A Parallel Abstract Interpreter for JavaScript

Kyle Dewey, Vineeth Kashyap, Ben Hardekopf

University of California, Santa Barbara

Overall Contributions

- A fundamentally new perspective on parallelizing static analysis
- A parallel static analysis for JavaScript based on this new perspective
- Improved speedups over closely related analyses
- Hypothesize that more parallelization is possible, with relevant data

Outline

- Background
- Prior work and core insight
- Evaluation
- Conclusions

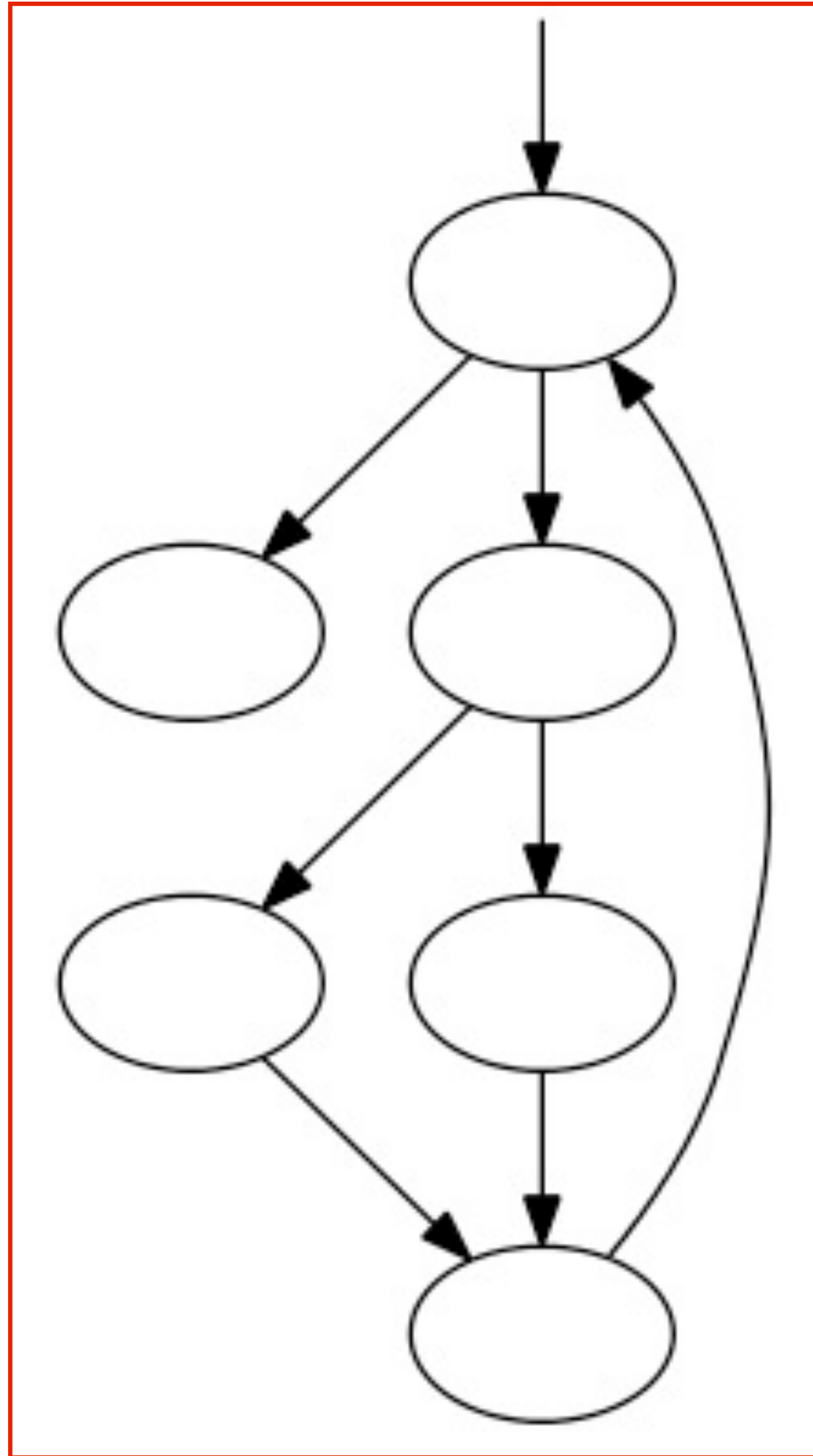
Outline

- Background
- Prior work and core insight
- Evaluation
- Conclusions

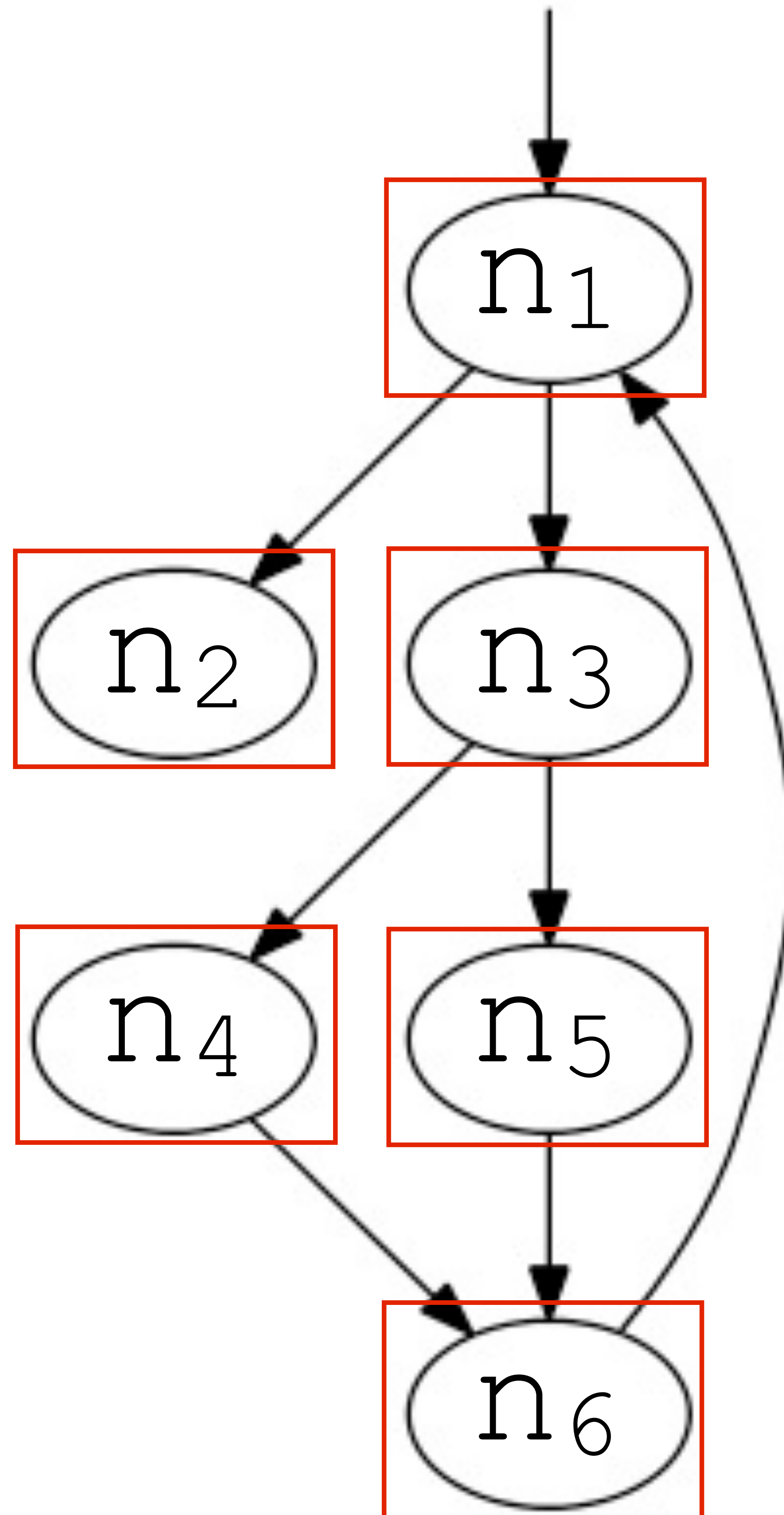
Dataflow Analysis

- Over a program's control flow graph
- Each node represents an equation to solve
- Edges define interdependencies between equations
    - Overall, a system of equations
- Find a fixpoint of the system

