On-the-fly Detection of Data Races in OpenMP Programs

Ok-Kyoon Ha

Dept. of Informatics, Gyeongsang National University, Rep. of Korea
Contents

- Parallel Execution Model
- On-the-fly Data Race Detection
- Efficient Thread Labeling
- Precise Detection Protocol
- Implementation
- Experimental Results
- Conclusion
Parallel Execution Model

- **Structured Fork-join Parallelism**
  - consists of at least one set of a fork operation and a corresponding join operation.
    - to identify accesses to a shared memory location on parallel threads
    - to produce a parallel program that is more robust and efficient

- **OpenMP Programs**
  - often employ loop level parallelism which uses the structured fork-join execution of parallel threads.
    - Nested parallelism
    - Series-parallel relationship
An Example of OpenMP Program

```c
float price; /* a shared variable */
#define NUM_RUNS 2

void par_func(int op, int i) {
    #pragma omp parallel for
    for (j=0; j<NUM_RUNS; j++) {
        ...
        do something
    }
}
```

[A POEG for an execution of the program]
Data Races

- The Most Notorious Class of Concurrency Bugs
  - occur when two concurrent threads access a shard memory location without explicit synchronization, and at least one of the accesses is a write.

Detecting data races is important, since they may lead to unpredictable results.
On-the-fly Data Race Detection

- **Happens-before Analysis**
  - uses a logical time stamp to determine the logical concurrency between two events
  - is quite difficult to be efficiently implemented

- **Lockset Analysis**
  - checks only violations of a locking discipline
  - produces too many false positives and cannot guarantee that a program is free from data races

A trade-off between efficiency and preciseness
Hybrid Analysis

- combines two methods to obtain both preciseness and performance for data race detection.
  - partially applies the happens-before analysis on an extended MSM of lockset analysis.

- Helgrind+
  - uses the state-of-the-art hybrid technique
  - reports data races with a lower rate of false alarms and reasonable performance.

There is still room for the improvement of preciseness
Practical Precise Data Race Detection

- We present an on-the-fly approach that combines an efficient thread labeling scheme, called eNR labeling, and a precise detection protocol, called EDP.
Efficient Thread Labeling

- **One-way Region (OR):** $[\lambda, <\alpha, \beta>]$
  - Nest region (NR): a number space which is a pair of lower and upper bound integers, denoted $<\alpha, \beta>$.
  - Join counter $\lambda$: the number of joined thread segment

- $T_0 \rightarrow T_1, \ldots, T_0 \rightarrow T_8$
- $T_1 \parallel T_2 \parallel \ldots \parallel T_8$
NR Labeling uses a list of ORs, called OH, for each thread segment.
eNR Labeling (OR + One-way Descriptor (OD))

- One-way descriptor (OD)
  - is a pair of pointers, \((m_i, L_i)\), to the ancestors of a thread segment

\[\begin{align*}
\text{OD}_7 & \bowtie (OR_5, OD_0) \\
\text{m}_7 & \text{OR}_5 \\
\text{L}_7 & \text{OD}_0 \\
\text{OD}_7 & \bowtie (OR_5, OD_0) \\
\text{m}_7 & \text{X} \\
\text{L}_7 & \text{L}_5 \\
\text{OD}_7 & \bowtie (OR_5, OD_0) \\
\end{align*}\]

- a pointer of OD is inherited from the parent thread segment by every fork operation.
Why eNR Labeling?

- The efficiency of eNR labeling
  - consists of a $OR$ of size $O(1)$ and $OD$ of size $O(1)$.
  - the time complexity is $O(1)$.
  - the space complexity is only $O(T)$

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Label Space</th>
<th>Labeling Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fork/Join</td>
</tr>
<tr>
<td>Vector Clocks</td>
<td>$O(T^2)$</td>
<td>$O(T)$</td>
</tr>
<tr>
<td>NR Labeling</td>
<td>$O(NT)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>eNR Labeling</td>
<td>$O(T)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

- $T$: the maximum number of simultaneously active thread segments
- $N$: the maximum nesting level of the monitored program
**Precise Detection Protocol**

- **Dinning & Schonberg’s Protocol (DSP)**
  - extends happens-before analysis using a lockset optimization.

**An Access History (X)**

<table>
<thead>
<tr>
<th></th>
<th>CS Read</th>
<th>CS Write</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_2$</td>
<td></td>
<td>$W_4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_3$</td>
<td></td>
<td>$W_7$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\ldots$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DSP is precise, but inefficient
EDP (Efficient Detection Protocol)

- is an improvement of DSP to solve the efficiency problem of the original protocol.

- maintains only two concurrent events into CS-Read, CS-Write, and Read set by *left-of-relation*.

  An Access History (X)

<table>
<thead>
<tr>
<th></th>
<th>CS Read</th>
<th>CS Write</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- filters out irrelevant events to consist of data races by checking *redundant lockset*.
  - EDP considers only first events of each event type in a thread segment.
Left-of-relation

- is a relative notion about two concurrent thread segments.
- is simply decided by comparing eNR labels
❖ Redundant Lockset

- The lockset of $R_2 = \{L_1, L_2\}$ is redundant with the lockset of $R_1=\{L_1\}$.

- $R_2$ never consists of a data race with $W_4$ by the lock $L_1$.

