Scalable and Precise Dynamic Datarace Detection for Structured Parallelism

Raghavan Raman   Jisheng Zhao   Vivek Sarkar

Rice University

Martin Vechev   Eran Yahav

ETH Zürich   Technion

RICE

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)

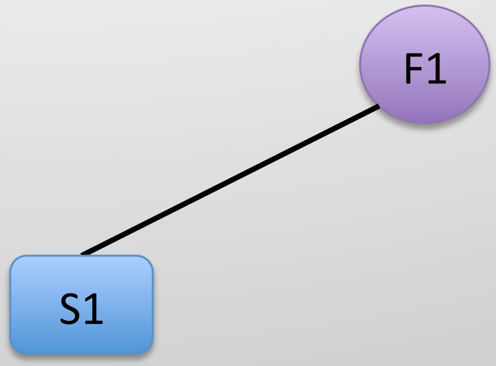
- Maintains parent-child relationships among async, finish, and step instances
  – A step is a maximal sequence of statements with no async or finish

DPST Example

```
finish { // F1
```

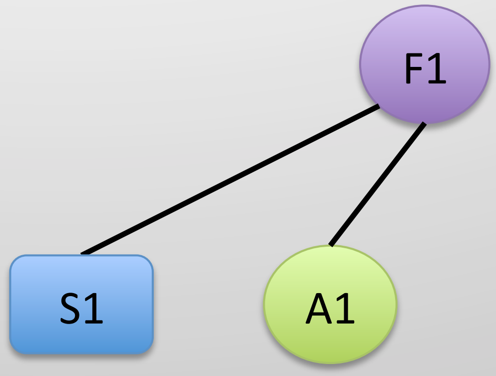
DPST Example

```
finish { // F1
   S1;
```



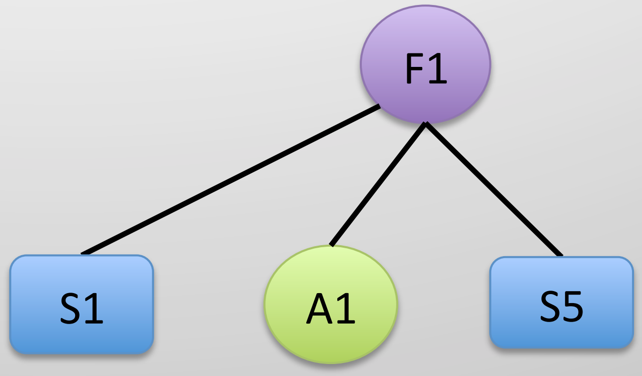
DPST Example

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

   }
```



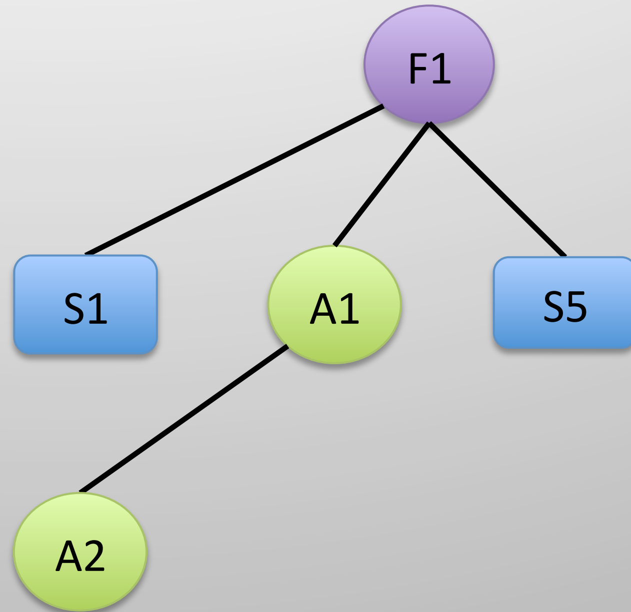
DPST Example

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

   }
   S5;
```



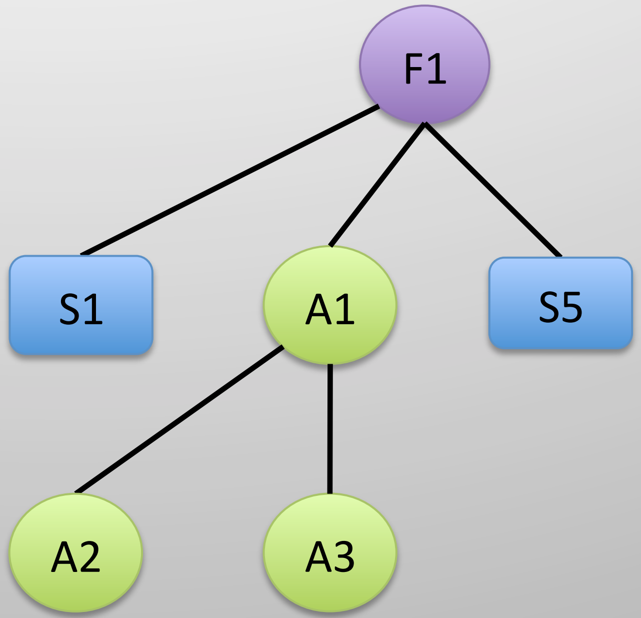
DPST Example

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

      }
   }
   S5;
```



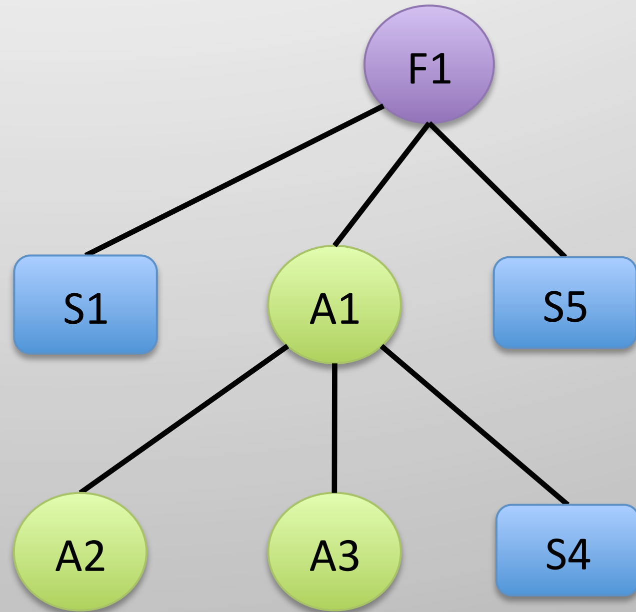
DPST Example

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

        }
        async { // A3

        }
    }
    S5;
```



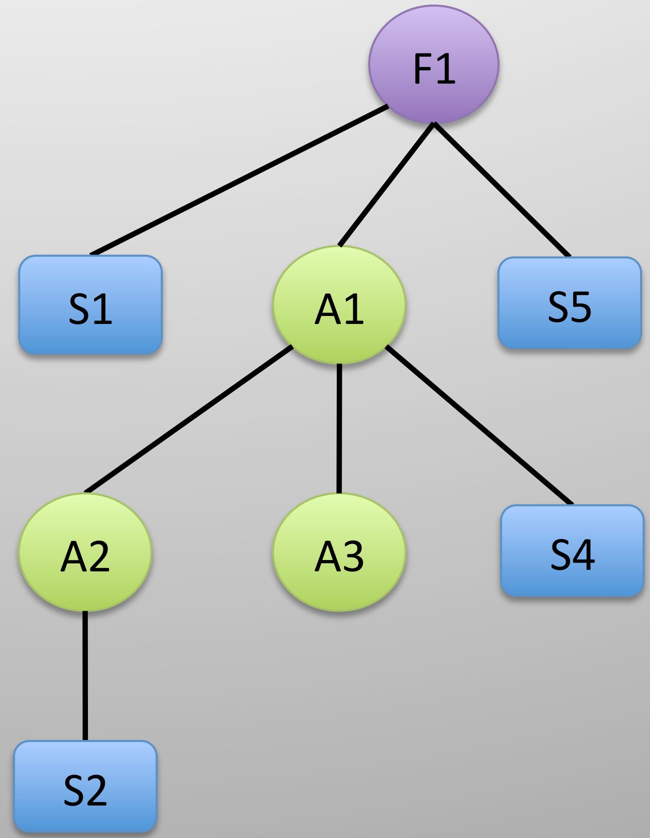
DPST Example

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

      }
      async { // A3

      }
      S4;
   }
   S5;
```



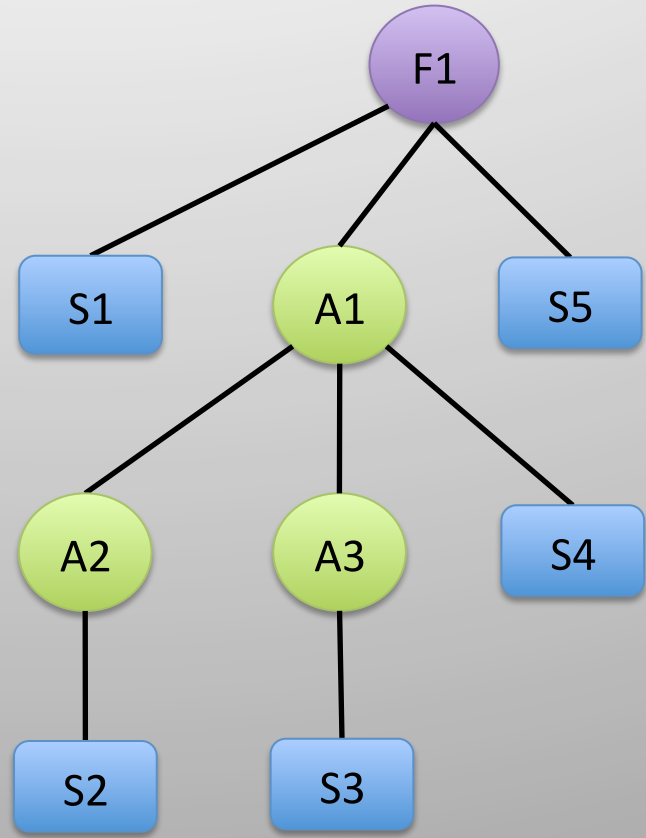
DPST Example

```
finish { // F1
   S1;
   async { // A1
      async { // A2
         S2;
      }
      async { // A3