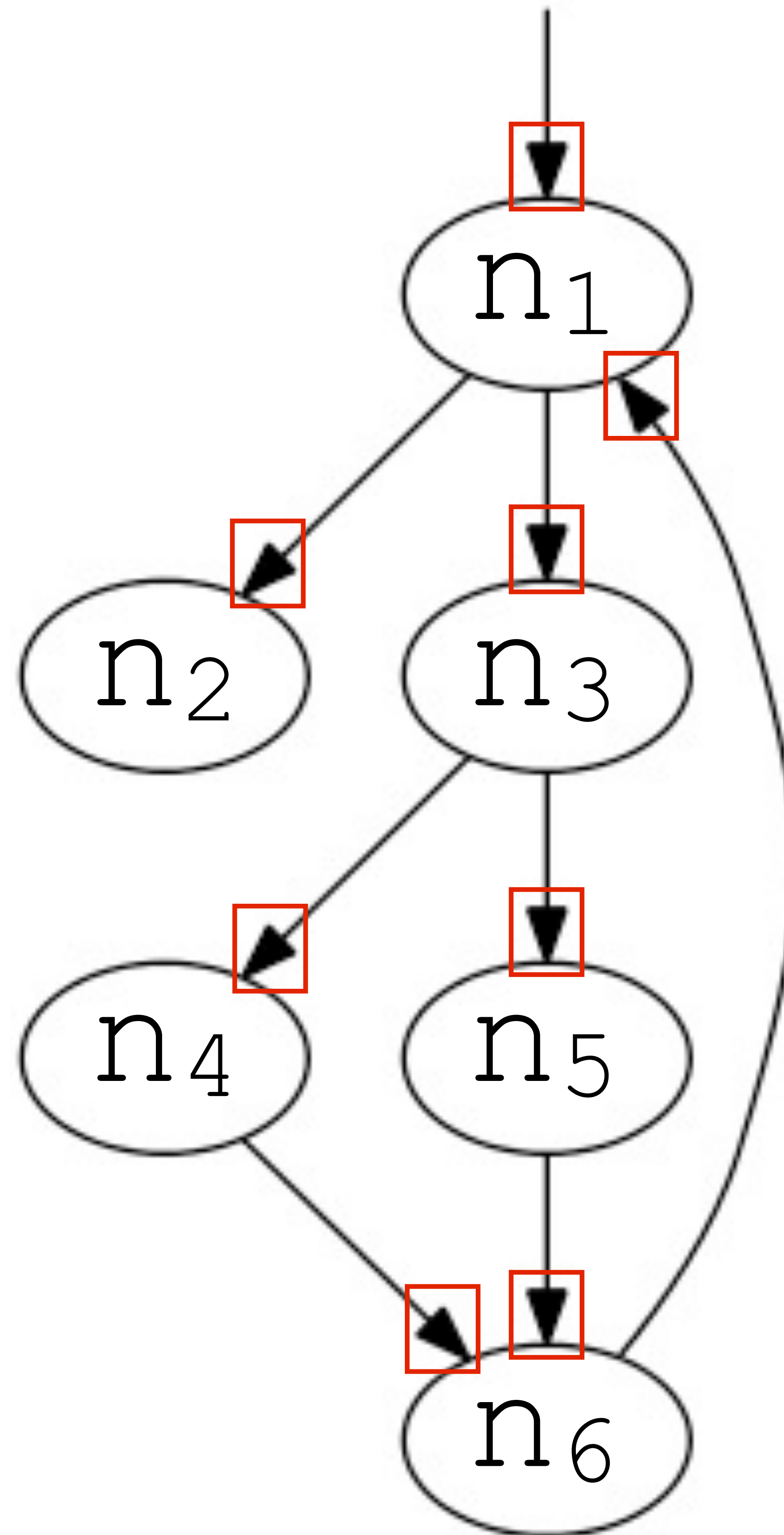
Traditional Dataflow Analysis



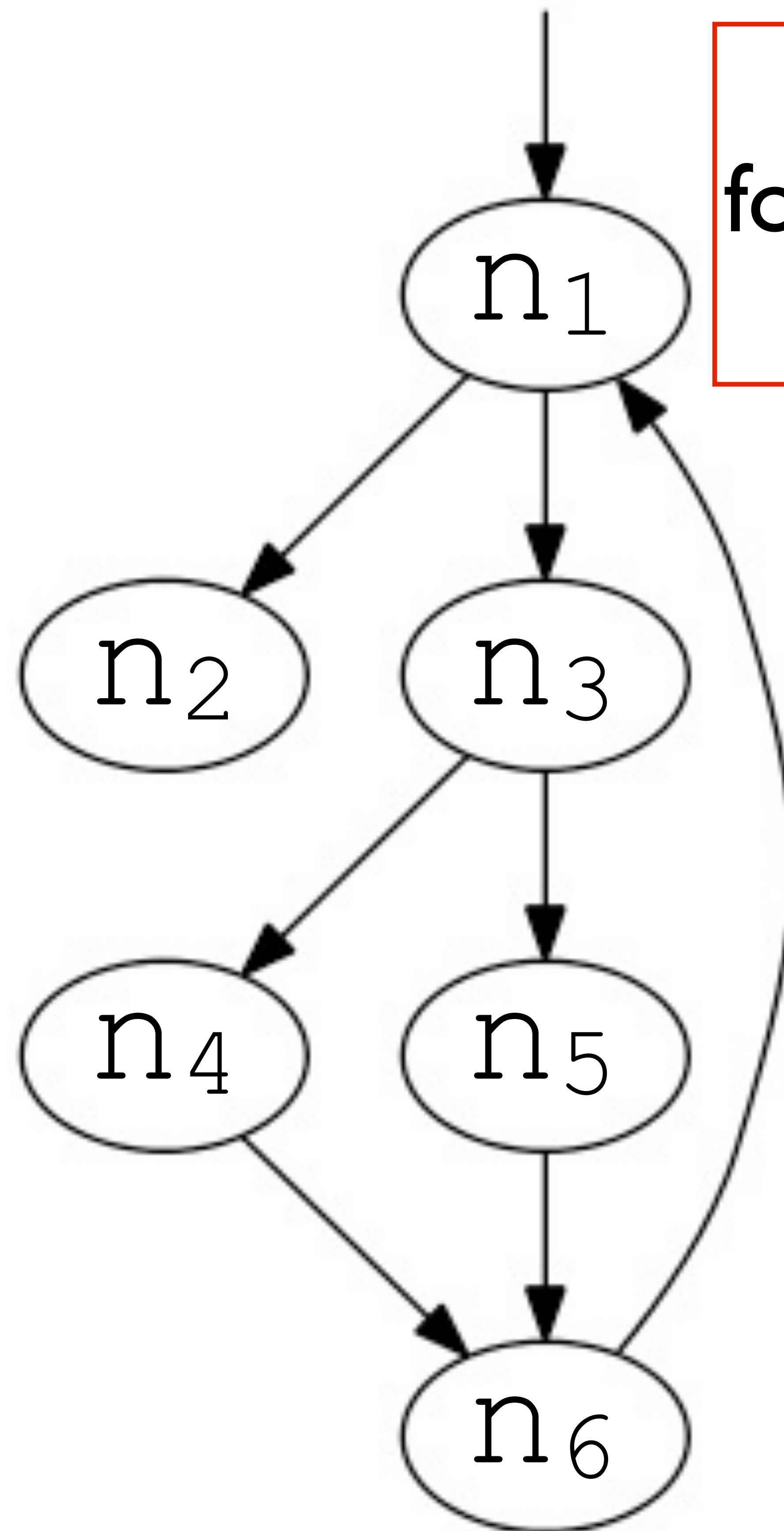
