A Theory of Data Race Detection

Utpal Banerjee, Zhiqiang Ma, Brian Bliss, Paul Petersen

Intel Corporation

Outline

- Introduction
- Basic Concepts
- Race Detection With Limited History
- Algorithms
- Conclusion

Introduction

- Consider a programming environment where a number of threads are active simultaneously.
- The instructions in these threads can be arbitrarily interleaved.
- If two threads access the same location x in shared memory and at least one of the accesses is a ‘write,’ then the final outcome of the program may depend on the order of the two accesses.
- This is another way of saying that a data race may exist in x.

The Devil Is In The Details

- Showing that a data-race exists may not be enough, you also need to know how to fix it
- Software is written using many layers of abstraction
- Because of this it may be difficult to understand the reason for a data-race without knowing the dynamic context (i.e. call stack)
  - Consider memory accesses in memcpy()
  - This is equally true for accesses by both threads
  - Do not forget the allocation point of the memory being accessed

Perfect or Practical?

- The cost of the analysis algorithm can limit
  - Amount of extra detail one can afford to keep
  - Applications that can that be analyzed
- An analysis algorithm be perfectly correct
  - But can be unusable because it does not fit into the limits of the computer being used
- The purpose of this paper and presentation is to describe a rigorous mathematical theory in which the tradeoff between the kinds of data races that can be detected versus the amount of access history kept

Basic Concepts

- Thread
- Segment
- Synchronization Operation (*Sync Op*)
- Posting Sync Op
- Receiving Sync Op
- Precedes
- Parallel

Partial Order Graph

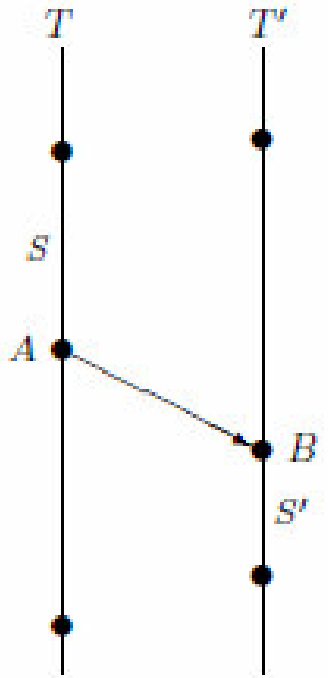

Figure 1: Segment S of Thread T precedes Segment S' of Thread T'.

Race Detection With Limited History

- Focus on a location x in shared memory that is accessed at least once during an execution of the given program
- Let there be n accesses to x during this execution
  - Let $\{S_1, S_2, \ldots, S_n\}$ denote the chronological sequence of segments (of threads in the program) for these accesses
  - Each entry in this sequence corresponds to one access
  - For example, the 4th and 5th accesses to x both came from the same segment S, we will have $S_4 = S_5 = S$.
- When the full access history is available
  - Let S_j be the current segment that has just accessed x
  - Let S_i be the previous segment to access x
  - If S_i and S_j are parallel we have a data race in x
    - Assuming one of the accesses is a write

Race Detection With Limited History (2)

- Due to space constraints it may not be possible to keep all the segments that have already accessed x at each point during program execution.
  - When we find a segment S_j that has just accessed x, a subset of the segments $S_1, S_2, \ldots, S_{j-1}$ may be available for comparison.
- Since we cannot expect to capture all data races that may be present in x, the important question is:
  - When there are data races in x, are we always able to report that at least one such race exists?
  - The answer depends on which subset of segments from the set $\{S_1, S_2, \ldots, S_{j-1}\}$ are available for comparison with S_j .

Algebra of Parallel Segments

- The algebra of parallel segments makes a limited record-keeping scheme viable
- Two segments causing a race that are ‘far apart’ in the sequence $\{S_k\}$, may force a race between two segments that are often relatively ‘close’
- When the goal is to detect if there is at least one race, it is enough to look for races between close pairs of segments
- You only need to keep a few segments that have accessed x in the ‘recent’ past

Adjacent Parallel Segments

Lemma 6. *Let $\{S_1, S_2, \ldots, S_n\}$ denote the chronological sequence of segments that have accessed a memory location x. Let $i, q_1, q_2, \ldots, q_t, j$ denote integers such that $1 \leq i < q_1 < q_2 < \cdots < q_t < j \leq n$. If the segments S_i and S_j are parallel, then the segments in at least one of the $(t+1)$ pairs:*

$$(S_i, S_{q_1}), (S_{q_1}, S_{q_2}), \ldots, (S_{q_{t-1}}, S_{q_t}), (S_{q_t}, S_j)$$

are parallel.