      }
      S4;
   }
   S5;
```



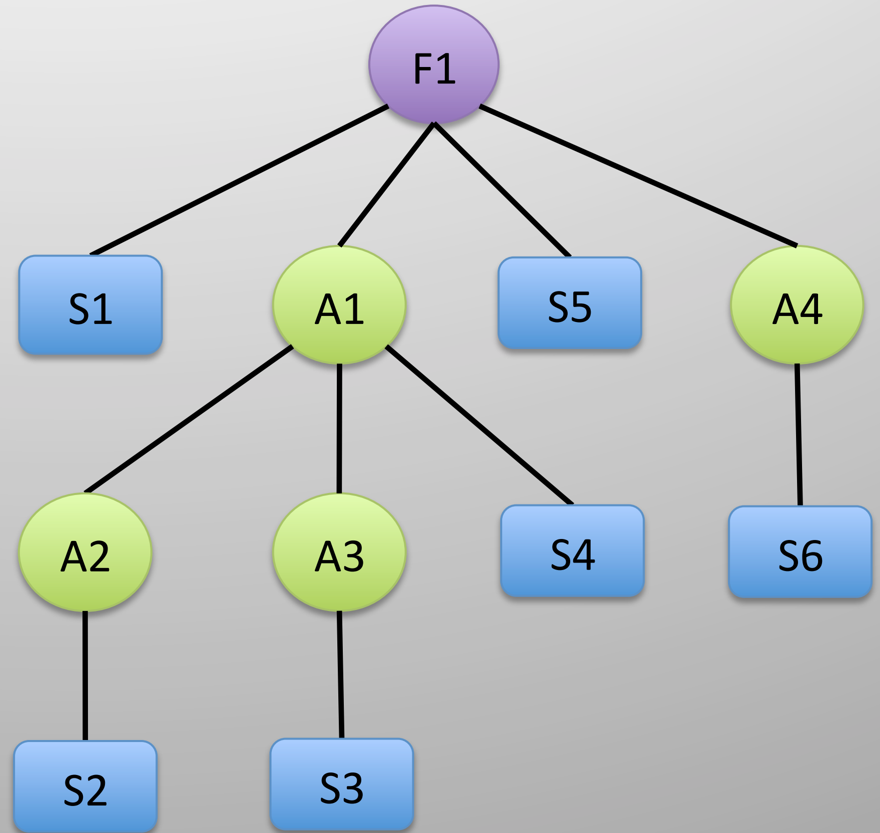
DPST Example

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



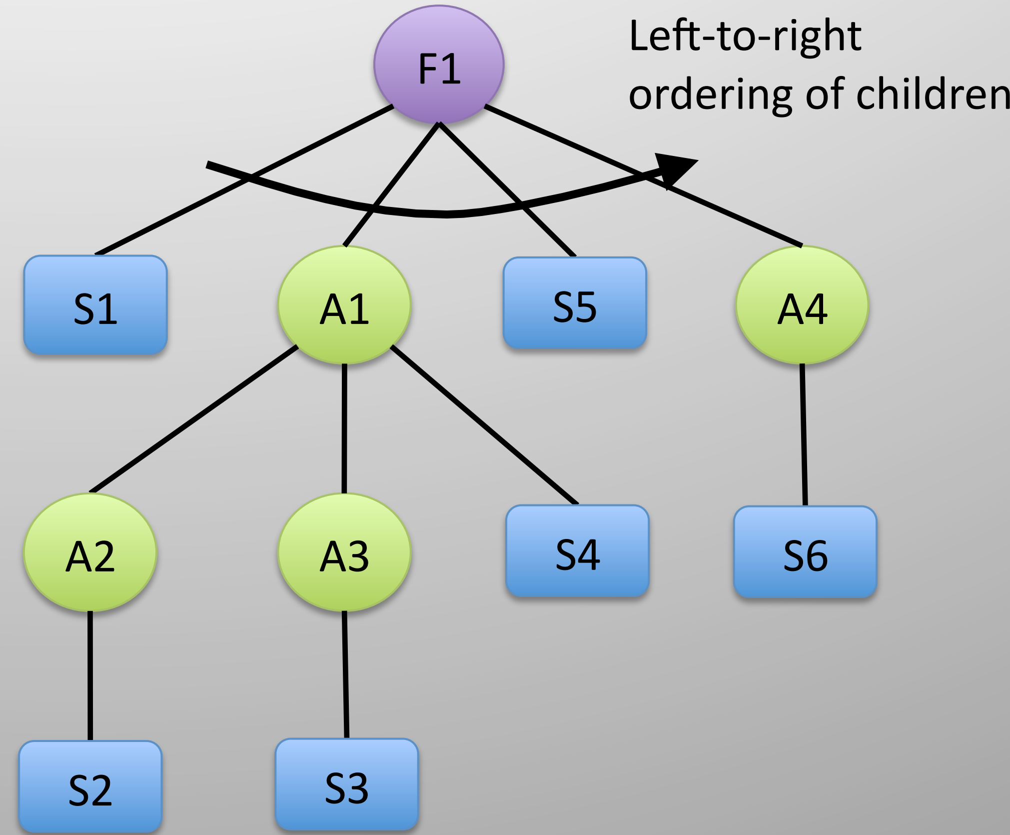
DPST Example

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



DPST Example

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



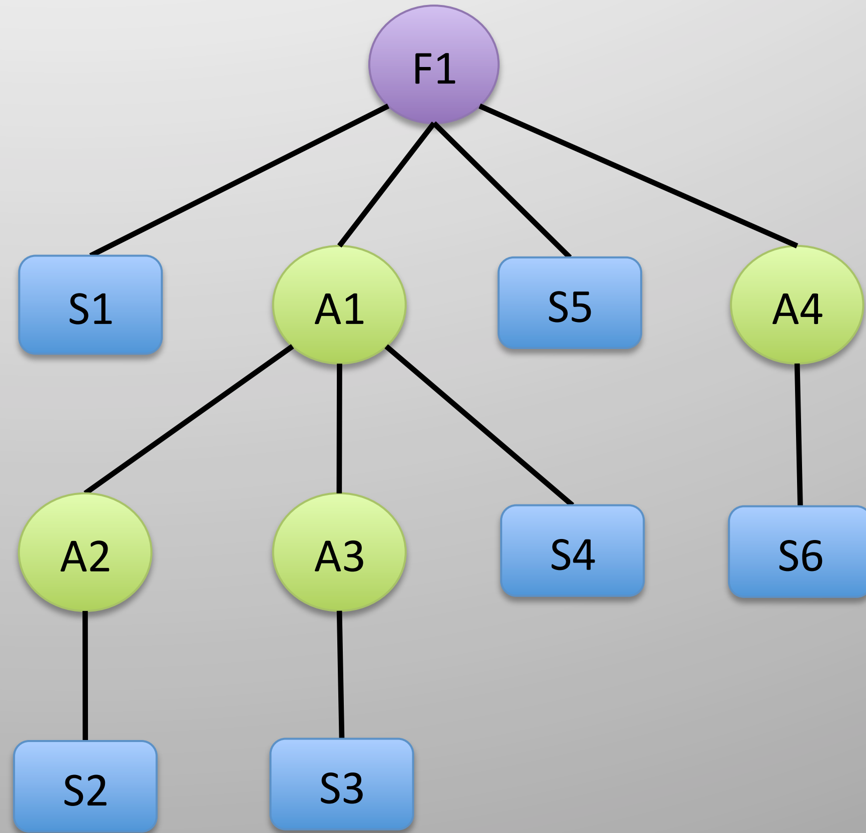
DPST Operations

- 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)

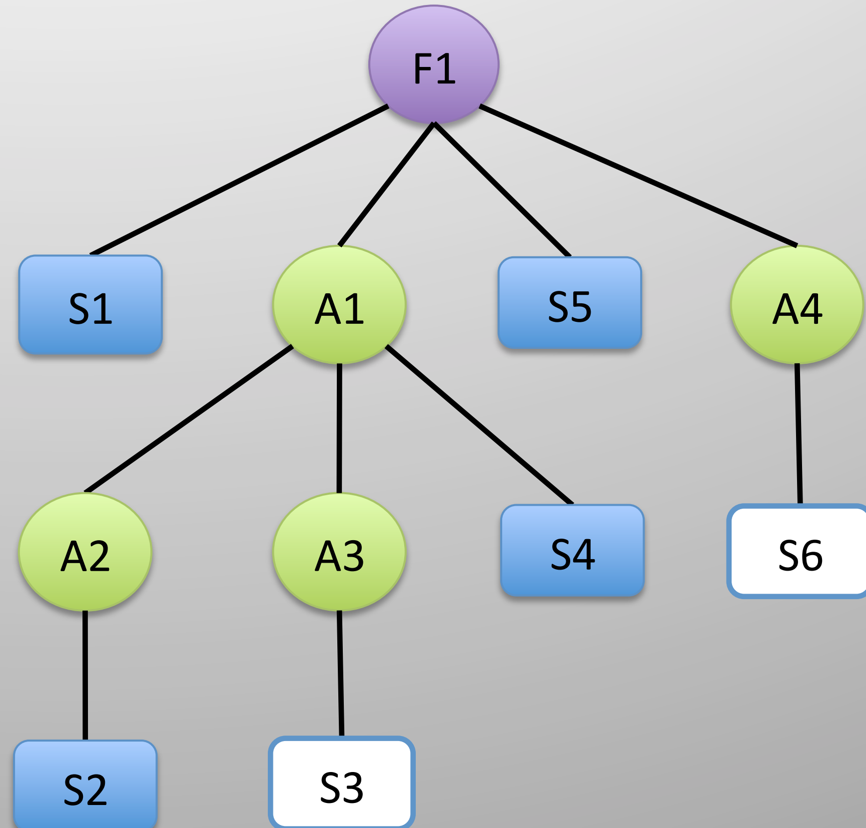
1) L = LCA (S1, S2)
2) C = child of L that is
   ancestor of S1
3) If C is async
   return true
   Else return false



