Detecting Data Races in Parallel Programs

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@cs.rice.edu
• Happens-before


• Lock covers

Outline

• Defining the problem
• Detecting data races with happened-before
• Detecting data races with lock covers: Eraser
• Discussion
Data Races

• Logically concurrent accesses to the same variable
  —no happens-before ordering

• At least one is a write

• No mechanism used to prevent simultaneous accesses
  —e.g., lock, atomic variable

• Notes
  —data race causes non-determinism, but is not synonymous
  —here data race = atomicity violation
The Problem

• Domain
  —shared-memory programs with nested fork-join parallelism
    – e.g. Cilk programs, nested parallel loops in OpenMP

• The challenge
  —schedule-dependent bugs: unpredictable erroneous behavior on some executions with a particular input data set

• Principal cause
  —unsafe communication through shared variables
    – “data races,” “access anomalies,” “determinacy races”
Fork-Join Parallelism

Nested parallel loops

... DOALL I=2,4 ...
...
IF (I.EQ.2) THEN
  DOALL J = 1,2 ...
  ENDDO
ENDIF
DOALL J=1,I ...
  ENDDO ...
ENDDO ...

...
Data Race Example

```
DOALL I=1,2
    PRIVATE T
    T=SUM
    SUM=T+I
ENDDO
PRINT SUM
```
Difficulty of Isolating Data Races

- Different executions may exhibit different behavior
  — behavior depends upon access interleaving order
- Adding tracepoints or breakpoints may cause erroneous behavior to disappear
  — alters probable interleavings
Approaches for Detecting Data Races

- **Static analysis**
  - examine program text only
  - conservative approximations (parallelizing compilers)
    - e.g. data dependences carried on loops
  - drawback: false positives

- **Post-mortem analysis**
  - examine execution traces
  - report only actual races (Intel thread checker)
  - drawback: large trace logs

- **On-the-fly analysis**
  - during execution
  - report only actual races without trace logs
  - drawback: potentially high run-time overhead
On-the-fly Method: Access Summaries

Making Asynchronous Parallelism Safe for the World
Guy Steele POPL 1989

- Detect the presence of races

- Protocol sketch
  - cell’s marked with state of nest of threads that accessed them
    - \<(p1, e1), (p2, e2) \ldots (pm, em) >
    - either \(p_i\) or \(e_i\) may be a *: more than one thread or operation type
  - access \(e\) on a “cell” by nested thread \(p_1, p_2, \ldots, p_k\)
    - “augment” cell marks with access \<(p1, e), (p2, e) \ldots (pk, e)>
  - idea: use bitvector encoding \([p1][e1][p2][e2]\ldots[pj][ej]\)
    - assign a distinct “k of m” bit-pattern to parallel siblings
    - update access history with bitwise-OR
  - maintain accessed cells in thread “responsibility set”
  - turn over “responsibility set” to parent when thread finishes
    - prune cell access state to bitvector reflecting current nesting depth
    - detect races at history pruning time
      - check access history for violations by finding illegal bit pattern
        - both \(p_j\) and \(e_j\) have more than \(k\) bits set

- Drawback: unable to pinpoint endpoints
On-the-fly Method: Access Histories

Pinpoint races present

• State
  — maintain information about thread concurrency
  — maintain an access history for each shared variable
    - contains (thread, access type) pairs

• When performing an access
  — use a protocol to augment an “access history” for the shared variable as necessary
  — determine if any of the accesses are conflicting
    - two or more accesses by concurrent threads
    - at least one is a write
to check access of type \( x \) to \( V \) by thread \( t_j \):

for each pair \((t_i, y)\) in \( V \)'s access history
  if Concurrentp\((t_i, t_j)\) and
    (Writep\((y)\) or Writep\((x)\)) then
      report a data race
  endif
  update \( V \)'s access history as necessary
endfor

• Difficulty: how many accesses must one store in a history?
  — can one store a list asymptotically shorter than \( O(T) \) for each
    variable and not miss any races?
  \( (T = \text{maximum amount of logical concurrency}) \)
Offset-span Labeling

• A thread’s offset span label requires space proportional to the nesting depth of the fork-join construct that spawned it.

