Scalable and Precise Dynamic Datarace Detection for Structured Parallelism

Raghavan Raman  Jisheng Zhao  Vivek Sarkar
Rice University

Martin Vechev  Eran Yahav
ETH Zürich  Technion

June 13, 2012
Parallel Programming

• Parallel programming is inherently hard
  – Need to reason about large number of interleavings

• Dataraces are a major source of errors
  – Manifest only in some of the possible schedules
  – Hard to detect, reproduce, and correct
Limitations of Past Work

- Worst case linear space and time overhead per memory access
- Report false positives and/or false negatives
- Dependent on scheduling techniques
- Require sequentialization of input programs
Structured Parallelism

• Trend in newer programming languages: Cilk, X10, Habanero Java (HJ), ...
  – Simplifies reasoning about parallel programs
  – Benefits: deadlock freedom, simpler analysis

• Datarace detection for structured parallelism
  – Different from that for unstructured parallelism
  – Logical parallelism is much larger than number of processors
Contribution

• First practical datarace detector which is parallel with constant space overhead
  – Scalable
  – Sound and Precise
Structured Parallelism in X10, Hj

• `async <stmt>`
  – Creates a new task that executes `<stmt>`

• `finish <stmt>`
  – Waits for all tasks spawned in `<stmt>` to complete
SPD3: Scalable Precise Dynamic Datarace Detection

• Identifying parallel accesses
  – Dynamic Program Structure Tree (DPST)

• Identifying interfering accesses
  – Access Summary
Dynamic Program Structure Tree (DPST)

- Maintains parent-child relationships among async, finish, and step instances
  - A step is a maximal sequence of statements with no async or finish
DPST Example

```c
finish { // F1
```
DPST Example

```java
finish {
    // F1
    S1;
}
```
finish { // F1
    S1;
    async { // A1
        }
}
finish { // F1
    S1;
    async { // A1

    }
    S5;

DPST Example

```
finish { // F1
    S1;
    async { // A1
        async { // A2
            
            }
        }
    }  

    S5;
```

finish { // F1
    S1;
    async { // A1
        async { // A2
            }
        }
    }
}  
S5;
DPST Example

```javascript
finish { // F1
    S1;
    async { // A1
        async { // A2

        }
        async { // A3

        } S4;
    } S5;
}
```
finish { // F1
    S1;
    async { // A1
        async { // A2
            S2;
        }
        async { // A3
            }
        S4;
    }
    S5;
}
DPST Example

```javascript
finish { // F1
    S1;
    async { // A1
        async { // A2
            S2;
        }
        async { // A3
            S3;
        }
        S4;
    }
    S5;
}
```
DPST Example

```javascript
finish { // F1
  S1;
  async { // A1
    async { // A2
      S2;
    }
    async { // A3
      S3;
    }
    S4;
  }
  S5;
  async { // A4
    S6;
  }
}
```
DPST Example

1: `finish {  // F1
2:   S1;
3:   async {  // A1
4:     async {  // A2
5:       S2;
6:     }
7:     async {  // A3
8:       S3;
9:     }
10:   S4;
11: }
12:   S5;
13:   async {  // A4
14:     S6;
15: }
16: }

Left-to-right ordering of children
DPST Operations

• InsertChild (Node n, Node p)
  – O(1) time
  – No synchronization needed

• DMHP (Node n1, Node n2)
  – O(H) time
    • H = height(LCA(n1, n2))
  – DMHP = Dynamic May Happen in Parallel
Identifying Parallel Accesses using DPST

DMHP (S1, S2)

1) L = LCA (S1, S2)
2) C = child of L that is ancestor of S1
3) If C is async
   return true
   Else return false
Identifying Parallel Accesses using DPST

DMHP (S1, S2)

1) $L = \text{LCA}(S1, S2)$
2) $C = \text{child of } L \text{ that is ancestor of } S1$
3) If $C$ is async
   return true
   Else return false
Identifying Parallel Accesses using DPST

DMHP (S1, S2)

1) L = LCA (S1, S2)
2) C = child of L that is ancestor of S1
3) If C is async
   return true
   Else return false
Identifying Parallel Accesses using DPST

DMHP (C)

1) \( L = \text{LCA} (S_1, S_2) \)
2) \( C = \text{child of } L \text{ that is ancestor of } S_1 \)
3) If \( C \) is async
   return true
   Else return false
Identifying Parallel Accesses using DPST

DMHP (S1, S2)

1) L = LCA (S1, S2)
2) C = child of L that is ancestor of S1
3) If C is async
   return true
   Else return false

A1 is an async => DMHP (S3, S6) = true
Identifying Parallel Accesses using DPST

DMHP (S1, S2)

1) L = LCA (S1, S2)
2) C = child of L that is ancestor of S1
3) If C is async
   return true
   Else return false
Identifying Parallel Accesses using DPST

DMHP (S1, S2)

1) L = LCA (S1, S2)
2) C = child of L that is ancestor of S1
3) If C is async return true
   Else return false
Identifying Parallel Accesses using DPST