Identifying Parallel Accesses using DPST

DMHP (S1, S2)

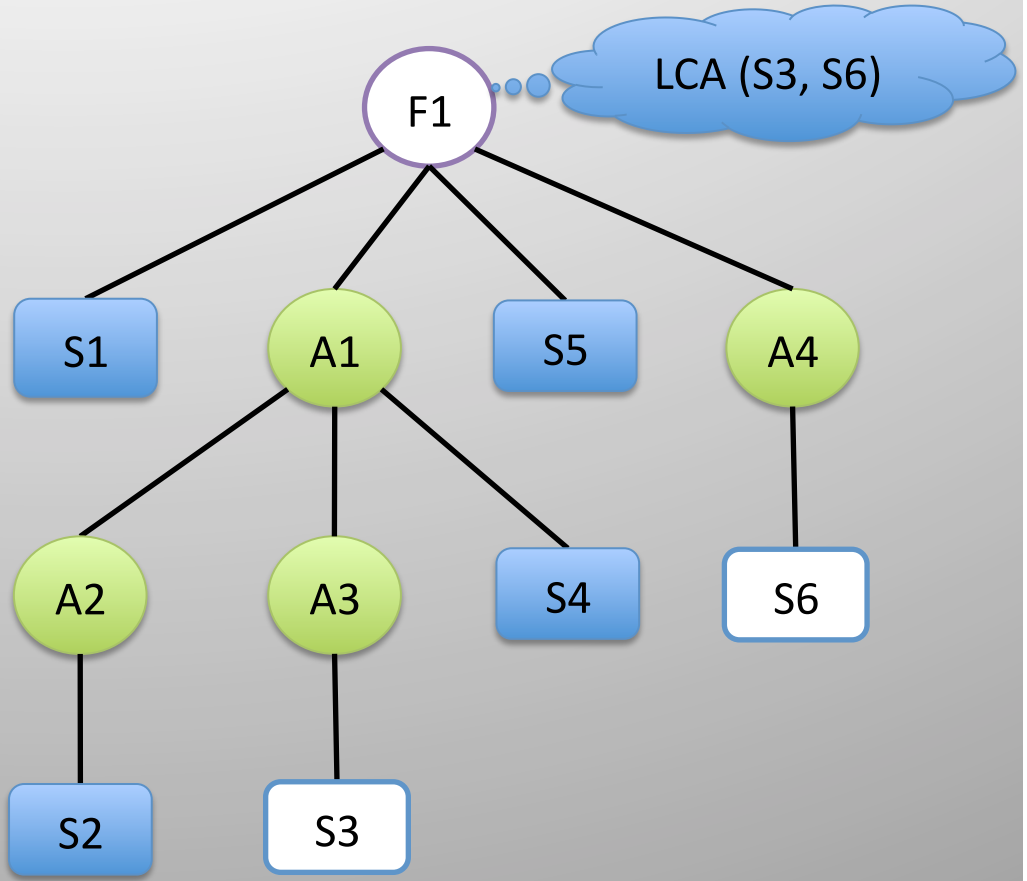
1) L = LCA (S1, S2)
2) C = child of L that is ancestor of S1
3) If C is async return true
   Else return false



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

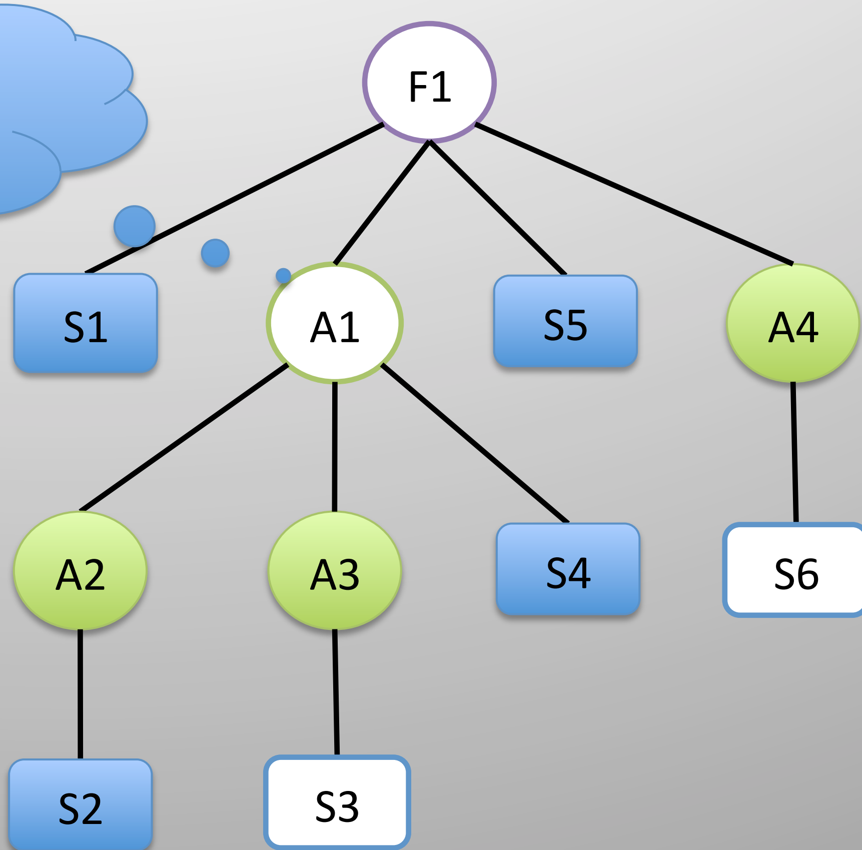


Identifying Parallel Accesses using DPST

DMHP (

Child of F1 that is ancestor of S3

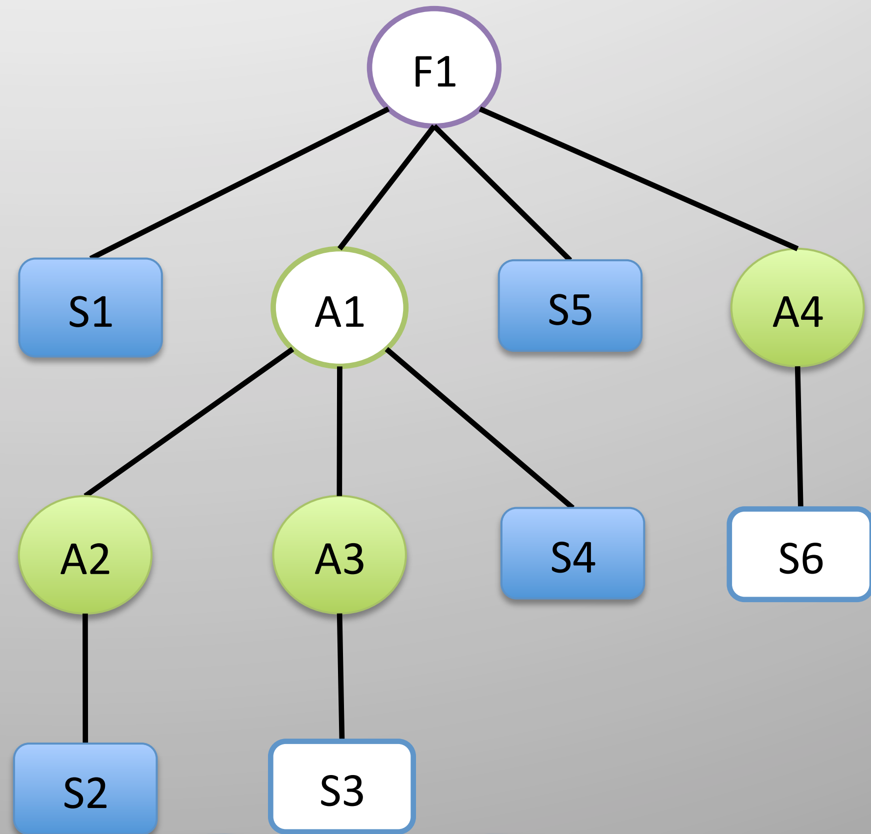
1) L = LCA (S1, S2)
2) C = child of L that is ancestor of S1
3) If C is async return true
   Else return false



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

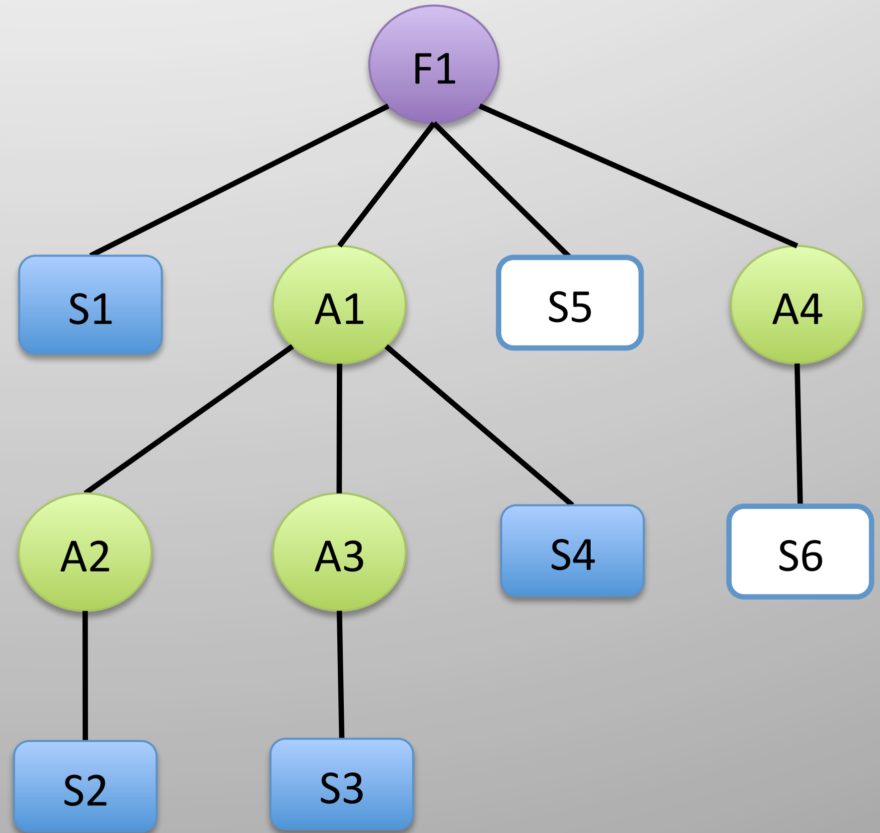


