Static and Dynamic Data Race Detection for Multithreaded Environments

Alperen Görmez
Thomas McCarthy
ECE 566 - Parallel Processing
University of Illinois at Chicago
Spring 2022

Outline

- Introduction to data races
- Static data race detection
- Dynamic data race detection

What causes a data race?

A data race occurs when:

- Two or more threads in a **single process** access the same memory location concurrently
- At least one of these threads is performing a write operation
- Threads are not using correct synchronization

Example

- Serial output = 100

- Parallel output = ???

```
int data_race () {
    int data = 0;

    #pragma omp parallel for
    for(int ii=0; ii<100; ii++) {
        data = data + 1;
    }

    return data;
}
```

Why do we care?

- Non-determinism in execution

- Torn read/write operations on some physical systems
  - Nonsensical combination of the data from the two operations
  - Can cause problems if used as control data
    - Crashes
    - Security Issues

How to prevent data races?

- Simple, use synchronization!
- Synchronization ensures that only one thread can affect the shared variable at a time.
- Other threads must wait to interact with the shared variable

```
int fixed_data_race () {
    int data = 0;

    #pragma omp parallel for
    for(int ii=0; ii<100; ii++) {
        #pragma omp critical
        {
            data = data + 1;
        }
    }

    return data;
}
```

New problem: Synchronization is expensive!

- Synchronization constructs enforce correct execution, but are expensive
  - Overhead of primitives
  - Less code can be parallelized

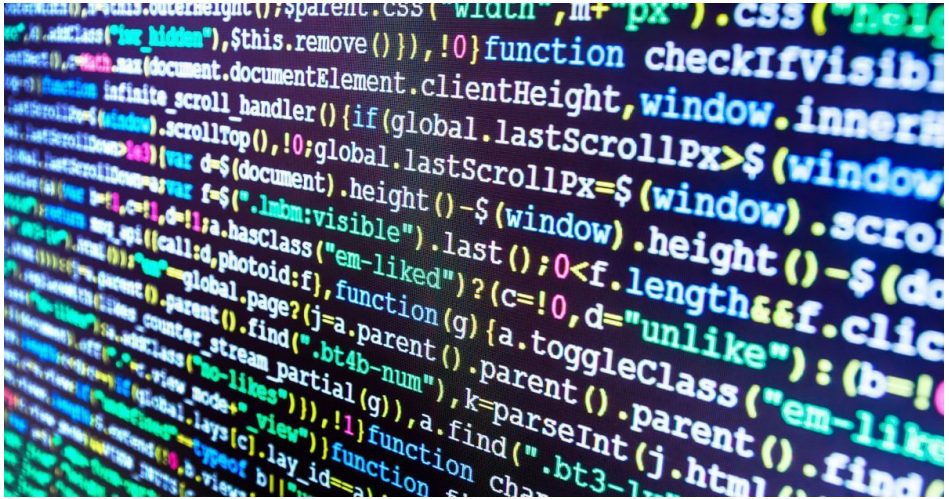
```
int bad_code (int* a, int* b, int* c, int len) {
    #pragma omp parallel for
    for(int ii=0; ii<len; ii++) {
        #pragma omp critical
        {
            c[ii] = a[ii] + b[ii];
        }
    }
    return data;
}
```

- Should **only** be used when necessary, nowhere else

New problem: Hard to find data races in real code

- Real code is complex, with thousands of lines with hundreds of variables

- Data race detection: methods and techniques to find data races in real life code



Static Data Race Detection

- Examination of the actual code for data races
  - Done at/before compilation of code
  - Good for detecting data races from incorrect locking
  - Struggles with timing related data races

- Three main steps:
  - Discover shared variables
  - Lock set analysis
  - Warning reduction

Step 1: Discover shared variables

- Shared variable: variable that can be accessed by multiple threads at **concurrent** points in their execution

- Types of shared variables:
  - Global variables (and their aliases) of threads
  - Pointers passed as parameters to API functions
  - Escape variables

```
int shared_vars(int arg_cnt) {
    int data = 0;

    #pragma omp parallel for
    for(int ii=0; ii<arg_cnt; ii++) {
        int local_data = ii;
        #pragma omp critical
        {
            data = data + local_data;
        }
    }

    return data;
}
```

Step 1: Discover shared variables

- Only variables of these types that are **written to** can potentially have data races

- All data races happen at accesses to elements of this subset of shared variables

```
int read_only_global(int* array, int arg_cnt) {
    int data = 0;

    #pragma omp parallel for
    for(int ii=0; ii<arg_cnt; ii++) {
        int local_data = array[ii];
        #pragma omp critical
        {
            data = data + local_data;
        }
    }

    return data;
}
```

