Dynamic Data Race Detection for OpenMP Programs

Yizi Gu   John Mellor-Crummey

Department of Computer Science
Rice University

RICE

15 November 2018

Data Race

- A bug in a concurrent program
  - two concurrent accesses to the same memory location
  - at least one of the memory accesses is a write

- No constraint on interleaving of instructions

- Unpredictable program behavior

OpenMP

- **Shared memory parallel programming model**
- **Widely used node-level programming model on today's supercomputers**
  - **typically used to accelerate calculations within an MPI rank**
- **Supports multiple styles of parallelization, e.g.**
  - **threads (parallel regions)**
  - **tasks (implicit and explicit)**
  - **work sharing (parallel loops, …)**
- **Multiple synchronization constructs**
  - **barriers**
  - **reductions**
  - **task dependences**
  - **task wait**
  - **locks and critical sections**

Motivation

- Data races are hard to debug manually
  — attaching a debugger or adding print statements biases instruction interleavings

- Automatic data race detection tools are desired

- Existing tools are not accurate, e.g.
  — false negatives for loops that carry a dependence
  — false positives for ordered critical sections

Related Work

- **Dynamic data race detection**
  - — cilkscreen for Cilk [Feng & Leiserson SPAA ’97; Cheng et al. SPAA’98]
  - — hybrid data race detection algorithm [O’Callahan et al. PPoPP’03]
  - — techniques for happens-before analysis
    - – reachability queries in DAG [Agrawal et al. SIAM’18]
    - – exploit synchronization properties to be efficient, e.g.
      2D-DAGs [Xu et al. PPoPP’18]
- **Data race detection for OpenMP programs**
  - — static analysis
    - – Polyhedral analysis [Ye et al. Correctness’18@SC]
  - — tools
    - – Archer [Atzeni et al. IPDPS’16]; Sword [Atzeni et al. IPDPS’18]
  - — benchmark suite
    - – DataRaceBench [Liao et al. SC’17]

Contributions

- **ROMP - a per-input accurate data race detector for abelian OpenMP programs atop the OMPT first-party tools API**
  - **https://github.com/zygyz/romp**

- **Extensions to OpenMP's OMPT API**
  - **additional callbacks needed by a precise race detector**