A1 is an async => DMHP (S3, S6) = true

Identifying Parallel Accesses using DPST

DMHP (S1, S2)

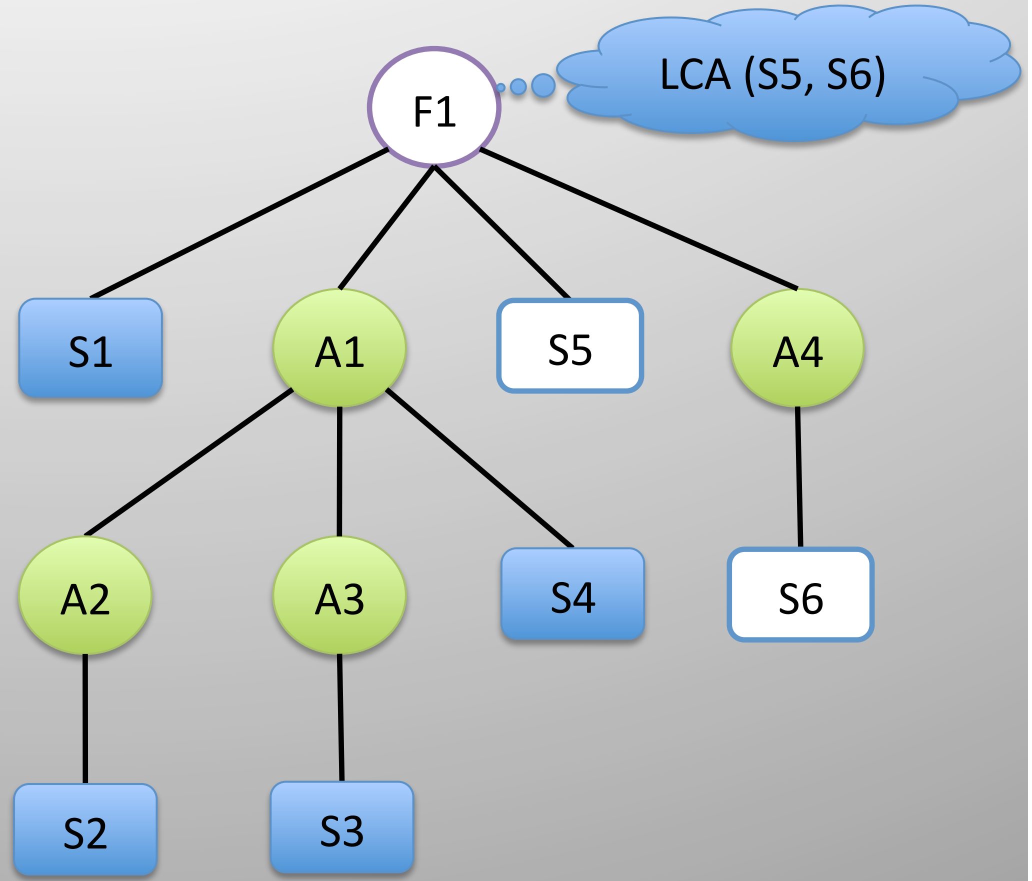
1) L = LCA (S1, S2)
2) C = child of L that is ancestor of S1
3) If C is async return true
   Else return false



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

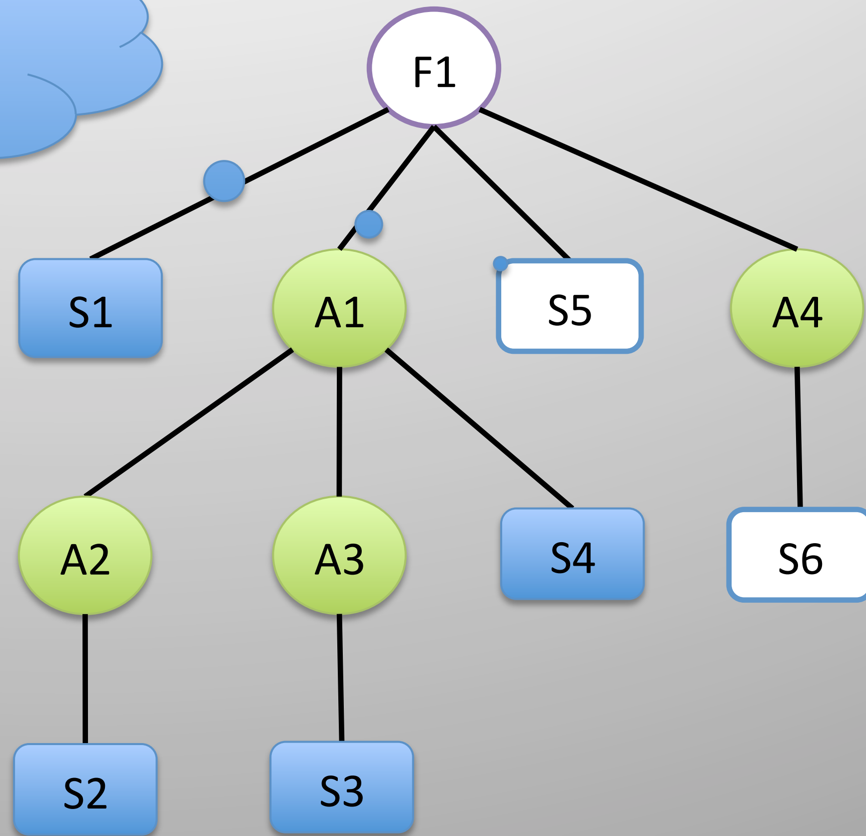


Identifying Parallel Accesses using DPST

DMHP (S

Child of F1 that is
ancestor of S5

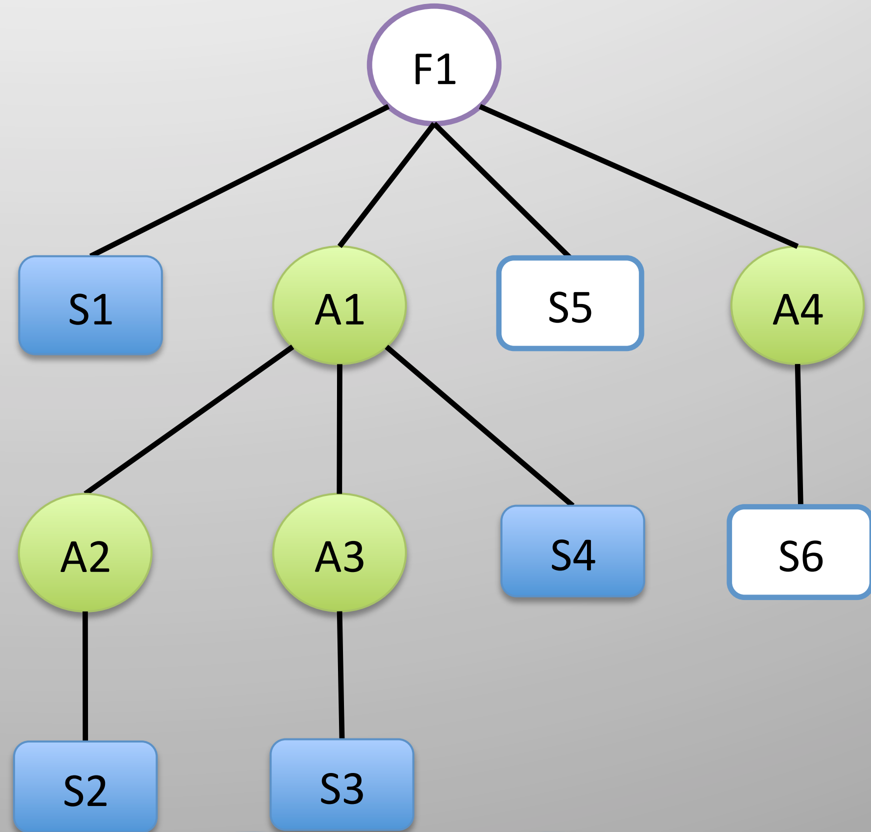
1) L = LCA (S1, S2)
2) C = child of L that is
 ancestor of S1
3) If C is async
 return true
 Else return false



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