Step 1: Discover shared variables

- Note: global variables can be accessed by local pointers

- Need to track which local pointers are in fact pointing to global variables

```
int local_pointers(int arg_cnt) {
    int data = 0;

    #pragma omp parallel for
    for(int ii=0; ii<arg_cnt; ii++) {
        int* local_ptr = &data;
        int local_data = ii;
        #pragma omp critical
        {
            *local_ptr = *local_ptr + local_data;
        }
    }

    return data;
}
```

Step 1: Discover shared variables

- Note: Only **part** of a global structure can be shared data, while the rest is not

- Track the shared/global status of each field separately

```
struct summed_array {
    int sum;
    int size;
    int* array;
};

void partial_shared_struct(summed_array* SA) {
    #pragma omp parallel for
    for(int ii=0; ii<(SA->size); ii++) {
        int local_data = SA->array[ii];
        #pragma omp critical
        {
            SA->sum = SA->sum + local_data;
        }
    }

    return
}
```

Step 2: Lockset analysis

- Once the potential places for data races have been found, the synchronization constructs used at each position need to be checked to see if correct synchronization already exists

- Locks are the most common form of synchronization construct used in parallel thread programs
  - Only one thread can hold a lock at a given time
  - Other threads must wait for the lock to be released before acquiring it

Set 2: Lockset analysis

- For each control location that could produce a data race, the set of all locks held by a thread at that location is calculated

- The sets of all locations that access the same variable must intersect to form a non-empty set

```
void locksets() {
    int data1 = 0;
    int data2 = 0;
    int data3 = 0;

    omp_lock_t L1, L2, L3;
    omp_init_lock(&L1);
    omp_init_lock(&L2);
    omp_init_lock(&L3);

    #pragma omp parallel for
    for(int ii=0; ii<100; ii++) {
        omp_set_lock(&L1);
        data1 = data1 + ii;
        omp_unset_lock(&L1);
        omp_set_lock(&L2);
        data2 = data2 + (ii*ii);
        omp_set_lock(&L3);
        data3 = data3 + (ii*ii*ii);
        omp_unset_lock(&L2);
        omp_unset_lock(&L3);
    }

    return (data1+data2+data3);
}
```

Step 2: Lockset analysis

- Problem: Most locks are passed to functions as local pointers. The values of these local pointers (and thus which locks they represent) are dependent on the parent function.

- Similar to shared variables, dataflow analysis is necessary to find which local pointers point to the same lock and prevent false positives.

```
void main() {
    int data;
    omp_lock_t lock;
    omp_init_lock(&lock);

    #pragma omp parallel for
    for(int ii=0; ii<100; ii++) {
        parallel_alg_1(&data, &lock, ii);
        parallel_alg_2(&data, &lock, ii);
    }
}

void parallel_alg_1(int* data, omp_lock_t* lock, int input) {
    int local_data = work1(input);
    omp_set_lock(lock);
    *data = *data + local_data;
    omp_unset_lock(lock);
}

void parallel_alg_2(int* data, omp_lock_t* lock, int input) {
    int local_data = work2(input);
    omp_set_lock(lock);
    *data = *data + local_data;
    omp_unset_lock(lock);
}
```

Step 3: Warning reduction

- Steps 1 and 2 are guaranteed to produce warnings for all of the data races produced by incorrect locking. However, they are **NOT** guaranteed to only identify actual data races. Some of the warnings are false positives.

- Adding additional synchronization for these false positives will slow down program execution without any benefit. As such, these false positives should be removed from the warning list.

Step 3: Warning reduction - Locking patterns

- Data races can only happen in **concurrent** portions of code

- Even if two potential data race locations have disjoint lock sets, the pattern of lock set acquisition can make concurrency impossible.

```
Thread 1:       Thread 2:
1: lock(L3)     1: lock(L3)
2: lock(L1)     2: lock(L2)
3: a = 1        3: c = 4
4: lock(L2)     4: lock(L1)
5: unlock(L3)   5: unlock(L3)
6: b = 2        6: d = 5
7: unlock(L2)   7: unlock(L1)
8: x = 3        8: x = 6
9: unlock(L1)   9: unlock(L2)
```

Step 3: Warning reduction - Locking patterns

- To find which positions can be reached concurrently, first find the set of locks set (and possibly unset) after each lock held at all positions

P	Lock Held	P(Lock Held)
T1/8	L1	L2
T2/8	L2	L1

```
Thread 1:       Thread 2:
1: lock(L3)     1: lock(L3)
2: lock(L1)     2: lock(L2)
3: a = 1        3: c = 4
4: lock(L2)     4: lock(L1)
5: unlock(L3)   5: unlock(L3)
6: b = 2        6: d = 5
7: unlock(L2)   7: unlock(L1)
8: x = 3        8: x = 6
9: unlock(L1)   9: unlock(L2)
```