- **An experimental evaluation of our tool**

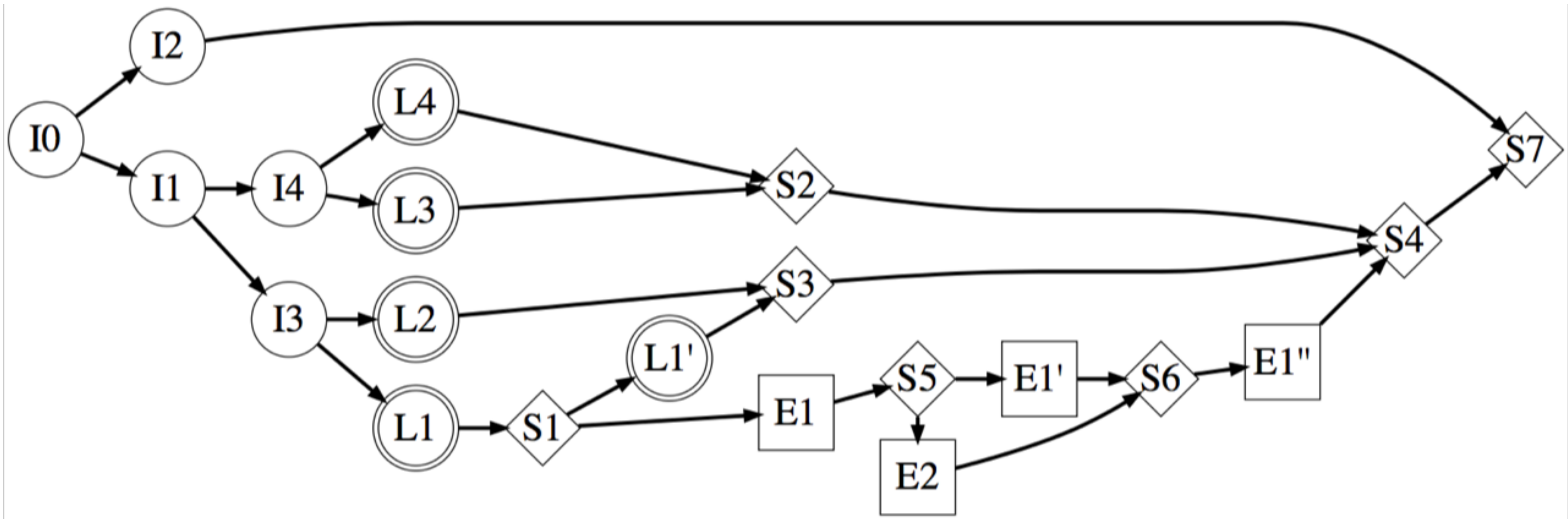
Approach

- **Assign labels to OpenMP tasks to reason about orderings**

- **Reason about concurrency between tasks**
  - — multiple concurrent tasks could be executed by the same physical thread in serial order
  - — logical concurrency exists regardless of thread schedule

- **Maintain access histories for shared variables**
  - — check if a load or store is a data race with prior accesses
  - — record information as needed

OpenMP Task Graph: 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
         }
     }
}
```

OpenMP Task Labeling

- **An OpenMP task label consists of *k* label segments**
  - create new task label upon task creation
  - *k* = nesting level of OpenMP tasks

- **Update task labels at OpenMP synchronization points**
  - rely on OMPT callback APIs
  - update fields in task labels to reflect synchronization operations

- **Reason about orderings by comparing task labels**
  - compare label segments in two task labels
  - O(*k*) time complexity

Task Label Segment

Field Name	Description
offset	relative id of the worker thread in the team
span	total number of worker threads in the team
iteration id	relative id of the iteration in the work-share loop (if applicable)
taskwait count	number of `taskwait` encountered by current task
taskcreate count	number of explicit tasks created by current task
loop count	number of work-share loops finished by current task
phase	number of times current task entering/exiting ordered critical
task waited	true if the task is waited by parent
task group info	encode task group order info
segment type	implicit/explicit/logical

Label Updating Rules — I

- **creating implicit tasks**

Input : Parent task label: L_p; Local Thread ID: 0; Team Size: s
Output: A task label L for the newly created implicit task T

```
S ← CreateNewLabelSegment()
S.segment_type ← implicit
S.offset ← 0
S.span ← s
L ← L_p.copy()
L.append(S)
return L
```

- **creating explicit tasks**

Input : Parent task label: L_p;
Output: A task label L for the newly created explicit task T