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

Access Summary

Program Memory

M

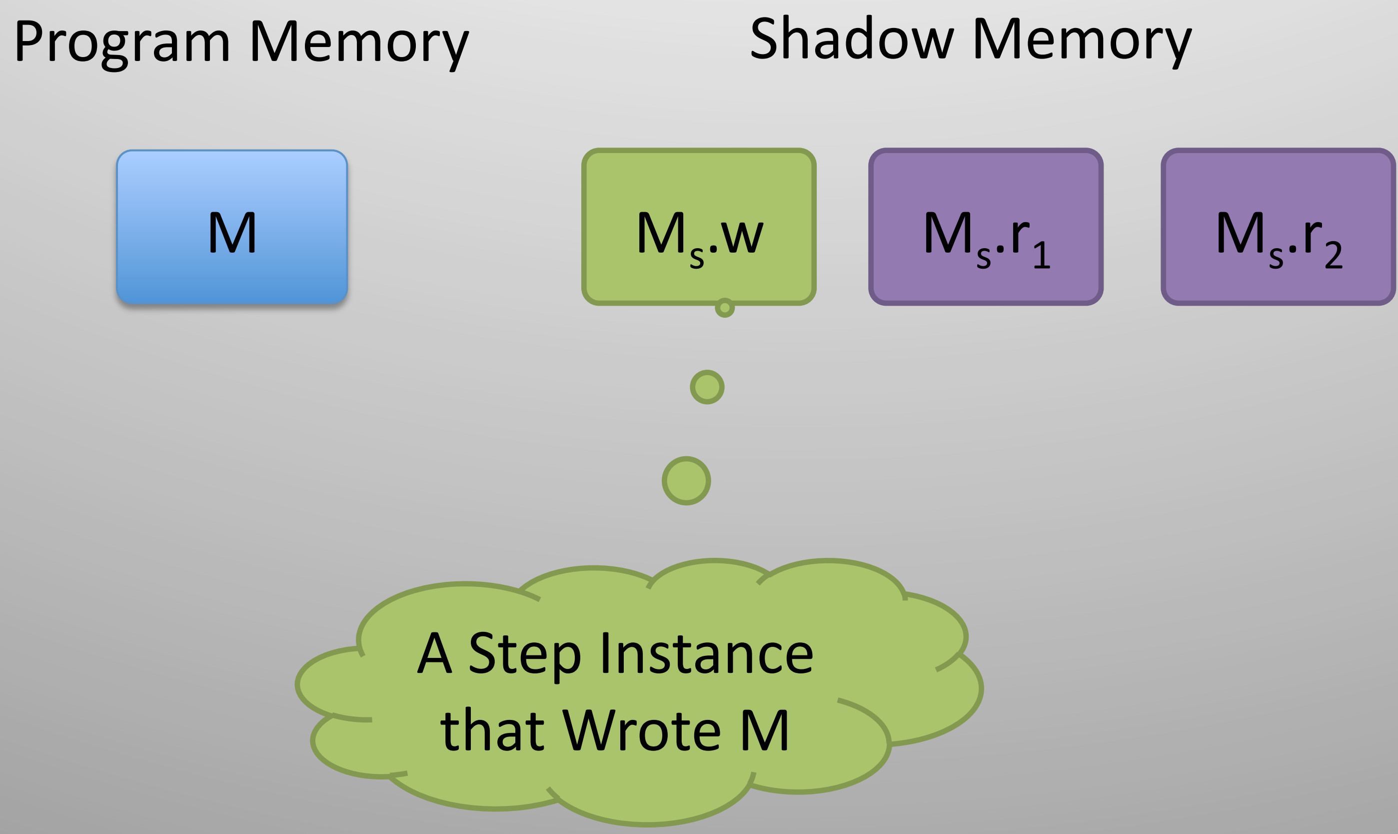
⋮

Shadow Memory

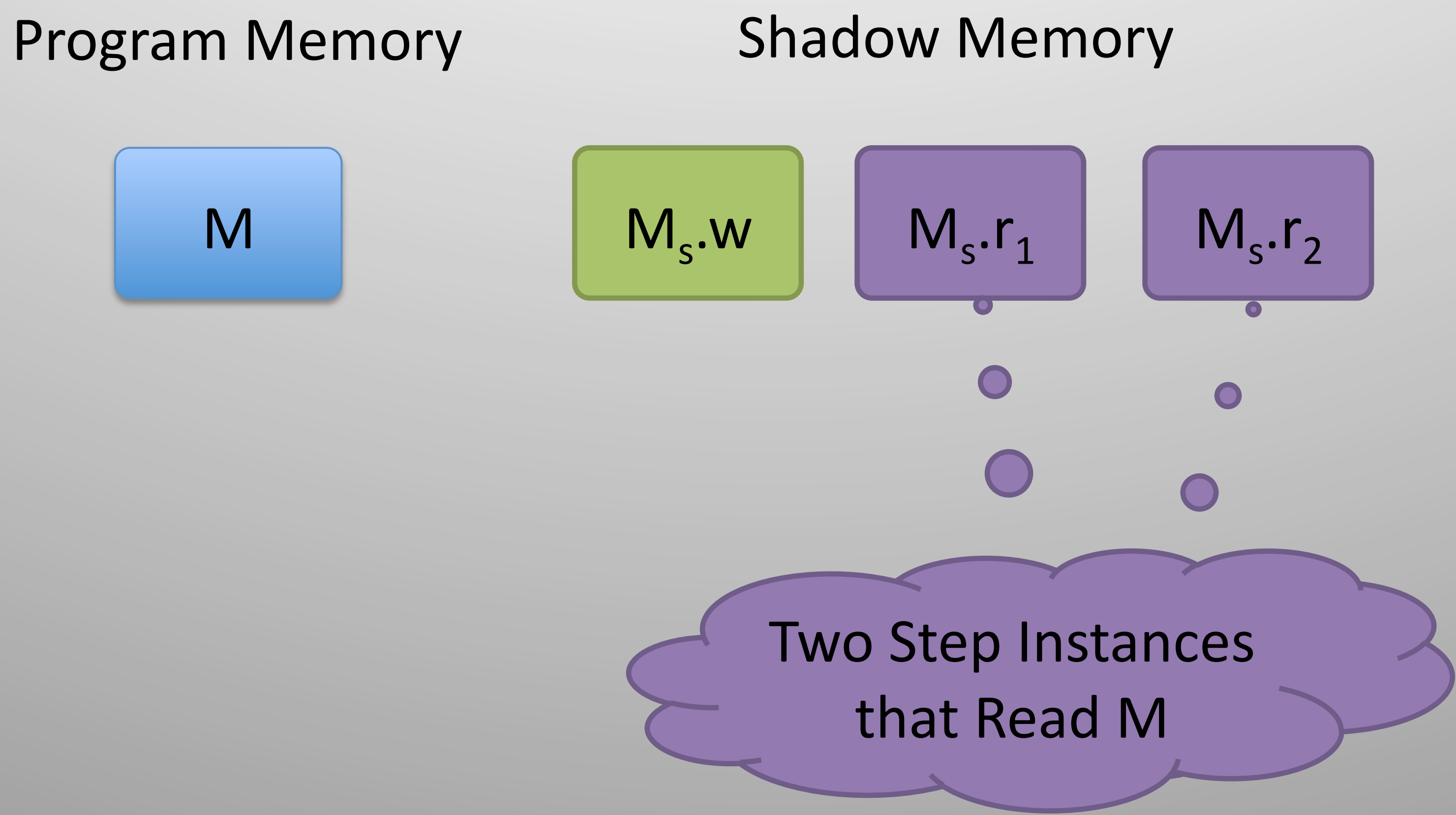
$M_s.w$ $M_s.r_1$ $M_s.r_2$

⋮

Access Summary



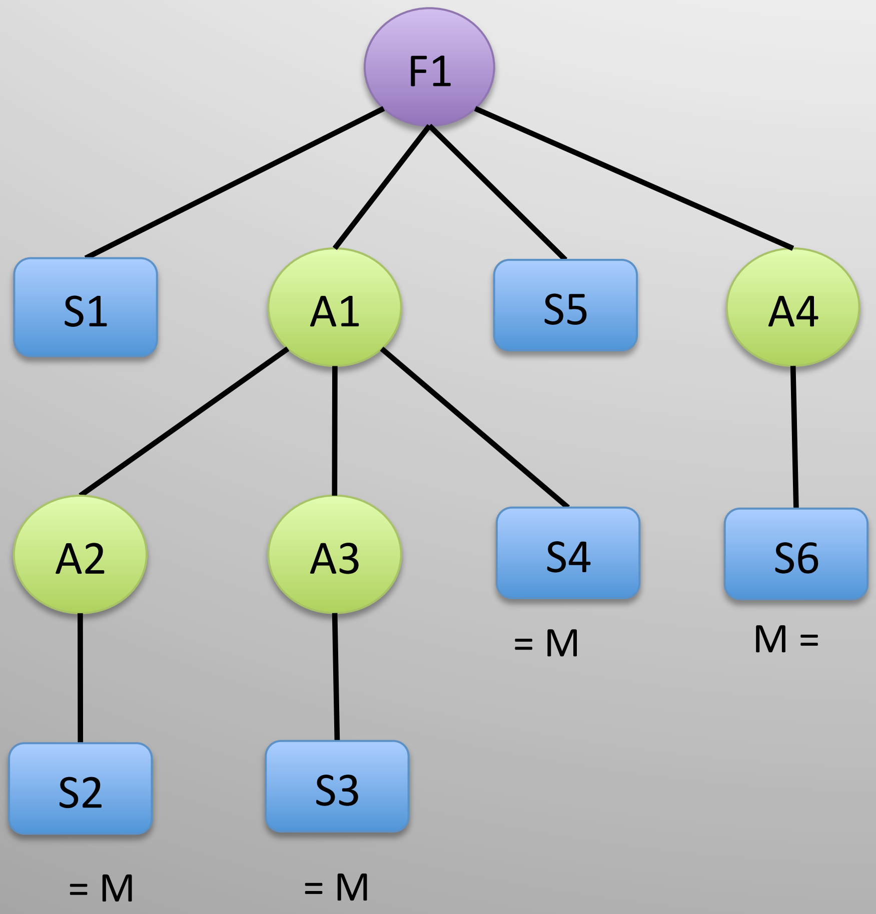
Access Summary



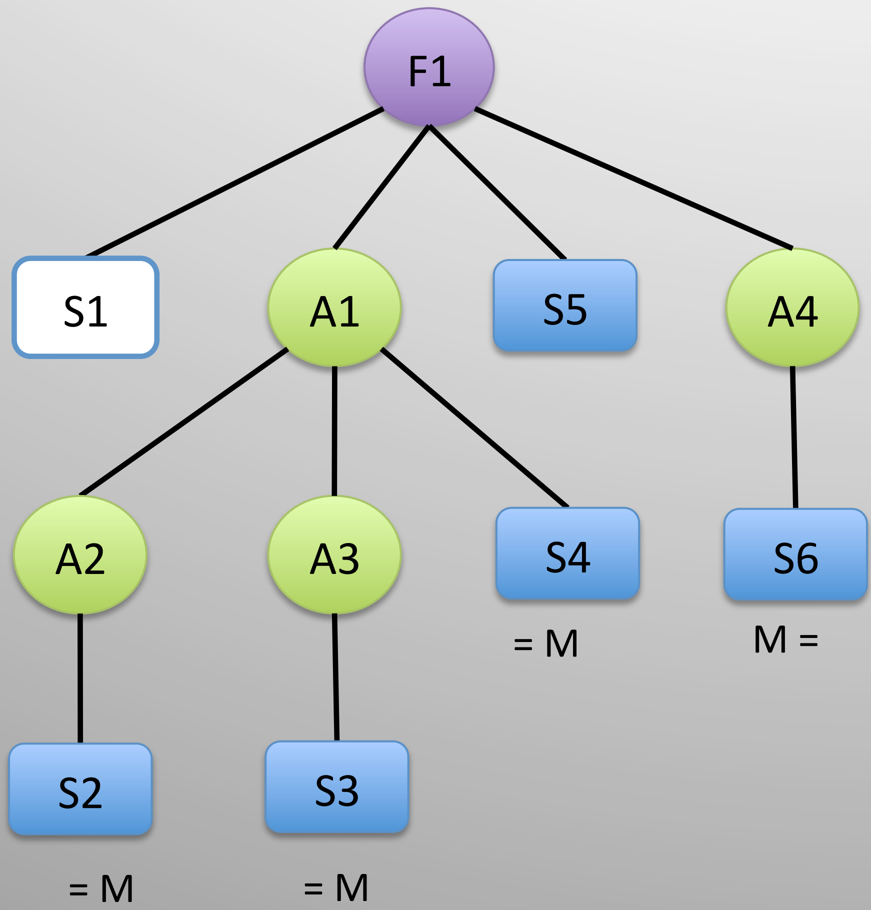
Access Summary Operations

- 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

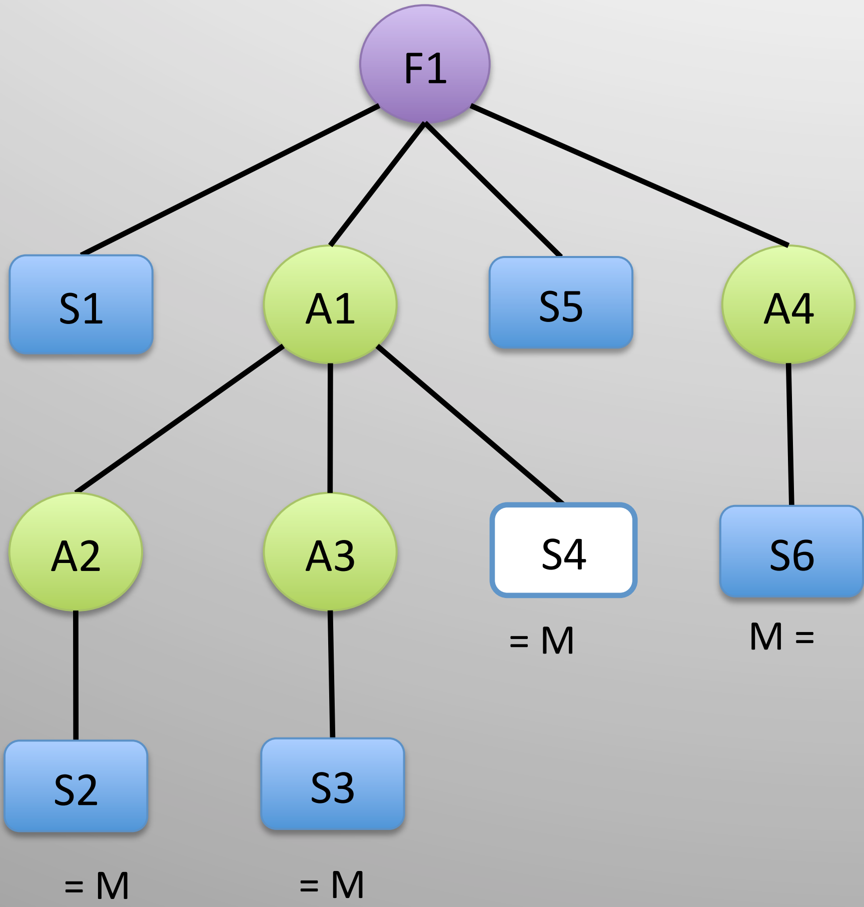


SPD3 Example



Executing Step	$M_s.r_1$	$M_s.r_2$	$M_s.w$
S1	null	null	null

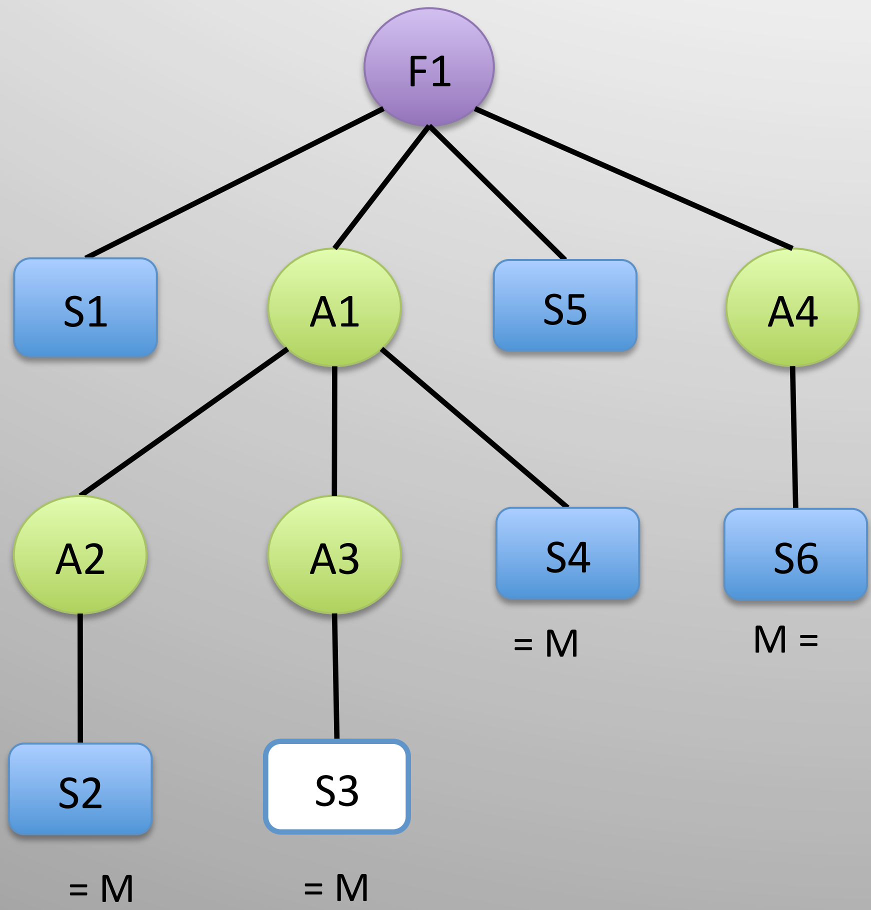
SPD3 Example