Traditional Dataflow Analysis



Traditional Dataflow Analysis

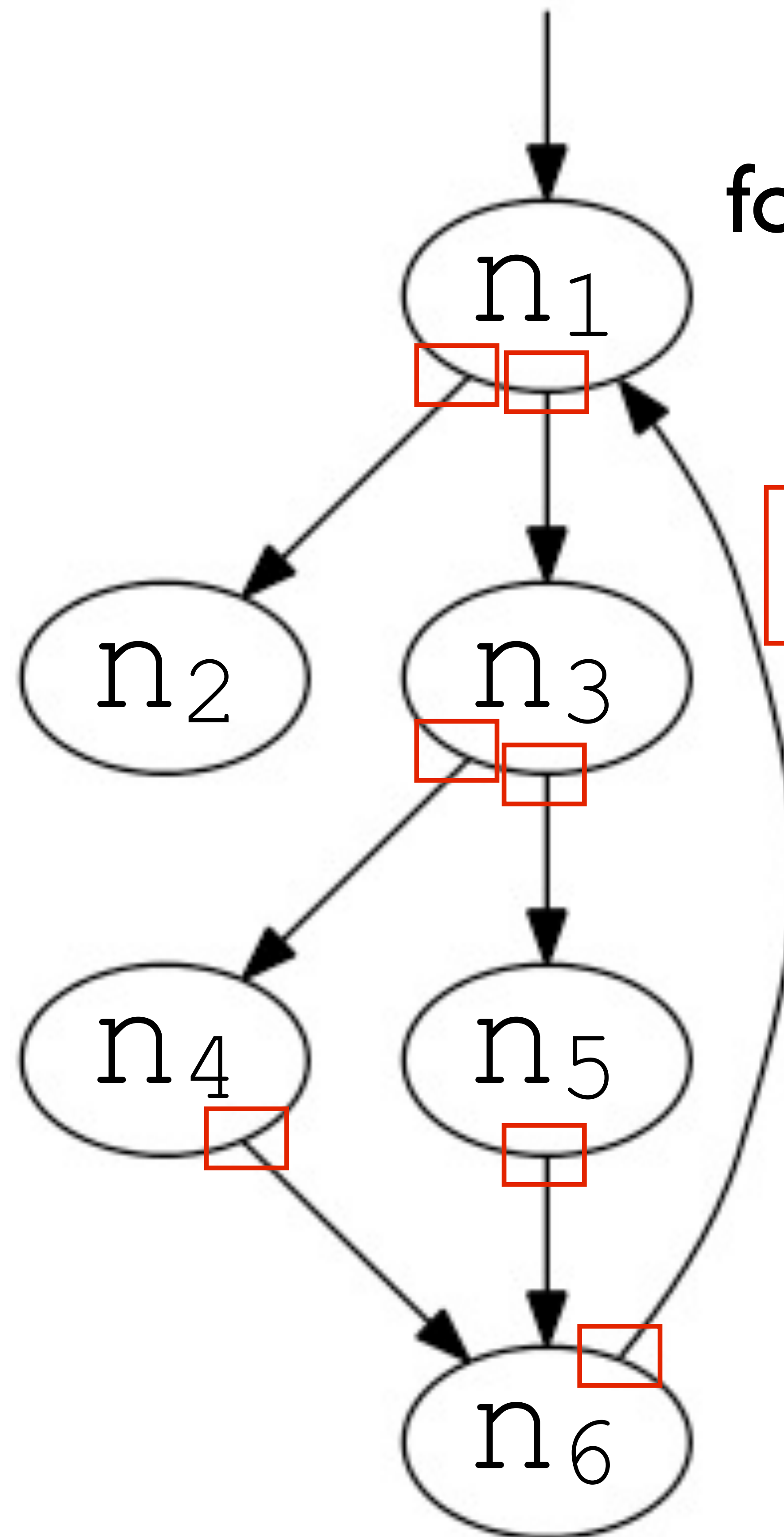