```
S ← CreateNewLabelSegment()
S.segment_type ← explicit
S.offset ← 0
S.span ← 1
L ← L_p.copy()
L.append(S)
L_p.last_segment.task_create_cnt += 1
return L
```

Label Updating Rules — II

- **creating logical tasks**

Input : Parent task label: L_p; Iteration ID: i
Output : New task label L

L ← DiscardLastSegment(L_p)
S ← CreateNewLabelSegment()
S.segment_type ← logical
S.offset ← 0
S.span ← 1
S.iteration_id ← i
L.append(S)
return L

- **encountering barriers**

Input : Task label L of task encountering the `barrier` directive
Output : Modified task label L'

S ← L.last_segment
L' ← DiscardLastSegment(L)
L'.last_segment.offset ← L'.last_segment.offset + L'.last_segment.span
L'.append(S)
return L'

Label Updating Rules — III

- **encountering taskwait**

Input : Task label L of task encountering the `taskwait` directive
Output : Modified task label L'

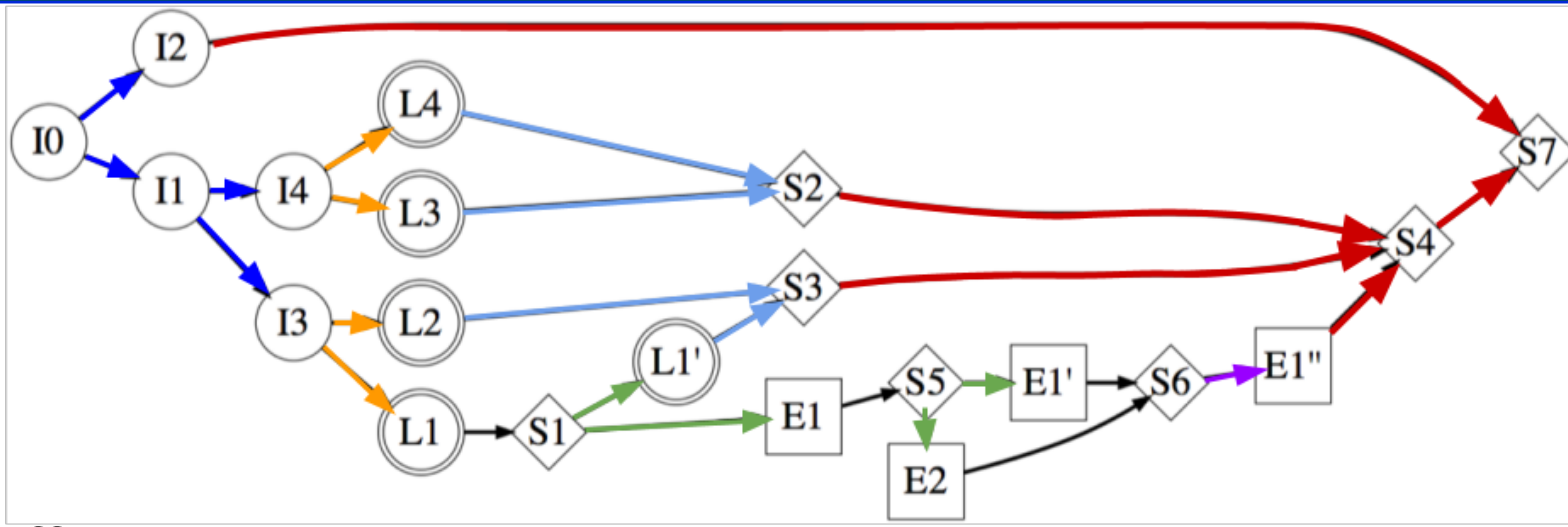
$L' \leftarrow$ L.copy()
L'.last_segment.taskwait_cnt $\leftarrow L'$.last_segment.takwait_cnt + 1
NotifyExplicitChildrenTaskWait();
return L'

- **end of work-share loop**

Input : Task label L of task exiting work-sharing loop construct
Output : Modified task label L'

$L' \leftarrow$ DiscardLastSegment(L)
L'.last_segment.loop_cnt $\leftarrow$ L'.last_segment.loop_cnt + 1
return L'

Task Labeling: Example



- [offset, span, iter-id, tw, tc, lp, phs, twtd, tg, type]
  - I0: [0,1,{0},imp]
  - I1: I0[0,2,{0},imp]
  - I3: I1[0,2, {0}, imp]
  - I4: I1[1,2,{0},imp]
  - L1: I3 [0,4,{0},lgc]
  - L1’: I3 [0,4, {tc:1}, lgc]
  - E1: L1[0,1,{0},exp]
  - E1’: L1[0,1,{tc:1},exp]
  - E2: E1[0,1, {0}, exp]
  - E1”: L1[0,1,{tc:1,tw:1}, exp]
  - I4 after S2: I1[1,2,{lp:1},imp]
  - I4 after S4: I0[2,2,{0},imp][1,2,{lp:1},imp]

Benefits of OpenMP Task Labeling

- **Decentralized reachability query**
  - — conduct query by comparing two labels
  - — multiple queries can execute in parallel

- **Concisely represents logical concurrency in OpenMP constructs**
  - — iterations in a work-sharing loop construct as logically concurrent tasks
  - — synchronization encoded in task labels

Per-Task Metadata

- Problem: some synchronization is unsuited for task labels
- Approach
  - use OMPT callbacks to notify ROMP of synchronization
  - store sync info in data structures associated with each task
- Examples
  - explicit task dependences
    - task dependence graph built according to dependence variables
    - search directed path in the graph for ordering relation
    - only inspected if no order can be determined using task labels
  - reductions
    - monitor entry/exit of reduction region by OMPT callback
  - critical/atomic sections
    - maintain a set of locks held during each memory access

A Hybrid Data Race Detection Algorithm

- **Invoke this algorithm to check if an access races with others**
- **When to prune accesses?**
  - **— current access ordered with a prior access**
  - **— one access performed with fewer locks**
    - **– discard an access record in the history**