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

Update $M_s.r_1$

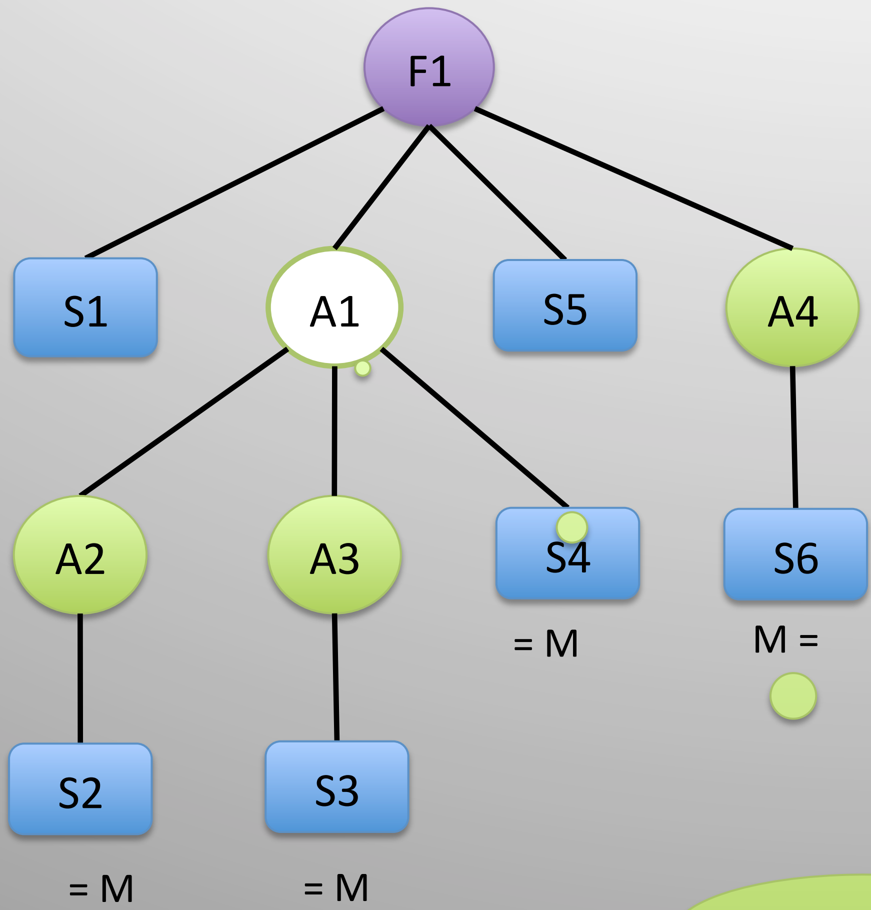
SPD3 Example



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 $M_s.r_2$

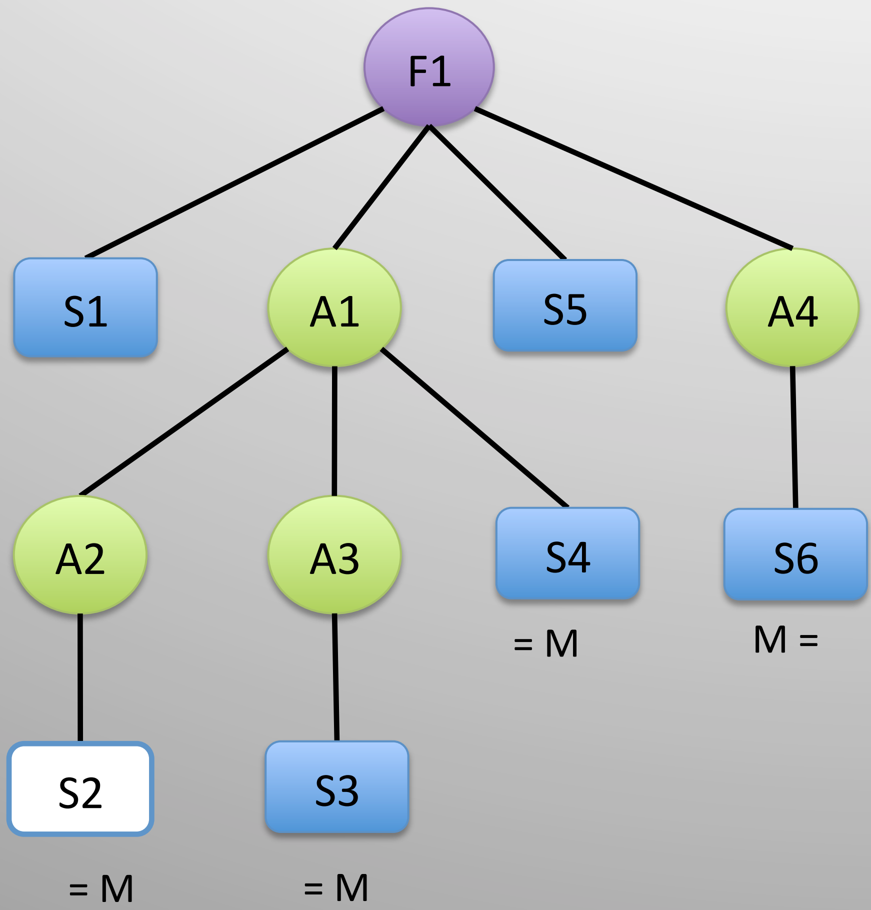
SPD3 Example



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

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

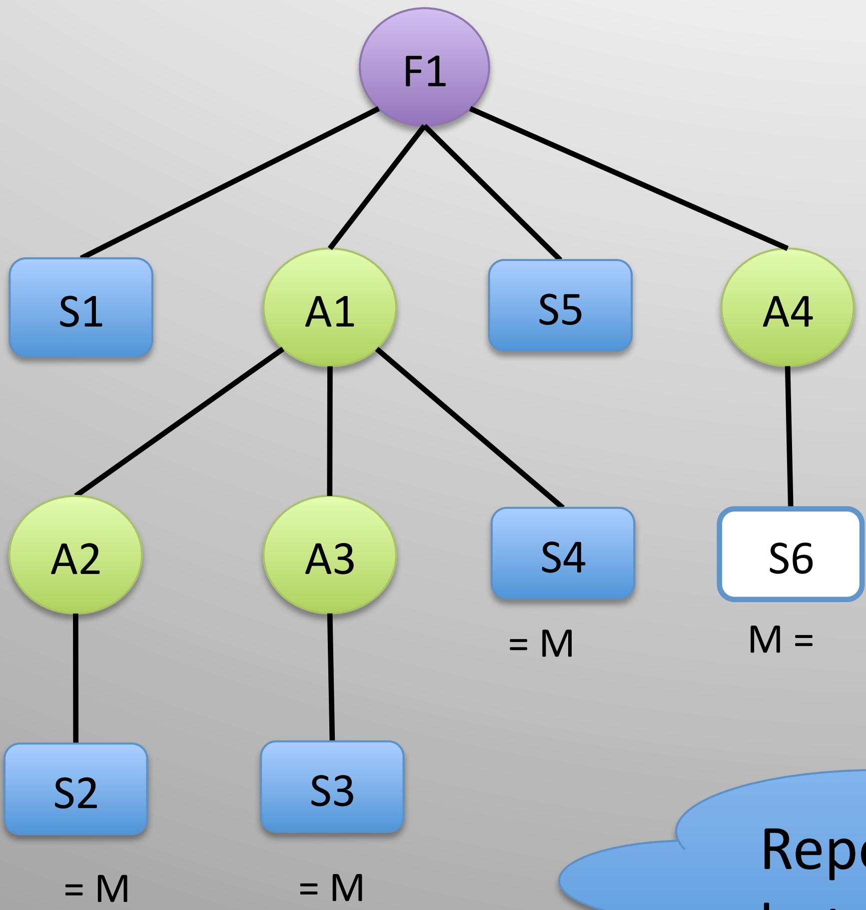
SPD3 Example



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

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
S6 (Write M)			

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

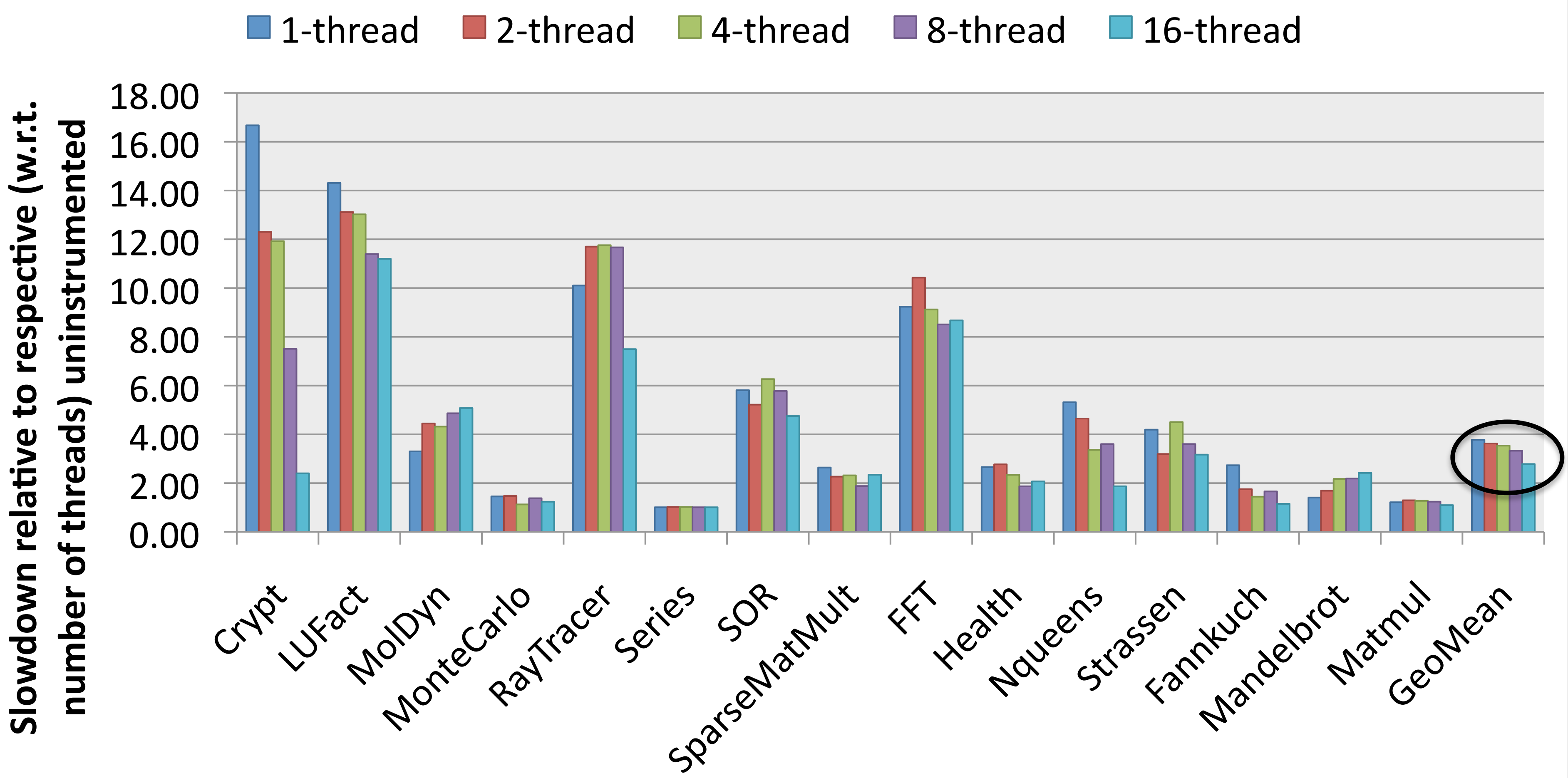