Traditional Dataflow Analysis



IN_k = meet(OUT_x)
for all predecessors x
of k

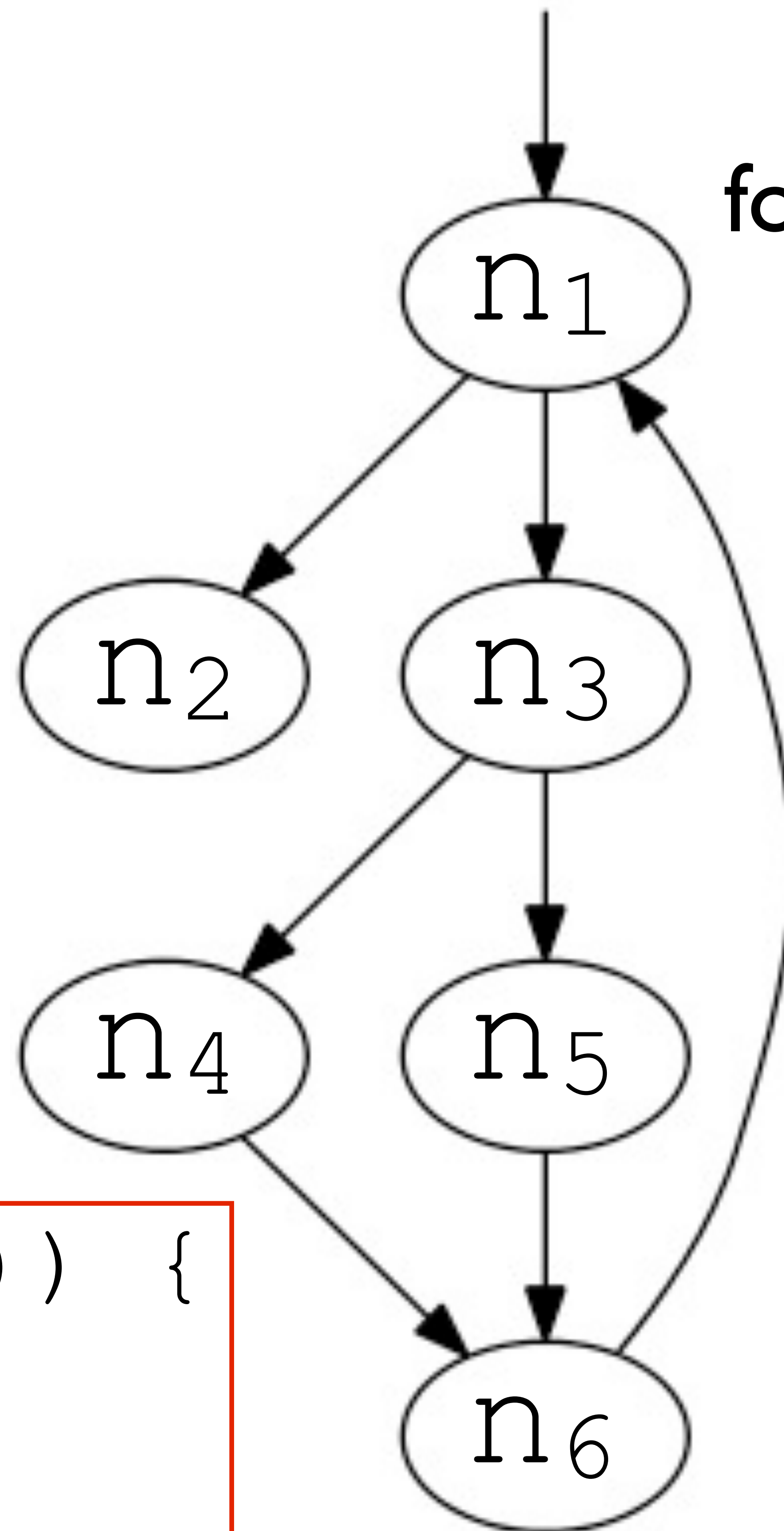
Traditional Dataflow Analysis



IN_k = meet(OUT_x) for all predecessors x of k

OUT_k = Xfer(IN_k)