• Description
  — each thread is labeled with a sequence of [offset, span] pairs that reflects its position in the graph.

• Labeling rules
  — initial thread gets [0,1]
  — a thread that is the i\textsuperscript{th} child spawned by an n-way fork of thread with label L gets label L [i,n]
  — a thread after a join that pairs with a fork of thread with label L[o,s] gets label L[o+s, s]
Offset Span Labeling Example
Offset-Span Labeling Protocol

- Access history per shared variable
  - last writer (last)
  - lowest “leftmost” reader (ll)
  - lowest “rightmost” reader (lr)

- checkread protocol
  - if concurrent with last writer, report RACE
  - update reader(s) in history as necessary

- checkwrite protocol
  - if concurrent with last writer or either reader, report RACE
checkwrite(access_history, thread_label)
    if access_history[^.W_{last}] \not\rightarrow_G^* thread_label then
        report a WRITE-WRITE data race
    endif
    if access_history[^.R_{1l}] \not\rightarrow_G^* thread_label or
        access_history[^.R_{1r}] \not\rightarrow_G^* thread_label then
        report a READ-WRITE data race
    endif
    access_history[^.W_{last}] := thread_label
end checkwrite
checkread(access_history, thread_label)
    if access_history^\cdot W_{last} \not\equiv^*_G thread_label then
        report a WRITE-READ data race
    endif
    if thread_label \prec_G access_history^\cdot R_{11} or
        access_history^\cdot R_{11} \Rightarrow_G thread_label then
        access_history^\cdot R_{11} := thread_label
    endif
    if access_history^\cdot R_{1r} \prec_G thread_label or
        access_history^\cdot R_{1r} \Rightarrow_G thread_label then
        access_history^\cdot R_{1r} := thread_label
    endif
end checkread
Principal Protocol Properties

- **Lemma WRITE-WRITE**
  - if any two writes are concurrent, then \texttt{checkwrite} will report a data race

- **Lemma READ-WRITE**
  - \texttt{checkwrite} will report a data race for any write that is logically concurrent with a temporally earlier read

- **Lemma WRITE-READ**
  - \texttt{checkread} will report a data race for any read that is logically concurrent with a temporally earlier write

- **Offset-span labeling theorem**
  - if an execution of a program with nested fork-join parallelism contains one or more data races, at least one will be reported
Lemma WRITE-WRITE

• If any two writes are logically concurrent, then checkwrite will report a data race

• Proof sketch:
  — access history contains last writer
  − checkwrite uses this to report a WRITE-WRITE race when two concurrent writes are temporally adjacent
  — if there are two concurrent writes to a shared variable, then there must be two temporally adjacent concurrent writes
  − proof by contradiction: if for all pairs of temporally adjacent writes, the first write of the pair sequentially precedes the second, then by transitivity none of the writes are concurrent
Data Race Detection Among Writers
Lemma READ-WRITE

- `checkwrite` will report a data race for any write that is logically concurrent with a temporally earlier read

- **Proof sketch**
  - an access history contains
    - the lowest leftmost reader
    - the lowest rightmost reader
  - a read write race would be missed if the present write was not concurrent with either of these reads, but was concurrent with some temporally earlier read
  - with the lowest leftmost and lowest rightmost readers, the join point is as low as possible, thus a READ-WRITE race will always be reported if any are present
Data Race Detection Example

Legend
Lowest Left Read
Lowest Right Read
Last Write

∃ X, = X
Data Race Detection Example

Legend
Lowest Left Read
Lowest Right Read
Last Write
Data Race Detection Example

Legend
Lowest Left Read
Lowest Right Read
Last Write
Data Race Detection Example

Legend
Lowest Left Read
Lowest Right Read
Last Write
Data Race Detection Example

Legend
Lowest Left Read
Lowest Right Read
Last Write
Data Race Detection Example

Legend
Lowest Left Read
Lowest Right Read
Last Write
Lemma WRITE-READ

• A data race will be reported if a read is logically-concurrent with some temporally earlier write

• Proof sketch
  — if the read is concurrent with the last write, checkread reports a WRITE-READ data race
  — if the read is concurrent with some earlier write, but not the last one, then there exists a pair of concurrent writes and Lemma WRITE-WRITE guarantees that a data race is reported
Thought Question

If we apply offset-span labeling in a single-threaded execution of a parallel loop, do we need all components of the access history?