Space Overhead

- 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

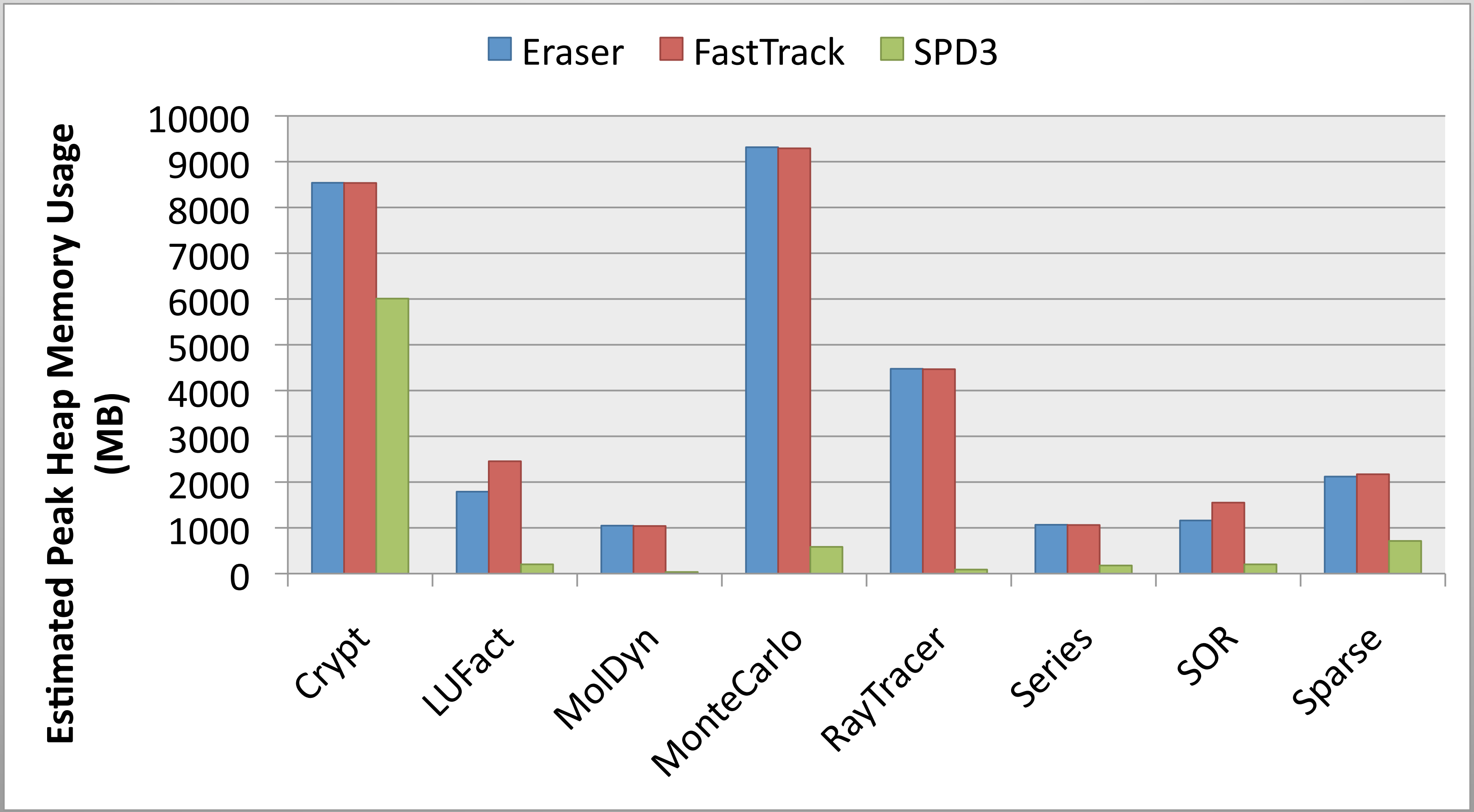
Empirical Evaluation

- 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

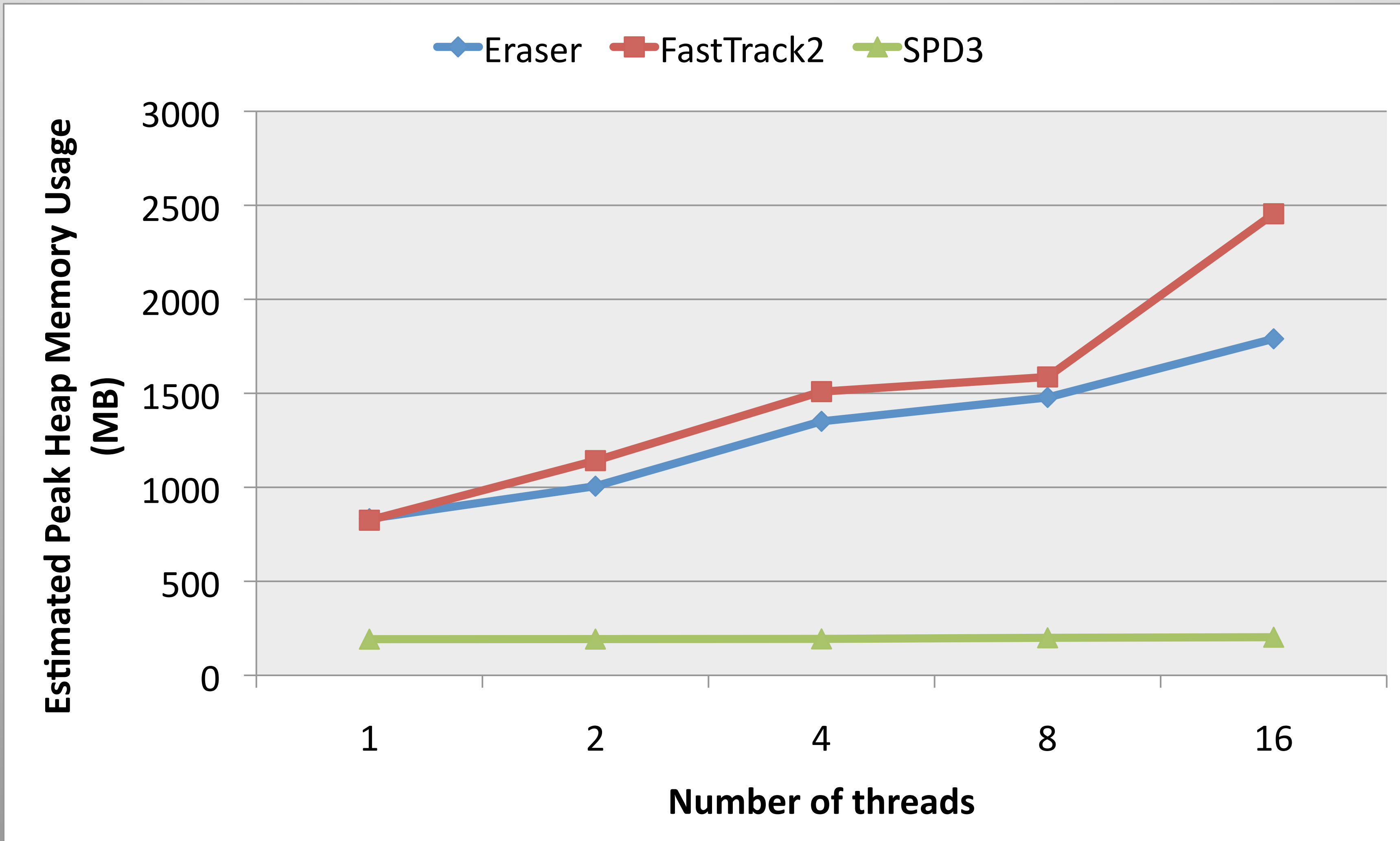


Estimated Peak Heap Memory Usage on 16-threads



RICE

Estimated Peak Heap Memory Usage LUFact Benchmark



Related Work: A Comparison

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

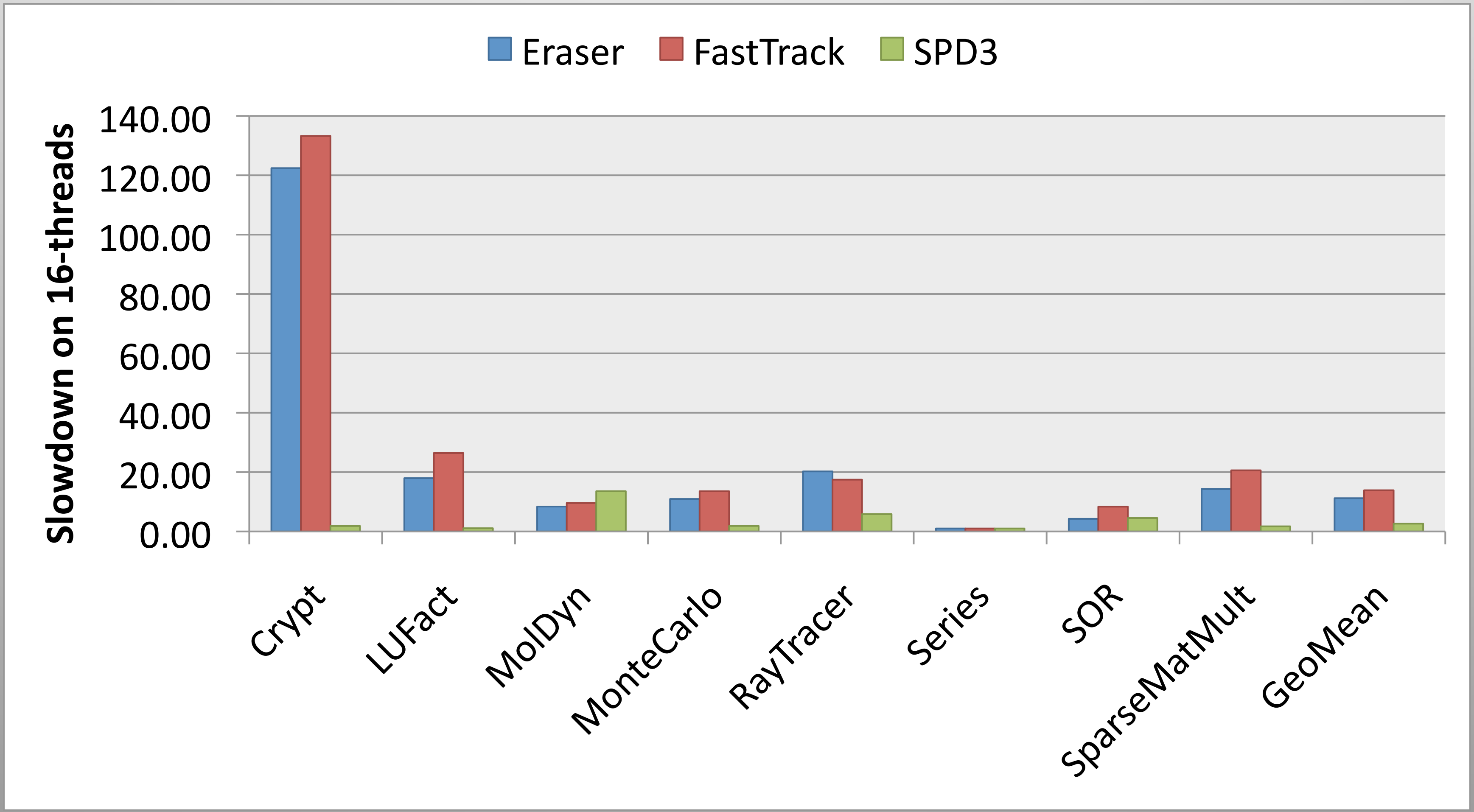
OTFDAA – On the fly detection of access anomalies (PLDI '89)
n – number of threads executing the program
N – maximum logical concurrency in the program