In particular, if i, q, j denote integers such that $1 \leq i < q < j \leq n$, and the segments S_i and S_j are parallel, then either S_i and S_q are parallel, or S_q and S_j are parallel (or both).

Adjacent Conflict Detection

Theorem 7. *If there is an access conflict in x, then there must exist at least one adjacent conflict in x.*

- An access conflict in x exists between two segments S_i and S_j, where $1 \leq i < j \leq n$, if the segments are parallel.
- An access conflict between S_i and S_j is called an adjacent access conflict, or simply an adjacent conflict, if the segments are also consecutive in the sequence $\{S_k\}$, that is, if $i = j - 1$.
- Only need to record the last read or last write
- This algorithm fails to predict a race in x, when there are races, but no adjacent races (only adjacent input conflicts).

Local Conflict Detection

- In the segment sequence $\{S_1, S_2, \ldots, S_n\}$ that access x
  - a member S_k is a read-segment if it reads x, or
  - a write segment if it writes x
- Consider any fixed member S_j for $1 < j \leq n$
- Segment S_i in the subsequence $\{S_1, S_2, \ldots, S_{j-1}\}$
  - Is the last-read segment of S_j, if S_i reads x and
  - The segments $S_{i+1}, S_{i+2}, \ldots, S_{j-1}$, if any, do not
- Similarly, S_i
  - Is the last-write segment of S_j, if S_i writes x and
  - The segments $S_{i+1}, S_{i+2}, \ldots, S_{j-1}$, if any, do not
- An access conflict in x between two segments S_i and S_j
  - Where $1 \leq i < j \leq n$
  - Is a local access conflict, or simply a local conflict
  - If S_i is either the last-write or the last-read segment of S_j

Local Conflict Limitations

Theorem 9. *If there is an output conflict in x, then there must exist at least one local output conflict in x.*

Theorem 10. *If there is an input conflict in x, then there must exist at least one local input conflict in x.*

Theorem 11. *If there is a flow conflict in x, then there must exist at least one local output or one local flow conflict in x.*

Theorem 12. *If there is an anti conflict in x, then there must exist at least one local input conflict or one local anti conflict in x.*

Near Adjacent

- To remedy the deficiency of the Local Conflict Detection Algorithm, we explored what, if any, clues are given by an anti conflict when it does not force a local race.
- We define a special class of anti conflicts that are more general than adjacent conflicts.
- An anti conflict in x between two segments S_i and S_j, where $1 \leq i < j \leq n$, is near-adjacent, if for each k in $i < k < j$, the segment S_k reads x and is parallel to S_i.
- An adjacent anti conflict is clearly near-adjacent

Theorem 15. *If there is an anti conflict in x, then there must exist at least one local output conflict or one near-adjacent anti conflict in x.*

Race Detection Algorithm

- For a memory location x
- Finds all dependences
  - local output
  - local flow
  - near-adjacent anti
- It can always detect if there is a data race in x

Repeat until program execution comes to an end:
  $S \leftarrow$ the next segment to access x;
  If S writes x
  then
    if $S^w \parallel S$
    then report a *local output conflict in* x,
    if $\mathcal{R} \neq \emptyset$
    then
      for each $S' \in \mathcal{R}$ such that $S' \parallel S$
        report a *near-adjacent anti conflict in* x,
      set $\mathcal{R} \leftarrow \emptyset$,
    set $S^w \leftarrow S$;
  else (i.e., if S reads x)
    if $S^w \parallel S$
    then report a *local flow conflict in* x,
    delete each member of $\mathcal{R}$ that is not parallel to S,
    put S in $\mathcal{R}$.
If at least one conflict of type output, flow, or anti has been reported, then
  report that *there is a data race in* x,
else
  report that *there is no data race in* x.

Conclusion

- All of the detection algorithms can be used in a practical situation with different goals in mind.
- The Intel® Thread Checker uses Local Conflict Detection
  - Trade off the ability to always detect data races
  - Allows conservation of memory usage
  - Keep the history of two previous accesses to a memory location
    - The Thread Checker manages to detect the existence of a data race in a vast majority of situations
    - Only misses a R->W data-race when masked by a R->R access
- The last near-adjacent read segment and last write segment is sufficient to detect at least one data race if races are present