— lowest leftmost reader
— lowest rightmost reader
— last writer
Offset-span Labeling Results Summary

Applicable to nested fork-join parallel programs

• No centralized bottleneck
  — labeling of each node can be determined locally from a single ancestor

• Constant length access history

• Time per access: $O(N)$, where $N$ is max fork-join nesting depth

• Space: $O(V + \min(VN, BN))$, where $B$ is total # threads
Efficient Detection of Determinacy Races

(SP-bags algorithm)

Feng and Leiserson
Review: Cilk Model

- **Parallel control** = `spawn`, `sync`, `return` from spawned function
- **Thread** = maximal sequence of instructions not containing parallel control (task in earlier terminology)

```cilk
int fib(n) {
    if (n < 2) return n;
    else {
        int n1, n2;
        n1 = spawn fib(n-1);
        n2 = spawn fib(n-2);
        sync;
        return (n1 + n2);
    }
}
```

- **Thread A:** if statement up to first spawn
- **Thread B:** computation of n-2 before 2nd spawn
- **Thread C:** n1 + n2 before the return
Principal Contributions

- **Shorter labels**
  - rather than having labels proportional in length to the maximum nesting depth, thread labels can have log V bits (a program constant)

- **Smaller space bound**
  - constant length labels
  - constant length access history per variable
  - O(V) space total

- **Lower time for checking concurrency**
  - check concurrency using Tarjan’s union-find $O(\alpha(v,v))$ time
    - $\alpha$ is Tarjan’s functional inverse of Ackermann’s function
Relative Contributions In a Nutshell

Compared to offset span labeling

- Reduced $O(VN)$ space to $O(V)$
- Reduced $O(N)$ time per access to $O(\alpha(v,v))$
- Drawback: their protocol is strongly sequential
  — can only be used to check for races within a sequential execution
SP-Bags Approach

- Fork-join parallelism as before
- Access history per variable
  - single-element reader
  - single-element writer
SP-Bags Protocol

Designed for the Cilk model

Each procedure F maintains two bags

- S-bag: set of thread IDs of F’s completed children that logically precede the current thread, as well as ID for F itself
- P-bag: set of thread IDs of descendants of F’s completed children that operate logically “in parallel” with the currently executing thread
SP-Bag Representation

Disjoint set union data structure

- \( \Sigma \), a dynamic collection of disjoint sets

- Operations
  - \texttt{Make\_set(x)}: \( \Sigma \leftarrow \Sigma \cup \{\{x]\} \)
  - \texttt{Union(X, Y)}: \( \Sigma \leftarrow \Sigma - \{X, Y\} + \{X \cup Y\} \)
  - \texttt{Find(x)}: returns the set \( X \in \Sigma \), such that \( x \in X \)

Tarjan shows that any \( m \) of these operations on \( n \) sets take a total of \( O(m \alpha(m, n)) \) time.
spawn procedure $F$:

$S_F \leftarrow \text{MAKE-SET}(F)$

$P_F \leftarrow \emptyset$

sync in a procedure $F$:

$S_F \leftarrow \text{UNION}(S_F, P_F)$

$P_F \leftarrow \emptyset$

return from procedure $F'$ to $F$:

$P_F \leftarrow \text{UNION}(P_F, S_{F'})$
**Figure 7**: A snapshot of the SP-bags data structures during the execution of a Cilk program. The ovals in the figure represent procedures that are currently on the runtime stack: $F_1$ spawns $F_6$, which spawns $F_9$, which spawns $F_{13}$. Each procedure contains an S-bag and a P-bag. Each descendant of a completed child of a procedure $F$ belongs either to $F$’s S-bag or to $F$’s P-bag. For example, $F_2$, $F_3$, $F_4$, and $F_5$ are descendants of $F_1$ that complete before $F_1$ spawns $F_6$, and so these procedures belong to either $F_1$’s S-bag or its P-bag. In addition, every procedure $F$ belongs to its own S-bag.
write a shared location \( l \) by procedure \( F \):

\[
\text{if } \text{FIND-SET}(\text{reader}(l)) \text{ is a P-bag}
\text{or } \text{FIND-SET}(\text{writer}(l)) \text{ is a P-bag}
\text{then } \text{a determinacy race exists}
\]

\[
\text{writer}(l) \leftarrow F
\]

read a shared location \( l \) by procedure \( F \):

\[
\text{if } \text{FIND-SET}(\text{writer}(l)) \text{ is a P-bag}
\text{then } \text{a determinacy race exists}
\]

\[
\text{if } \text{FIND-SET}(\text{reader}(l)) \text{ is an S-bag}
\text{then } \text{reader}(l) \leftarrow F
\]
How about the Slick Bounds?

- Let n be the number of spawned procedures
- Total number of make_set, union, find operations is at most T, serial running time
- $O(T \alpha(v,v))$ running time, where T is the serial running time
- $O(V + n)$ space
  - shadow space takes $O(V)$
  - disjoint set data structure takes $O(n)$
- Use garbage collection to reduce time and space bounds
  - run algorithm for V time steps
  - garbage collect
    - max of V thread ids can be in use
    - remove unused IDs from disjoint-set data structure
      - $O(v \alpha(v,v))$ time
    - algorithm needs only $O(V)$ space with garbage collection
Eraser

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Tom Anderson
Preventing Races with Monitors

• **Approach**
  — single anonymous lock
  — shared data accessed only through methods
  — methods must obtain lock

• **Benefits**
  — serialized access to monitor data
  — static, compile-time guarantee that no races exist

• **Drawbacks**
  — all shared variables must be static globals
    – no dynamic allocation
  — no fine-grain parallelism
    – e.g. over array elements
Happens-before Relation

- Partial order on all events of all threads
- Benefits
  - less restrictive than monitors
  - tools test for happens-before between threads
    - observe every data reference and sync operation
    - if not ordered by happens before, data race could occur
  - supports fine-grain parallelism
- Drawbacks
  - efficient implementation is difficult
  - “effectiveness depends on interleaving”
Eraser Paper Claims

• Race detection based on “happens before”
  — “requires per thread info about accesses per shared variable”
    – not quite right
      requires per variable information about accesses by threads
      requires information about thread ordering
  — “highly depends on interleaving”
    – not quite right either
Happened-before Relation and Locks

Thread 1

\[
\text{lock} (\text{mu}); \\
\text{unlock} (\text{mu});
\]

Thread 2

\[
\text{lock} (\text{mu}); \\
\text{unlock} (\text{mu});
\]

\[
v := v + 1; \\
v := v + 1;
\]
Eraser Motivating Example

Thread 1

\[ y := y+1; \]

\[ \text{lock}(\text{mu}); \]

\[ v := v+1; \]

\[ \text{unlock}(\text{mu}); \]

Thread 2

\[ \text{lock}(\text{mu}); \]

\[ v := v+1; \]

\[ \text{unlock}(\text{mu}); \]

\[ y := y+1; \]
Lockset Algorithm

• Every shared variable access must be protected by a lock
  — differs from race detection for scientific computation

• Eraser checks whether program respects this discipline
  — monitor the reads and writes to variables
  — monitor the locks held
  — infer relation between locks and variables from execution history
    — maintain set of candidate locks $C(v)$ for each shared variable $v$

Let $locks\_held(t)$ be the set of locks held by thread $t$.

For each $v$, initialize $C(v)$ to the set of all locks.