Step 3: Warning reduction - Locking patterns

- Two positions P_1 and P_2 are **NOT** concurrent if there exists two locks L_1 and L_2 such that L_1 is in $P_1(L_2)$ and L_2 is in $P_2(L_1)$

L_1 = L2
L_2 = L1

P	Lock Held	P(Lock Held)
T1/8	L1	L2
T2/8	L2	L1

```
Thread 1:       Thread 2:
1: lock(L3)     1: lock(L3)
2: lock(L1)     2: lock(L2)
3: a = 1        3: c = 4
4: lock(L2)     4: lock(L1)
5: unlock(L3)   5: unlock(L3)
6: b = 2        6: d = 5
7: unlock(L2)   7: unlock(L1)
8: x = 3        8: x = 6
9: unlock(L1)   9: unlock(L2)
```

Static data race detection - End

- Once all possible data race locations have been found and as many false positives as possible have been thrown out, synchronization constructs can be added to the remaining locations to ensure correct program execution

Dynamic data race detection

- Employs a tracing mechanism to see if a particular run of the program caused a data race
- On the fly: stores partial trace information for detecting as data race occurs
- Less false alarms compared to static detection

Drawbacks of dynamic data race detection

- For t threads of n instructions each, number of possible orders is around t^{nt}
- Feasible data races: data races we could see in any execution
- Exactly locating feasible data races is NP hard
- Apparent data races: approximations of feasible data races based on synchronization behavior in an execution
- Locating all apparent races is NP hard

Drawbacks of dynamic data race detection

- Checks a particular run. Not possible to detect all possible data races. Needs to check all possible runs.

- Overhead

- Trade-off: low overhead leads to missing more data races

Detection algorithms

- `S:` synchronization object
- `release/unlock(S):` thread releases S
- `acquire/lock(S):` thread acquires S
- `barrier(S):` all threads release S as they reach the barrier, then acquire

Detection algorithms

1. DJIT+

- Execution of threads is split into sequence of time frames
- Each thread `t` has vector (vector clock, VC) of time frames st_t
- For each index `u`, $st_t[u]$ stores latest local time frame of thread `u`, whose release operation is known to `t`
- Recording the clock of the most recent write to each variable `S` by each thread `t`
- `release(S)` starts a new time frame
- Access history of shared location `v` has ar_v and aw_v
- Based on happens-before relationship

Detection algorithms

1. DJIT+

- Initialization
  - $\forall i\ st_t[i] \leftarrow 1$
  - $\forall i\ st_S[i] \leftarrow 0$
  - Access history of shared location `v`: $\forall i\ ar_v[i] \leftarrow 0$, $\forall i\ aw_v[i] \leftarrow 0$
- `release(S)`
  - $st_t[t] \leftarrow st_t[t] + 1$
  - $\forall i\ st_S[i] \leftarrow \max(st_t[i], st_S[i])$
- `acquire(S)`
  - $\forall i\ st_t[i] \leftarrow \max(st_t[i], st_S[i])$

Detection algorithms

1. DJIT+

- Access to shared location
  - $ar_v[t] \leftarrow st_t[t]$ or $aw_v[t] \leftarrow st_t[t]$
  - Checking for data race:
    - If read: check if another thread u wrote to v and $aw_v[u] \geq st_t[u]$. This means t checks if it knows a release that preceded the write in u
    - If write: check if there exists another thread u such that $aw_v[u] \geq st_t[u]$ OR $ar_v[u] \geq st_t[u]$. This means t checks all reads and writes by other threads to v.
- Does not imply the program is race free

Detection algorithms

2. Lockset

- Time frames idea, with locks
- Checks a sufficient condition for data-race-freedom
- Locking discipline: each shared location is protected by the same lock on each access to it
- All accesses are made while holding the lock

Detection algorithms

2. Lockset

- How to know which lock protects what?
- Infer

```
AcquireLock( A );
AcquireLock( B );
x ++;
ReleaseLock( B );
ReleaseLock( A );


AcquireLock( B );
AcquireLock( C );
x ++;
ReleaseLock( C );
ReleaseLock( B );
```

Erickson

Detection algorithms

2. Lockset

- Candidate set `C(v)` for each shared location `v`: set of all locks that could potentially be protecting `v` so far
- `locks_held(t)` for each thread `t`: set of locks currently held by `t`

Detection algorithms

2. Lockset

- Initialization:
  - ∀ `v`, initialize `C(v)` to the set of all possible locks
- Upon access to `v` by `t`
  - `lh ← locks_held(t)`
  - When acquired, `lh ← lh ∪ {lock}`
  - When released, `lh ← lh - {lock}`
  - `C(v) ← C(v) ∩ lh`
  - If `C(v) = ∅`, race warning is issued.

- If a shared location is accessed when `C(v)` is empty, locking discipline is violated.