Traditional Dataflow Analysis



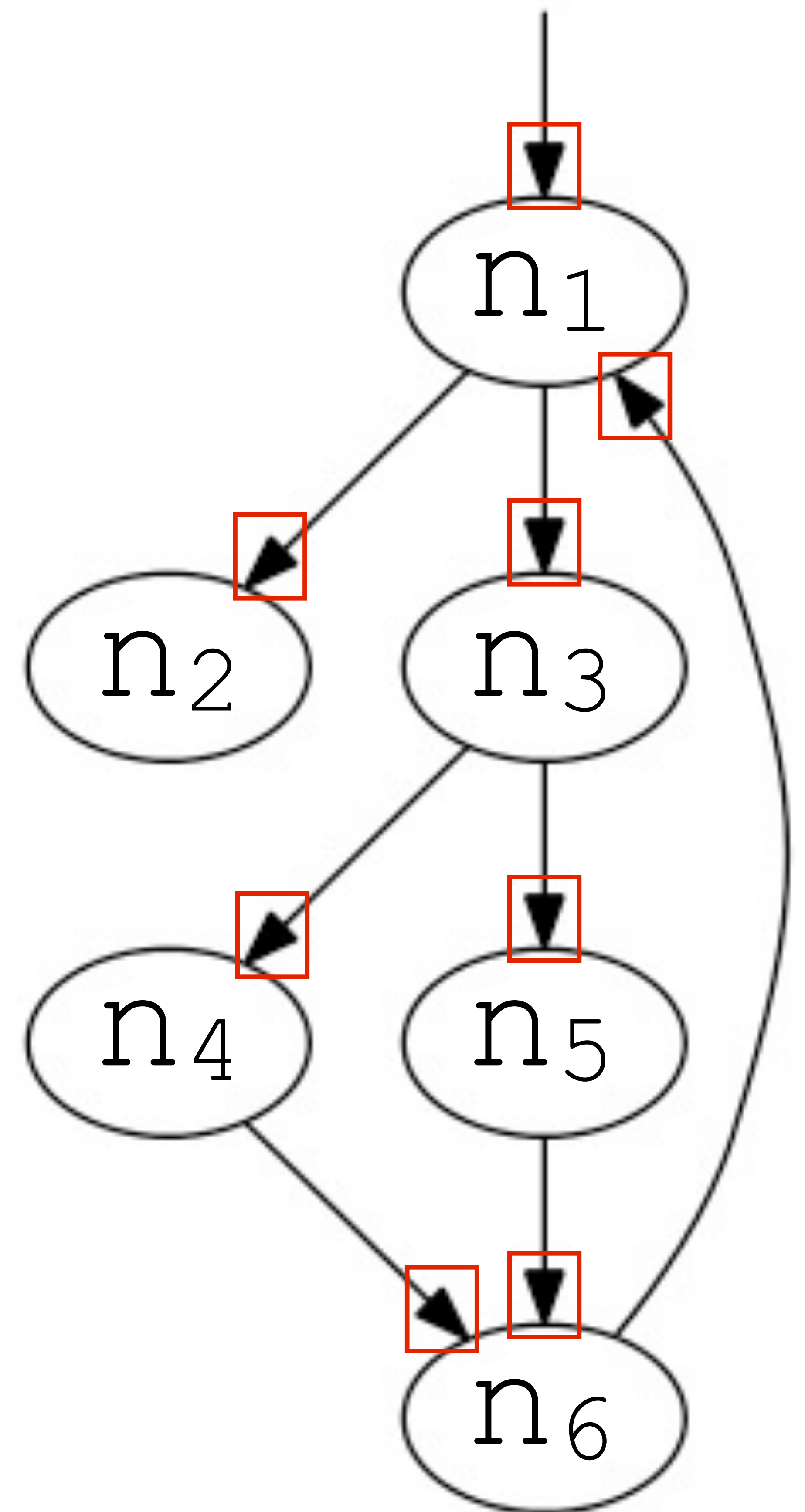
IN_k = meet(OUT_x) for all predecessors x of k

OUT_k = Xfer(IN_k)

```
while (!fix()) {
  repeat();
}
```

Parallelization Problem

- Meets require synchronization and imply sequential dependencies
- Meets are everywhere



To maximize parallelism, we need an alternative analysis perspective

Outline

- Background
- Prior work and core insight
- Evaluation
- Conclusions

State Transition Representation

- Prior work: represent program analysis as a graph reachability problem on an infinite state transition system with state merging

State Transition Representation

- Prior work: represent program analysis as a graph reachability problem on an infinite state transition system with state merging
    - Is a particular possible program state reachable from some initial program state?