On each access to $v$ by thread $t$,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\}$, then issue a warning.
Locksets in Action

Discovering a potential race with the lockset algorithm

<table>
<thead>
<tr>
<th>Program</th>
<th>locks_held</th>
<th>C(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{}</td>
<td>{mu1,mu2}</td>
</tr>
<tr>
<td>lock(mu1);</td>
<td>{mu1}</td>
<td></td>
</tr>
<tr>
<td>v := v+1;</td>
<td></td>
<td>{mu1}</td>
</tr>
<tr>
<td>unlock(mu1);</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>lock(mu2);</td>
<td>{mu2}</td>
<td></td>
</tr>
<tr>
<td>v := v+1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(mu2);</td>
<td>{}</td>
<td></td>
</tr>
</tbody>
</table>
Issues with Simple Lockset Algorithm

- **Initialization**
  - shared variables frequently initialized without holding a lock

- **Read-shared data**
  - data written only during initialization and read-shared thereafter can be read without locks

- **Read-write locks**
  - multiple readers
  - single writer
Handling Read Sharing

• Delay refinement of candidate set until after initialization
• Reads and writes by first thread before end of initialization — no effect on candidate set
• When does initialization end? Upon access by second thread

Eraser’s Memory States

report races here
Handling Read-write Locks

Adjust locking discipline

• Some lock \( m \) must be held in \text{write} mode for writes of \( v \)
• Lock \( m \) is held in \textit{some} mode (read or write) for reads of \( v \)

For each \( v \), initialize \( C(v) \) to the set of all locks.

On each read of \( v \) by thread \( t \),
set \( C(v) := C(v) \cap \text{locks\_held}(t) \);
if \( C(v) = {} \), then issue a warning.

On each write of \( v \) by thread \( t \),
set \( C(v) := C(v) \cap \text{write\_locks\_held}(t) \);
if \( C(v) = {} \), then issue a warning.
Eraser Implementation

• Use binary rewriter (ATOM) to augment program
  — instrument calls to acquire or release a lock
    – enables maintain lock\_held(t) for each thread t
  — instrument each call to storage allocator
    – enables initialization of C(v) for dynamically allocated data
  — instrument loads and stores to global locations and heap
    – don’t instrument stack accesses (indexed off SP)
    – maintain information per 32-bit word (smallest memory coherence unit)

• Eraser reports
  — file and line number at which race discovered
  — backtrace listing of all active stack frames
  — (thread id, memory address, access type, PC, SP)

• Can log all accesses to a variable if origin of race is unclear
Representing Candidate Lock Sets

- Not a list of locks per memory location!

- Observation:
  - never more than 10K distinct sets of locks observed in a program

- Approach
  - lock set = \{ sorted list of lock addresses \}
  - table of lock sets
    - entries in table are immutable
    - index table with hashing
  - shadow word per variable
    - 30 bit lock set id; 2-bit variable state (eraser state machine state)
    - find shadow word for V by adding fixed displacement to &V
Performance

• Execution slows by a factor of 10-30
  — could improve by inlining checking rather than procedure call
    – e.g. Pin could do this
  — static analysis could eliminate need to check everything
False Alarms with Eraser

• Sources of false alarms
  — memory reuse without resetting shadow memory
    – occurs with program-managed free lists, private allocators
      no way for eraser to know item is protected by new locks
  — private locks
    – implementations of reader-writer locks outside pthreads
      unknown to Eraser
    – interrupt masking (assign pseudo-lock per interrupt level)
  — benign races
    – don’t affect correctness of program
      some intentional
      some accidental
  — post-wait synchronization

• Suppress false alarms with manual annotation
  — EraserIgnoreOn()
  — EraserIgnoreOff()
Eraser in Practice

• Found races
  — 3 of 4 server programs
  — undergraduate programs

• False alarms
  — able to suppress using annotations
Eraser Weakness

• Doesn’t understand happens-before
  — leads to special case for variable initialization
    – turn on race detection when second thread starts accessing data