Detection algorithms

2. Lockset

- No warnings = the algorithm guarantees that no data race is present on the current execution
- However, warning does not imply a data race

Detection algorithms

- Lockset cannot distinguish between real race and false alarm. It is too strict.
- DJIT+ detects only the ones that occurred
- Every data race is a violation of the locking discipline: it can be said that Lockset and DJIT+ detect the superset and subset of the races
- Number of checks in DJIT+ can be reduced if additional information from Lockset is used
  - If current access to v does not empty $C(v)$, it means v is still protected and we can be sure this access will not result in a data race
- Storage overhead as well as runtime: O(n)

Multirace: a logging mechanism

- Shared locations = minipages
- Accessing a minipage through a wrong view will result in page fault
- Each minipage can be referenced through the following views:
  - NoAccess: will catch every access to it
  - ReadOnly: will catch writes
  - ReadWrite: no page faults
- Multirace = hybrid DJIT+ Lockset

DataCollider: near zero overhead

- A runtime tool for finding data races
- Low runtime overheads
- Can successfully find many errors in Windows kernel, Windows Shell, IE, SQL server…
- Previous work: log data and synchronization operations at runtime, infer conflicting accesses using happens-before (DJIT+) or lockset reasoning

DataCollider: near zero overhead

- Challenges:
  - Large runtime overhead. Intel Thread Checker has 200x overhead
  - Complex synchronization semantics: Synchronizations can be complex. Can result in false data races.
  - Recording the state of the program is expensive for logging methods

DataCollider: near zero overhead

- Key ideas:
  - Use random sampling for accesses as data race candidates. User controlled overhead.
- Algorithm:
  - Randomly sprinkle code breakpoints on memory accesses
  - When a breakpoint fires at an access to `x`, delay for a small time window
  - Read `x` before and after the time window
  - Ensure a user-defined # of code breakpoint firings/s
- Sampling
  - Trade-off: overhead vs likelihood of finding a data race

DataCollider: near zero overhead

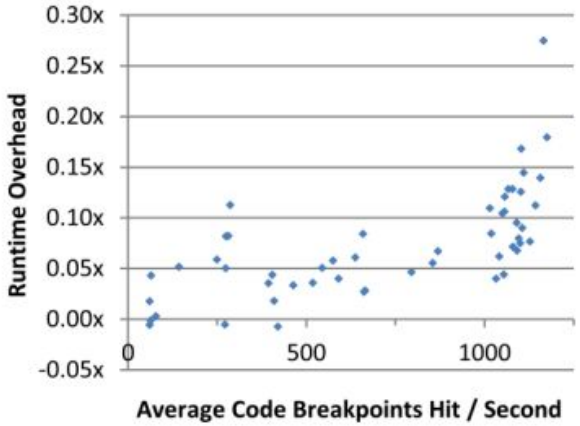


Figure 5: Runtime overhead of DataCollider with increasing sampling rate, measured in terms of the number of code breakpoints firing per second. The overhead tends to zero as the sampling rate is reduced, indicating that the tool has negligible base overhead.

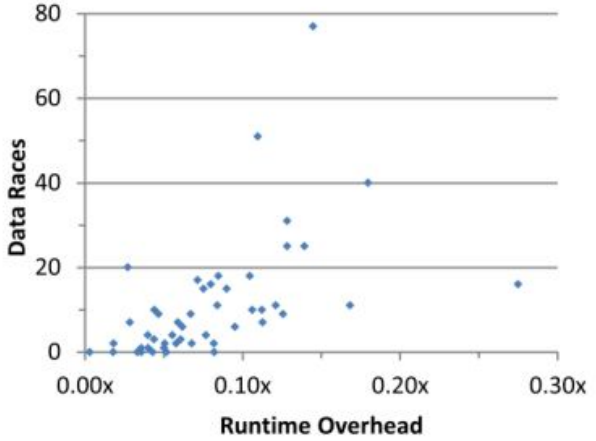


Figure 6: The number of data races, uniquely identified by the pair of racing program locations, with the runtime overhead. DataCollider is able to report data race even under overheads under 5%

DataCollider: near zero overhead

- Puts the user in control of the overhead
- Incapable of false data races
- Trivial to implement, requires no knowledge of synchronization methods
- Provides full debugging information

References

1. Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A. (2007). Fast and Accurate Static Data-Race Detection for Concurrent Programs. In: Damm, W., Hermanns, H. (eds) Computer Aided Verification. CAV 2007. Lecture Notes in Computer Science, vol 4590. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-73368-3_26
2. Pozniansky, Eli, and Assaf Schuster. "MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs." *Concurrency and Computation: Practice and Experience* 19.3 (2007): 327-340.
3. Erickson, John, et al. "Effective Data-Race Detection for the Kernel." *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010.
4. Erickson, John, et al. “Dynamic Analyses for Data Race Detection.” [Slideshow]. Microsoft.