ROMP found a data race at line 127, column 10 of c_loopA.badSolution. It is a data race caused by a loop-carried dependence. We found that `cpp_qsomp7` contains Intel-specific OpenMP pragmas, namely, `#pragma intel omp taskq`, which is not recognized by the clang compiler. Without the effect of this pragma, each thread launches an instance of the outermost explicit task to sort the vector instead of just one. This causes data races because

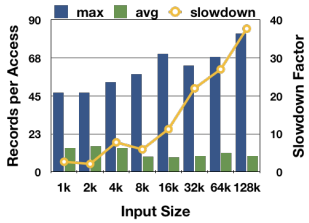
[1]Some of our new test cases have been added to DataRaceBench 1.2.0.

[2]For `cpp_qsomp{1|2|3|4|6|7}`, argument '-test' sets the number of running thread to 1, while cpp_qsomp5 fixes the number of threads to 2.

[3] Original `cpp_qsomp3` and `cpp_qsomp4` crash because they access an empty vector called `globalStackWrite`. We fixed the programs by initializing the vector before the parallel region and performed the experiments.



(a) ROMP running time for OmpSCR's cpp_qsomp3 using 2 – 48 threads to sort 10^6 numbers on a system with 48 single-threaded cores.



(b) ROMP slowdown factor and the maximum/average number of access records per access for OmpSCR's cpp_qsomp3 using 48 threads with different input sizes.

the vector will to be sorted concurrently by all OpenMP threads. With this compiled code, ROMP instantly identifies the data race. Since the idea of `cpp_qsomp7` is to parallelize the divide-and-conquer phase of quicksort using OpenMP explicit tasking, instead of ignoring this benchmark program, we modified the directives of the code to make it comply with OpenMP standard. With this modification, it should not contain a data race. Our result shows that ROMP does not report any data race for this modified version of the program.

The data race ROMP found in `c_md` is related with a read that is beyond the end of an allocated array in line 178. That read accessed another array written by concurrent tasks.

In terms of geometric mean, ROMP's slowdown is comparable to that of Archer, while the memory overhead of ROMP is roughly 2.5× smaller than that of Archer.

ROMP and Archer both have a large slowdown for some of the benchmark programs. For ROMP, we found that if a memory address is widely read shared (e.g., a global variable that each task frequently reads), the lock protecting the access history results in signifiant lock contention. ROMP's performance could be improved by reducing the lock contention. To investigate the relation between ROMP's slowdown and the number of access records visited upon each checking, we collected the histogram with respect to the number of access records for the benchmark programs. Table II lists the maximum and average number of access records visited per access. `cpp_qsomp7` recursively forks explicit tasks whose task labels are different. The maximum number of records per access is proportional to the number of different tasks in the program, because a shared variable could be accessed by every task and each task contains a unique task label.

D. Parallel Execution

Some prior data race detectors require serial execution of a program [9], [16]. Like Archer and SWORD, ROMP executes OpenMP programs in parallel while checking them for data races. Figure 3a shows the execution time of `cpp_qsomp3` being checked by ROMP with different numbers of OpenMP threads. When the number of threads is below 16, ROMP's running time improves with additional threads.

Microbenchmark	Note	Original Time(s)	Original Mem(kb)	ROMP					Archer		
				Type	Time	Mem	ML	AL	Type	Time	Mem
c_fft	ST	0.112	1.28e4	TN	7.23×	8.40×	48	24	TN	18.0×	11.9×
c_fft6	ST	0.045	1.64e4	TN	3.31×	2.69×	48	17	TN	29.1×	11.7×
c_jacobi01	ST	0.97	5.98e5	TN	286×	28.7×	48	1	RE	–	–
c_jacobi02	ST	0.85	5.95e5	TN	447×	28.9×	48	1	RE	–	–
c_loopA.badSolution	ST	0.066	1.02e4	TP	7.64×	3.35×	48	1.6	TP	6.10×	17.7×
c_loopA.solution1	ST	0.092	1.06e4	TN	10.3×	4.29×	48	1.7	TN	6.28×	21.9×
c_loopA.solution2	ST	0.1	9.37e3	TN	6.13×	3.38×	48	1.6	TN	6.35×	22.5×
c_loopA.solution3	ST	0.09	1.00e4	TN	7.02×	3.26×	48	1.6	TN	6.63×	21.4×
c_loopB.badSolution1	ST	0.089	1.02e4	TP	5.81×	3.59×	48	1.6	TP	6.36×	21.6×
c_loopB.badSolution2	ST	9.53	9.78e3	TP	8.82×	4.44×	48	1.4	TP	1.27×	21.3×
c_loopB.pipelineSolution	ST	0.14	9.4e3	TN	5.63×	3.74×	48	1.6	TN	15.8×	19.6×
c_lu	ST	0.268	1.32e4	TN	607×	310×	499	64	CE	–	–
c_mandel	ST	0.079	9.62e3	TN	1.53×	2.93×	47	17.9	TN	25.9×	18.0×
c_md 256 10	CR	0.27	1.08e4	TP	213×	2.74×	49	21	FN	8.7×	22.1×
c_pi	ST	0.047	9.87e3	TN	2.28×	2.64×	47	19	TN	32.9×	16.5×
c_qsort	ST	0.065	9.31e3	TN	4.09×	3.27×	48	3.1	TN	26.0×	13.7×
c_testPath	ST	0.028	9.53e3	TP	2.79×	2.78×	47	21.4	TP	46.6×	20.5×
cpp_qsomp1 10^7 48 10^3	MT	0.78	4.83e4	TP	85.5×	31.2×	96	5	TP	5.53×	8.49×
cpp_qsomp2 10^7 48 10^3	MT	0.743	4.97e3	TP	91.8×	30.6×	96	4.7	TP	5.27×	8.56×
cpp_qsomp3 10^7 48 10^3	MT	0.643	5.09e4	TP	106×	29.8×	78	3.1	TP	6.06×	8.34×
cpp_qsomp4 $2 \cdot 10^5$ 24 10	CA	0.13	1.03e4	TP	6.78×	4.84×	24	2.6	TP	1194×	18.2×
cpp_qsomp5 $2 \cdot 10^5$ 24 10	CA	0.04	9.71e3	TP	24.5×	3.63×	1	1	TP	541×	14.9×
cpp_qsomp6 10^7 48 10^3	MT	0.772	4.81e4	TP	85.8×	31.5×	96	6.1	TP	5.37×	8.74×
cpp_qsomp7 $2 \cdot 10^4$ 24 10	CA	0.026	1.04e4	TN	169×	3.00×	3372	123	TN	491×	495×
Summary	Mean		–	–	91.5×	23.1×	–	–	–	118×	39.2×
	Median		–	–	8.23×	3.69×	–	–	–	8.7×	18×
	Geometric Mean		–	–	20.5×	7.28×	–	–	–	18.2×	18.3×