DMHP (S1, S2)

1) \( L = \text{LCA} (S1, S2) \)
2) \( C = \text{child of } L \text{ that is ancestor of } S1 \)
3) If \( C \) is async return true
   Else return false

Child of F1 that is ancestor of S5
Identifying Parallel Accesses using DPST

DMHP (S1, S2)

1) L = LCA (S1, S2)
2) C = child of L that is ancestor of S1
3) If C is async
   return true
   Else return false

S5 is NOT an async => DMHP (S5, S6) = false
Access Summary

Program Memory

\[
M \quad M_s.w \quad M_s.r_1 \quad M_s.r_2
\]

Shadow Memory

\[
\vdots \quad \vdots
\]
Access Summary

Program Memory

- M

Shadow Memory

- $M_s.w$
- $M_s.r_1$
- $M_s.r_2$

A Step Instance that Wrote M
Access Summary

Program Memory

- **M**

Shadow Memory

- **M_s.w**
- **M_s.r_1**
- **M_s.r_2**

Two Step Instances that Read M
Access Summary Operations

• WriteCheck (Step S, Memory M)
  – Check for access that interferes with a write of M by S

• ReadCheck (Step S, Memory M)
  – Check for access that interferes with a read of M by S
SPD3 Example

F1
  ┌── S1
  │   └── A1
  │       ┌── S5
  │       │   └── A4
  │       └── A2
  │           └── S2
  │                   = M
  │               ┌── S3
  │                   │   = M
  │               │   └── S4
  │                   │       = M
  │               └── S6
  │                   │       M =
SPD3 Example

<table>
<thead>
<tr>
<th>Executing Step</th>
<th>$M_s.r_1$</th>
<th>$M_s.r_2$</th>
<th>$M_s.w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

F1

S1
A1
S5
A4

A2
S2
= M

A3
S3
= M

S4
= M

S6
M =

A1
S5
A4

F1

RICE
SPD3 Example

Executing Step | $M_s.r_1$ | $M_s.r_2$ | $M_s.w$
---|---|---|---
S1 | null | null | null
S4 (Read M) | S4 | null | null

Update $M_s.r_1$
SPD3 Example

Executing Step | $M_{s.r_1}$ | $M_{s.r_2}$ | $M_{s.w}$
--- | --- | --- | ---
S1 | null | null | null
S4 (Read M) | S4 | null | null
S3 (Read M) | S4 | S3 | null

Update $M_{s.r_2}$
SPD3 Example

S3, S4 stand for subtree under LCA(S3,S4)

<table>
<thead>
<tr>
<th>Executing Step</th>
<th>$M_s.r_1$</th>
<th>$M_s.r_2$</th>
<th>$M_s.w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>S4 (Read M)</td>
<td>S4</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>S3 (Read M)</td>
<td>S4</td>
<td>S3</td>
<td>null</td>
</tr>
</tbody>
</table>
SPD3 Example

Executing Step | $M_s.r_1$ | $M_s.r_2$ | $M_s.w$
--- | --- | --- | ---
S1 | null | null | null
S4 (Read M) | S4 | null | null
S3 (Read M) | S4 | S3 | null
S2 (Read M) | S4 | S3 | null

S2 is in the subtree under LCA(S3, S4) => Ignore S2
Report a Read-Write Datarace between steps S4 and S6
SPD3 Algorithm

• At async, finish, and step boundaries
  – Update the DPST

• On every access to a memory M, *atomically*
  – Read the fields of its shadow memory, $M_s$
  – Perform ReadCheck or WriteCheck as appropriate
  – Update the fields of $M_s$, if necessary
Space Overhead

- DPST: $O(a+f)$
  - ‘a’ is the number of async instances
  - ‘f’ is the number of finish instances

- Shadow Memory: $O(1)$ per memory location
Empirical Evaluation

• Experimental Setup
  – 16-core (4x4) Intel Xeon 2.4GHz system
    • 30 GB memory
    • Red Hat Linux (RHEL 5)
    • Sun Hotspot JDK 1.6
  – All benchmarks written in HJ using only Finish/Async constructs
    • Executed using the adaptive work-stealing runtime
  – SPD3 algorithm
    • Implemented in Java with static optimizations
SPD3 is Scalable

The chart shows the slowdown relative to respective (w.r.t. number of threads) uninstrumented for different benchmarks and the number of threads: 1-thread, 2-thread, 4-thread, 8-thread, 16-thread. The benchmarks include Crypt, LUFact, MolDyn, MonteCarlo, RayTracer, Series, SOR, SparseMatMult, FFT, Health, Nqueens, Strassen, Fannkuch, Mandelbrot, Matmul, and GeoMean.
Estimated Peak Heap Memory Usage on 16-threads

- Eraser
- FastTrack
- SPD3

Estimated Peak Heap Memory Usage (MB)
Estimated Peak Heap Memory Usage
LUFact Benchmark

Estimated Peak Heap Memory Usage (MB)