❖ The efficiency of EDP can be firmly established by the left-of-relation and the filtering method.
Detecting Data Races using EDP

An Access History ($X$)

<table>
<thead>
<tr>
<th>CS Read</th>
<th>CS Write</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>$W_4$</td>
<td></td>
<td>$W_7$</td>
</tr>
<tr>
<td>$R_6$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

offers epochally reduced space and time complexity as $O(1)$
Implementation

- **Environment**
  - Intel Xeon Quad-core 2 CPUs, 8GB RAM
  - Kernel 2.6 of Linux operating system, gcc 4.4.4 compiler

- **An Easy-to-use Tool**
  - implemented a detector as a Pin-tool on top of Pin framework
Experimental Results

Experimentation

• compared the preciseness and the efficiency of our Pin-Tool with Helgrind+ using a set of benchmark programs.

• Benchmarks

<table>
<thead>
<tr>
<th>Programs</th>
<th>Application Domain</th>
<th>Problem Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>OmpSCR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>Signal Processing</td>
<td>64 signals</td>
</tr>
<tr>
<td>FFT6</td>
<td>Signal Processing</td>
<td>64 signals, 100 iterations</td>
</tr>
<tr>
<td>MD</td>
<td>High Performance</td>
<td>512 particles, 5 simulation steps</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>High Performance</td>
<td>1024 points (to explore Mandelbort graph)</td>
</tr>
<tr>
<td>Pi</td>
<td>High Performance</td>
<td>5,000,000 iterations (to find PI value)</td>
</tr>
<tr>
<td>LU</td>
<td>High Performance</td>
<td>300 columns/rows (dense matrix)</td>
</tr>
<tr>
<td>PARSEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Freqmine</td>
<td>Data Mining</td>
<td>1,000 synthetic transactions, minimum support 3</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>Financial Analysis</td>
<td>4,096 options</td>
</tr>
<tr>
<td>bodytrack</td>
<td>Computer Vision</td>
<td>4 cameras, 1 frame, 100 particles, 3 layers</td>
</tr>
</tbody>
</table>
# Preciseness

<table>
<thead>
<tr>
<th>Programs</th>
<th>Lines</th>
<th>Memory Accesses</th>
<th>Nesting Depth</th>
<th># of Locks</th>
<th>Reported Races</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Read</td>
<td>Write</td>
<td></td>
<td>Read</td>
<td>Write</td>
</tr>
<tr>
<td>FFT</td>
<td>1513</td>
<td>3539K</td>
<td>492K</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FFT6</td>
<td>2332</td>
<td>22852K</td>
<td>15399K</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MD</td>
<td>1663</td>
<td>53584K</td>
<td>9451K</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>1014</td>
<td>144537K</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pi</td>
<td>977</td>
<td>20000K</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LU</td>
<td>1276</td>
<td>125999K</td>
<td>8999K</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>freqmine</td>
<td>13076</td>
<td>2665K</td>
<td>1062K</td>
<td>1</td>
<td>1590</td>
<td>2</td>
</tr>
<tr>
<td>PARSEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blackscholes</td>
<td>593</td>
<td>16630K</td>
<td>819K</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bodytrack</td>
<td>81456</td>
<td>234132K</td>
<td>7423K</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Efficiency (Time)
- Our Pin-tool is 6.5 times faster than Hd\(^+\), and reduces the runtime overhead to 19% of Hd\(^+\).
Efficiency (Memory)

- Our Pin-tool consumes about 1.3 times the memory space of Helgrind+ in the average case.
Conclusion

Practical Precise Data Race Detection

- We presented a combined technique.
- eNR labeling creates the labels for each thread segment with $O(1)$ time and space overhead even in the worst case.
- EDP precisely detects data races with epochally reduced space complexity as $O(1)$ for each access history.

Experimental Results

- We implemented the technique as an easy-to-use tool and compared it with Helgrind+ using a set of benchmarks.
- Empirical results show that our technique is the most precise and efficient for OpenMP programs.
Thank You for your attention

<Q & A>
Appendix

Efficiency (Time)

Average Run Time

- Thread Analyzer
- Thread Checker
- Helgrind+
- Our Pin-Tool

Benchmarks

FFT, FFT6, MD, Mandelbrot, Pi, LU, freqmine, blackscholes, bodytrack
Efficiency (Memory)

![Average Memory Consumption Graph](image)

- **Thread Analyzer**
- **Thread Checker**
- **Helgrind+**
- **Our Pin-Tool**

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Space (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>1200</td>
</tr>
<tr>
<td>FFT6</td>
<td>1100</td>
</tr>
<tr>
<td>MD</td>
<td>1100</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>1100</td>
</tr>
<tr>
<td>Pi</td>
<td>1100</td>
</tr>
<tr>
<td>LU</td>
<td>1100</td>
</tr>
<tr>
<td>freqmine</td>
<td>1400</td>
</tr>
<tr>
<td>blackscholes</td>
<td>1300</td>
</tr>
<tr>
<td>bodytrack</td>
<td>1300</td>
</tr>
</tbody>
</table>
The Example of Data Race in LU Application

- Sequential Execution

```
[ymsim@rhe15 TEST]$ ./LU_seq
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1

-------------------
1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
1 1 1 0 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 1 0 0 0
1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1
[ymsim@rhe15 TEST]$ 
```

- Parallel Execution

```
[ymsim@rhe15 TEST]$ ./LU
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1

-------------------
1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
1 1 1 0 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 1 0 0 0
1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1
[ymsim@rhe15 TEST]$ 
```