    - **– don't add a record for the current access**

Input : Memory address being accessed: l
Current access record <ϵ,a,h>
History of accesses: $History$
Output: Report data race on l if current access is racing with history accesses

skip_current $\leftarrow$ FALSE
foreach <ϵ', a', h'> $\in$ *History[l]* **do**
  if $\epsilon \parallel \epsilon' \wedge h \cap h' = \emptyset \wedge (a = w \vee a' = w)$ **then**
    report a data race
  if $((a' = w \wedge a = w) \vee a' = r) \wedge h' \supseteq h \wedge \epsilon' \rightarrow \epsilon$ **then**
    History[l] $\leftarrow$ History[l] - <ϵ', a', h'>
    continue

  if $((a' = r \wedge a = r) \vee a' = w) \wedge h \supseteq h' \wedge \epsilon' \Rightarrow \epsilon$ **then**
    skip_current $\leftarrow$ TRUE

if *skip_current is FALSE* **then**
  History[l] $\leftarrow$ History[l] $\cup$ <ϵ, a, h>

- **Adapts All-Sets algorithm for Cilk to enable parallel execution [Cheng et al. SPAA'98]**

Data Environment Tracking: Motivation

- **A task may use storage on the stack and in the heap**
- **When a task finishes, it releases its storage**
- **Later a logically concurrent task could reuse that storage**
- **At run time we must track such information so that we don't report this as a data race**

Data Environment Tracking: Method

- **Maintain shared/private state for each memory cell**
- **Mark a memory cell as private if**
  - — **its address falls within range of stack of accessing thread**
  - — **its address is below the accessing task's base**
- **Mark a memory cell as shared if**
  - — **its address falls out of range of stack of accessing thread**
  - — **accessed by exp. tasks, its addr. falls out of task private region**
- **Mark memory cells as deallocated upon task schedule point**
  - — **implicit tasks are bound to physical threads, private variable is deallocated when finishes**
  - — **for explicit tasks, mark as deallocated upon task schedule**

Correctness

- Abelian property of a parallel program [Cheng et al. SPAA'98]
  - a program is abelian if
    - any critical sections protected by the same lock commute
    - thread schedule does not affect control flow
    - all locks are properly nested
- Provided that a program is abelian and does not use SIMD construct
  - if data races exist with respect to a shared variable, ROMP reports at least one
    - no need to report all instances of the same data race
  - if no data race exists with respect to a shared variable, ROMP does not report one

Extensions to OMPT APIs

- **ompt_callback_reduction**
  - notify tools of entering/exiting reduction region
- **ompt_callback_dispatch**
  - notify tools of dispatching of a logical task in a work-sharing construct
- **ompt_get_task_memory**
  - get the range of memory allocated for a task

Shadow Memory

- **Store access histories**

- **Two level page table**
  - — **similar to Valgrind [N.Nethercote et al. VEE'07]**

- **Use locks to protect concurrent accesses to shadow memory**