TABLE II: Overheads and data race detection results on the OmpSCR benchmark suite with 48 threads. ML means the maximum number of access records visited per access. AL means the average number of access records visited per access. RE means the augmented program encounters a runtime error. CE means compilation error. Measurements shown are the average of 10 independent runs. The timings for ROMP in this table are measured without printing details of each data race.

VII. CONCLUSIONS

This paper describes ROMP—a precise race detector that supports all OpenMP constructs except SIMD. While we have not yet experimented with OpenMP's TARGET pragma, we believe that we can generate TARGET code for the host and analyze it with ROMP just like we do host code.

While Archer relies on support for race detection in the LLVM OpenMP runtime, ROMP lives on top of OMPT callbacks and thus should work with any OpenMP runtime library that supports the emerging OpenMP 5.0 standard.

Experiments show that unlike Archer, the most capable OpenMP race detector to date, ROMP has perfect accuracy, precision, and recall when checking for races in executables for OpenMP programs. ROMP improves upon Archer's precision by reasoning about the logical concurrency in an OpenMP program rather than the concurrency of implementation threads and tracking data environments. One important shortcoming of ROMP at present is that it does not apply thorough checking to all dynamically-loaded shared libraries used by an application. Remedying this deficiency is the subject of future work.

For OmpSCR benchmark programs, ROMP's space overhead is 2.5× smaller than that of Archer, while its time overhead is comparable to that of Archer. Using Rice University's HPCToolkit performance tools [26] to analyze the time overhead of data race detection with ROMP gave us some insights into scaling and performance issues that arose with some of the benchmarks. Widely-shared data that is concurrently accessed by multiple threads slows program executions monitored by ROMP because ROMP protects shadow memory slots using mutual exclusion. We see three principal opportunities for improving ROMP's performance. First, access histories for widely shared data contain more entries than necessary to detect representative races. While the complex concurrency and synchronization relationships in OpenMP make it non-trivial to prune access histories by dropping labels of concurrent tasks without missing races, we have identified some opportunities for doing so. This would mean refining the pruning performed by Algorithm 1 using knowledge of OpenMP concurrency to keep only labels *necessary* for detecting representative races rather than those that are *sufficient* for detecting races. Second, if we use the aforementioned idea to avoid modifying an access history on most every read, using a reader-writer lock might provide additional concurrency for shadow memory inspection. Third, we identified that ROMP maintains access histories for read-only data, which could never be the subject of a race; accesses to read-only data need not be monitored. Realizing these improvements is the subject of ongoing work.

ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation under Grant No.1450273. We thank Mark Krentel for contributing key insights related to the design of task labels associated with OpenMP `ordered` sections.

REFERENCES

[1] OpenMP Language Committee, "OpenMP Application Programming Interface, version 4.5," http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, November 2015.

[2] National Energy Research Scientific Supercomputing Center, "Programming models," http://www.nersc.gov/users/computational-systems/cori/programming/programming-models, December 2017, last accessed: March 28, 2018.

[3] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/125826.125861

[4] A. Dinning and E. Schonberg, "An empirical comparison of monitoring algorithms for access anomaly detection," in *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*, ser. PPOPP '90. New York, NY, USA: ACM, 1990, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/99163.99165

[5] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in Cilk programs," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '97. New York, NY, USA: ACM, 1997, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/258492.258493

[6] 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.