Number of threads

Eraser  FastTrack2  SPD3
# Related Work: A Comparison

<table>
<thead>
<tr>
<th>Properties</th>
<th>OTFDAA</th>
<th>Offset-Span</th>
<th>SP-bags</th>
<th>SP-hybrid</th>
<th>FastTrack</th>
<th>ESP-bags</th>
<th>SPD3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Language</td>
<td>Nested Fork-Join &amp; Synchronization operations</td>
<td>Nested Fork-Join</td>
<td>Spawn-Sync</td>
<td>Spawn-Sync</td>
<td>Unstructured Fork-Join</td>
<td>Async-Finish</td>
<td>Async-Finish</td>
</tr>
<tr>
<td>Space Overhead per memory location</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Guarantees</td>
<td>Per-Schedule</td>
<td>Per-Input</td>
<td>Per-Input</td>
<td>Per-Input</td>
<td>Per-Input</td>
<td>Per-Input</td>
<td>Per-Input</td>
</tr>
<tr>
<td>Empirical Evaluation</td>
<td>No</td>
<td>Minimal</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Execute Program in Parallel</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dependent on Scheduling technique</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

OTFDAA – On the fly detection of access anomalies (PLDI ’89)  
$n$ – number of threads executing the program  
$N$ – maximum logical concurrency in the program
Summary

• First practical datarace detector which is parallel with constant space overhead
  – Dynamic Program Structure Tree
  – Access Summary
Slowdown of Crypt Benchmark

The graph shows the slowdown relative to a 16-thread baseline for three different benchmarks: Eraser, FastTrack2, and SPD3. The x-axis represents the number of threads, while the y-axis shows the slowdown relative to the 16-thread baseline.

- Eraser: The slowdown is relatively constant across different thread counts, with a slight increase as the number of threads increases.
- FastTrack2: This benchmark shows a significant increase in slowdown when the number of threads is doubled from 1 to 2, and then a peak at 4 threads before decreasing as the number of threads increases further.
- SPD3: The slowdown for SPD3 increases significantly as the number of threads increases, with a sharp peak at 4 threads before decreasing.

The graph illustrates the impact of thread scaling on the performance of these benchmarks, highlighting the differences in their scalability characteristics.
Comparison with Eraser and FastTrack

• Eraser (TOCS ‘97) and FastTrack (PLDI ’09)
  – Implementation obtained from RoadRunner tool
  – Use RoadRunner as the baseline (RR-Base)
  – Use the Java versions of the benchmarks

• SPD3
  – HJ program without instrumentation (HJ-Base) is the baseline
Slowdown of Crypt Benchmark

![Graph showing slowdown relative to 16-thread RR-Base for different benchmarks and thread counts. The x-axis represents the number of threads, ranging from 1 to 16. The y-axis represents slowdown relative to 16-thread RR-Base, ranging from 0.00 to 300.00. The graph includes lines for FastTrack, Eraser, SPD3, HJ-Base, and RR-Base. The FastTrack line is the most prominent, showing a peak around 16 threads.]
Estimated Peak Heap Memory Usage
LUFact Benchmark

![Graph showing estimated peak heap memory usage for different benchmarks with varying number of threads.](image-url)
# Slowdown on 16-threads

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RR-Base Time(s)</th>
<th>Eraser Slowdown</th>
<th>FastTrack Slowdown</th>
<th>HJ-Base Time(s)</th>
<th>SPD3 Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypt</td>
<td>0.362</td>
<td>122.40</td>
<td>133.24</td>
<td>0.585</td>
<td>1.84</td>
</tr>
<tr>
<td>LUFact*</td>
<td>1.47</td>
<td>17.95</td>
<td>26.41</td>
<td>5.411</td>
<td>1.08</td>
</tr>
<tr>
<td>MolDyn*</td>
<td>16.185</td>
<td>8.39</td>
<td>9.59</td>
<td>3.75</td>
<td>13.56</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>2.878</td>
<td>10.95</td>
<td>13.54</td>
<td>5.605</td>
<td>1.86</td>
</tr>
<tr>
<td>RayTracer*</td>
<td>2.186</td>
<td>20.23</td>
<td>17.45</td>
<td>19.974</td>
<td>5.84</td>
</tr>
<tr>
<td>Series</td>
<td>112.515</td>
<td>1.00</td>
<td>1.00</td>
<td>88.768</td>
<td>1.00</td>
</tr>
<tr>
<td>SOR*</td>
<td>0.914</td>
<td>4.26</td>
<td>8.36</td>
<td>2.604</td>
<td>4.53</td>
</tr>
<tr>
<td>SparseMatMult</td>
<td>2.746</td>
<td>14.29</td>
<td>20.59</td>
<td>4.607</td>
<td>1.72</td>
</tr>
<tr>
<td><strong>Geometric Mean</strong></td>
<td><strong>11.21</strong></td>
<td><strong>13.87</strong></td>
<td></td>
<td></td>
<td><strong>2.63</strong></td>
</tr>
</tbody>
</table>

The original Java versions of the benchmarks marked with * had races. For these benchmarks, race-free versions were used for SPD3 but the original versions were used for Eraser and FastTrack.