  - — **when checking multiple concurrent access to the same location**
  - — **mutual exclusion to avoid data races when maintaining shadow memory**

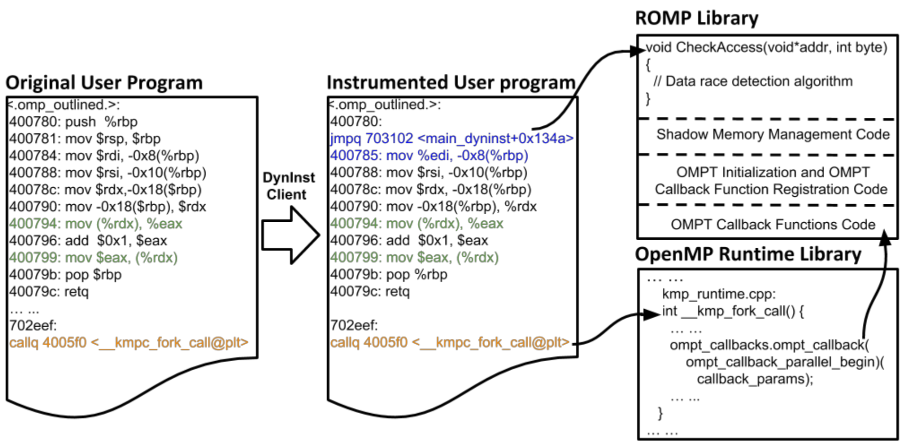
Optimization

- **Don’t check accesses to read-only data**

- **Dynamic checking shortcut**
  — **lock contention when accessing shadow memory**
    - **both checking procedures run in parallel**
    - **both check accesses to the same memory location**
    - **at least one of the checks is for a write access**
  — **directly report the data race**
    - **no need to invoke the checking protocol**

Implementation

- **Implement ROMP functionality as a library**
- **Use a binary instrumentation tool to insert a call to ROMP to check if a load or store is involved in a data race**
  - **— DynInst [CS@UW-Madison]**
- **Maintain access histories in shadow memory associated with each program variable**



Evaluation Metrics

- **Recall: TP / (TP + FN)**
  - — fraction of real races reported
- **Precision: TP/(TP + FP)**
  - — fraction of races reported that are real
- **Accuracy: (TP + TN) / (TP + TN + FP + FN)**
  - — correct check results / total checks

Evaluation

- System: single-node, four 12-core AMD Opterons, 128GB of memory

- Evaluate accuracy
  - compare ROMP with Archer using DataRaceBench
  - run test script provided by DataRaceBench
    - modified test script to add option for testing ROMP
  - not including test cases for containing SIMD constructs
    - limitation of dynamic data race detectors

- Evaluate performance
  - compare ROMP with Archer using OmpSCR

Results: Accuracy

ROMP is more accurate than the best prior race detector for OpenMP programs

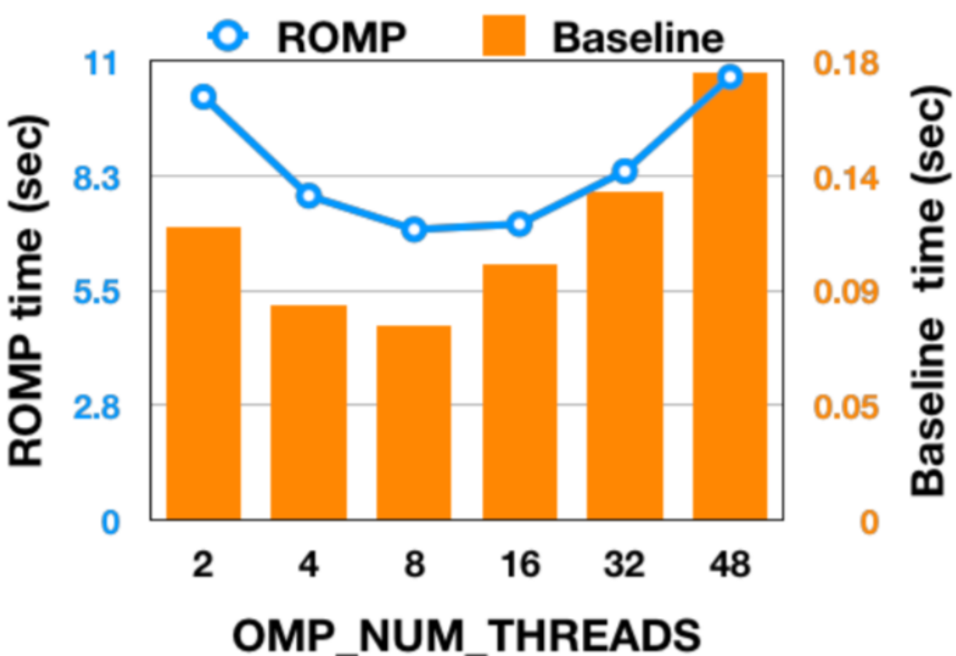
	Archer	ROMP
Precision	0.79	1.0
Recall	0.79	1.0
Accuracy	0.78	1.0

Results: Performance

	Run time Overhead		Memory Overhead	
	Archer	ROMP	Archer	ROMP
Mean	130	91.5	39.2	23.1
Median	8.70	8.23	18.0	3.69
Geo-Mean	18.6	20.5	18.3	7.28

- Run time overhead
  - Archer is 10% faster on OmpSCR
- Space overhead
  - ROMP uses 2.5x less space on OmpSCR

Results: Scalability



A parallel quick sort program
Input size: 1 million

Input size	Time overhead
1k	2.7x
2k	2.1x
4k	7.7x
8k	5.9x
16k	11x
32k	21x
64k	27x

A parallel quick sort program
Number of threads: 16

Summary and Ongoing Work

- **Contributions**
  - **ROMP: a per-input accurate race detector for abelian OpenMP programs**
    - **more precise than prior OpenMP race detectors**
  - **experimental evaluation**
    - **ROMP uses comparable time and less space than Archer**
  - **extensions to OpenMP's OMPT API for precise race detection**
- **Ongoing work**
  - **instrument all dynamic libraries**
  - **reduce overhead and improve scalability**
    - **prune long access histories for widely-shared data**
      - **current pruning criteria are correct but conservative**
      - **exploit properties of OpenMP parallel constructs to prune more access records: keep only necessary rather than sufficient records**