Summary

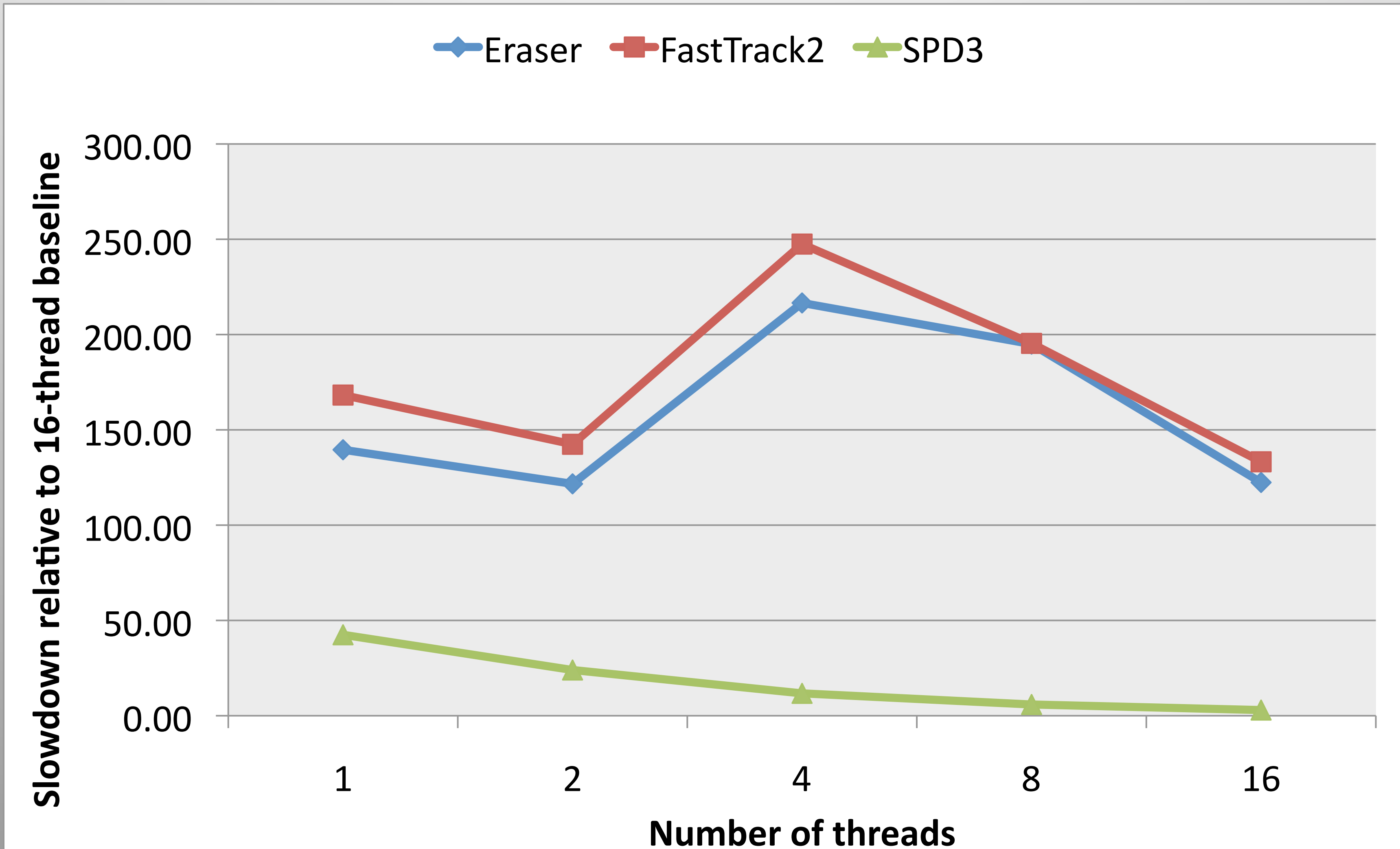
- First practical datarace detector which is parallel with constant space overhead
  - Dynamic Program Structure Tree
  - Access Summary

Backup

Slowdown on 16-threads



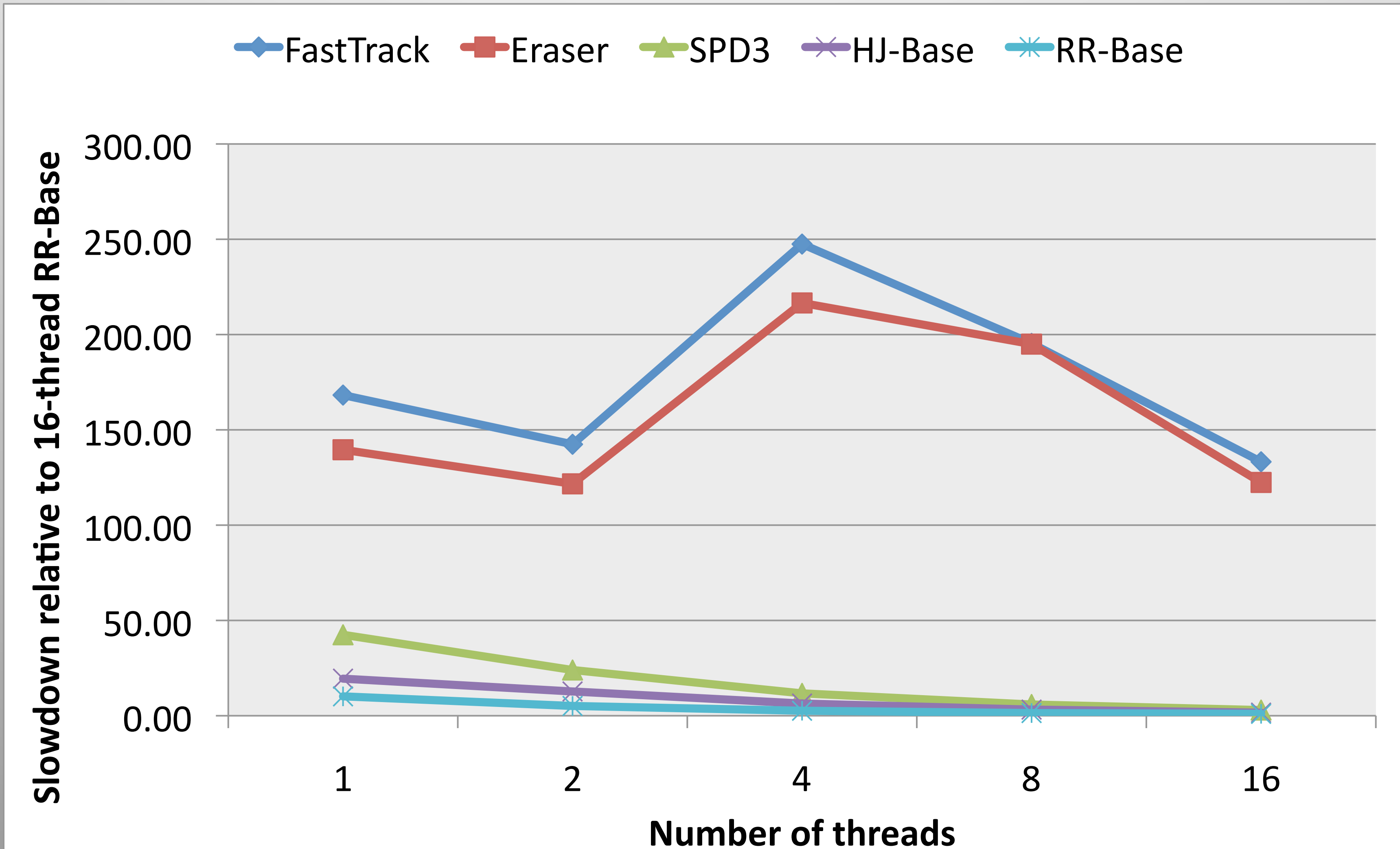
Slowdown of Crypt Benchmark



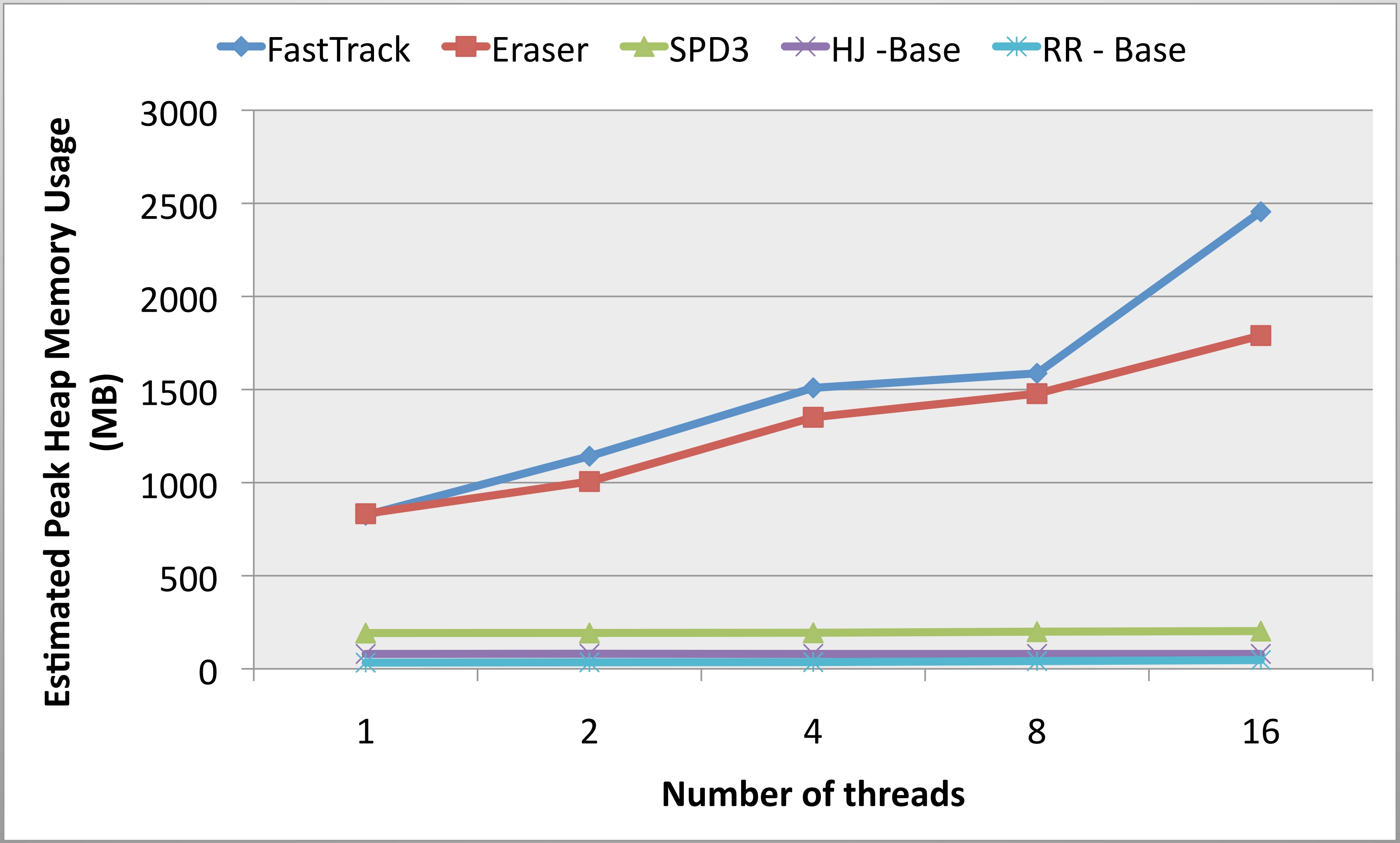
Comparison with Eraser and FastTrack

- 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



Estimated Peak Heap Memory Usage
LUFact Benchmark



Slowdown on 16-threads

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

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.

RICE