[7] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "Archer: Effectively spotting data races in large OpenMP applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 53–62.

[8] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, I. Laguna, G. L. Lee, and D. H. Ahn, "Sword: A bounded memory-overhead detector of OpenMP data races in production runs," in *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, to appear.

[9] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark, "Detecting data races in Cilk programs that use locks," in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '98. New York, NY, USA: ACM, 1998, pp. 298–309.

[10] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 121–133. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542490

[11] E. Pozniansky and A. Schuster, "MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007. [Online]. Available: http://dx.doi.org/10.1002/cpe.1064

[12] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992. [Online]. Available: http://doi.acm.org/10.1145/130616.130623

[13] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[14] R. E. Tarjan, "Applications of path compression on balanced trees," *J. ACM*, vol. 26, no. 4, pp. 690–715, Oct. 1979. [Online]. Available: http://doi.acm.org/10.1145/322154.322161

[15] 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

[16] K. Agrawal, J. Devietti, J. T. Fineman, I.-T. A. Lee, R. Utterback, and C. Xu, "Race detection and reachability in nearly series-parallel DAGs," in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '18. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2018, pp. 156–171. [Online]. Available: http://dl.acm.org/citation.cfm?id=3174304.3175277

[17] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 62–71.

[18] H. Ok-Kyoon and Y.-K. Jun, "Efficient thread labeling for on-the-fly race detection of programs with nested parallelism," in *Proceedings of the International Conference on Advanced Software Engineering and Its Applications*, 2011, pp. 424–436.

[19] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun, "On-the-fly detection of data races in OpenMP programs," in *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, ser. PADTAD 2012. New York, NY, USA: ACM, 2012, pp. 1–10.

[20] Computer Sciences Department, University of Wisconsin-Madison and Computer Science Department, University of Maryland, "Dyninst API," http://www.dyninst.org/dyninst, last accessed: January 28, 2018.

[21] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/1254810.1254820

[22] J. M. Mellor-Crummey and M. L. Scott, "Synchronization without contention," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 269–278. [Online]. Available: http://doi.acm.org/10.1145/106972.106999

[23] OpenMP Language Committee, "OpenMP Technical Report 7: Version 5.0 public comment draft," http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf, July 2018.

[24] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin, "DataRaceBench: A benchmark suite for systematic evaluation of data race detection tools," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 11:1–11:14. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126958

[25] A. J. Dorta, C. Rodriguez, F. d. Sande, and A. Gonzalez-Escribano, "The openmp source code repository," in *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 244–250. [Online]. Available: https://doi.org/10.1109/EMPDP.2005.41

[26] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.v22:6

A. ARTIFACT DESCRIPTION

1) Overview:

- *How software can be obtained.* ROMP can be downloaded from https://github.com/zygyz/romp .
- *Hardware dependencies.* We used a single-node server equipped with four 1.9GHz 12-core AMD Opteron 6168 processors and 128GB of memory. The server runs a version of the Linux operating system. Currently ROMP runs on x86_64 platforms. Any x86_64 platform that runs Linux and supports execution of OpenMP programs should be supported.
- *Software dependencies.* One needs to install several software listed below to compile and run romp. Installation details are listed on https://github.com/zygyz/romp/blob/master/README.md
  1) Clang version 8.0.0 or later version (https://github.com/llvm-mirror/clang)
  2) elfutils-0.173 (https://sourceware.org/elfutils/)
  3) DynInst (https://github.com/dyninst/dyninst)
  4) LLVM OpenMP runtime library. Note: ROMP requires several OMPT callbacks that have not yet been modified/implemented in the official LLVM OpenMP repository. For that reason, one should use the llvm-openmp library provided at https://github.com/zygyz/romp/tree/master/pkgs-src/llvm-openmp/openmp
  5) tcmalloc (https://github.com/gperftools/gperftools)

2) Installation: Installation instructions can be found at https://github.com/zygyz/romp/blob/master/README.md

3) Evaluation Workflow: One can evaluate romp with different benchmark suites. Detailed instructions can be found on https://github.com/zygyz/romp/blob/master/README.md

1) *DataRaceBench.* We provide DataRaceBench at https://github.com/zygyz/romp/tree/master/tests/dataracebench . To enable automatic evaluation of romp, we modified the test-harness.sh script provided in the original source. To conduct the evaluation, one can invoke `./check-data-races.sh --romp`
2) *OmpSCR.* We provide OmpSCR at https://github.com/zygyz/romp/tree/master/tests/OmpSCR_v2.0 . We fixed two missing initialization bugs in cpp_qsomp3 and cpp_qsomp4 that cause the original programs to crash. To conduct the evaluation, one should first compile all the benchmark programs. And then instrument each program with DynInst client `omp_race_client` in https://github.com/zygyz/romp/tree/master/pkgs-src/dyninst-client .

4) Evaluation and Results:

1) *DataRaceBench.* check-data-races.sh will generate csv files stored in a sub-directory called results. A detailed description can be found at https://github.com/zygyz/romp/tree/master/tests/dataracebench
2) *OmpSCR.* To evaluate the run time and memory overhead, one can invoke `/usr/bin/time -f "%E %M"` .