State Transition Representation

- Prior work: represent program analysis as a graph reachability problem on an infinite state transition system with state merging
  - Rules for deriving a new state from an existing state
  - Potentially an infinite number of states
  - Start from some initial program state

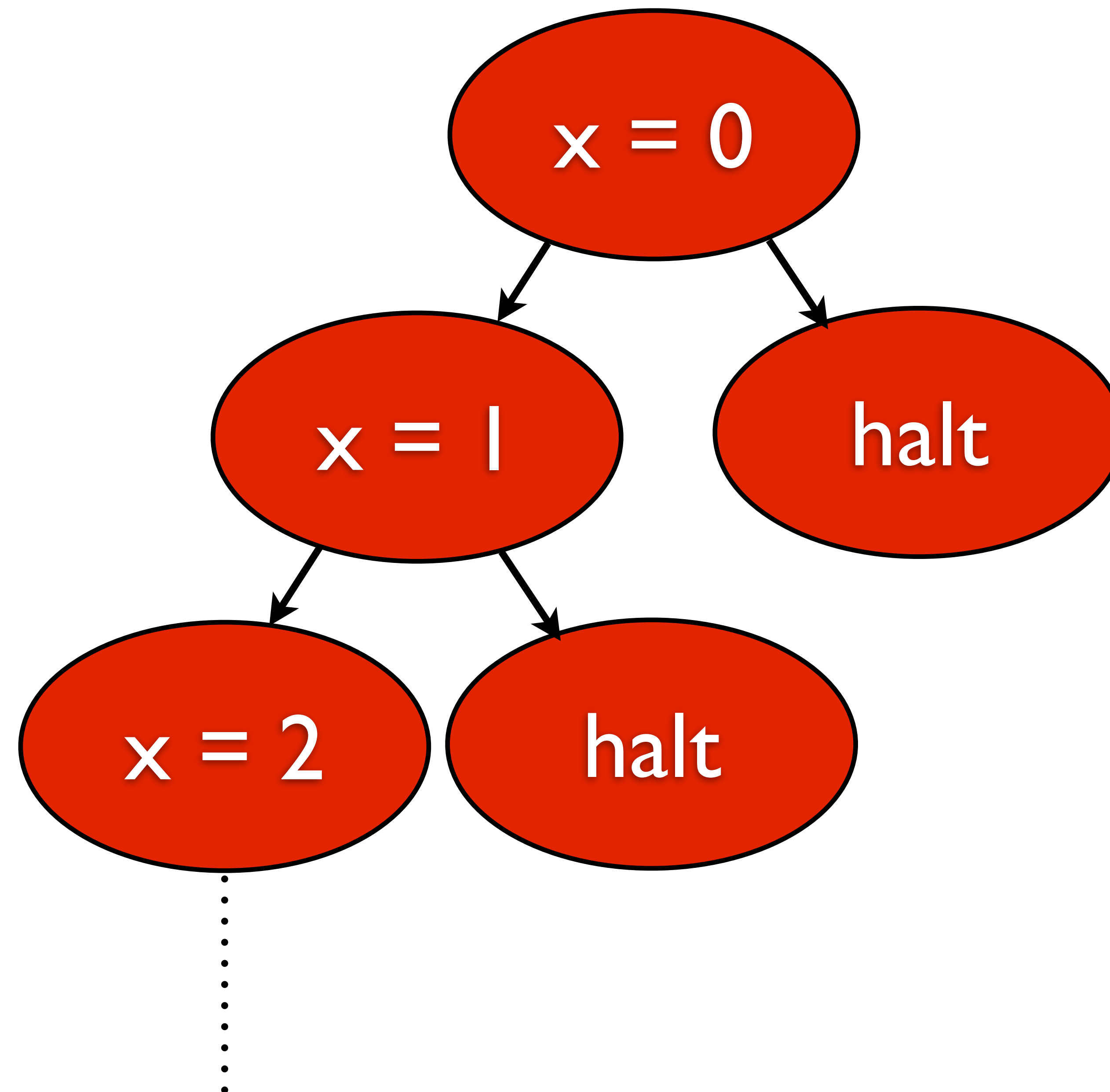
State Transition Representation

- Prior work: represent program analysis as a graph reachability problem on an infinite state transition system with state merging
  - Selectively merge states to keep things reasonably finite
  - Many different merging strategies are possible, and correspond to different analysis sensitivities (e.g., k-CFA)

Example

```
1: int x = 0;
2: while (randBool()) {
3:   x++;
4: }
```

```
1: int x = 0;
2: while (randBool()) {
3:   x++;
4: }
```

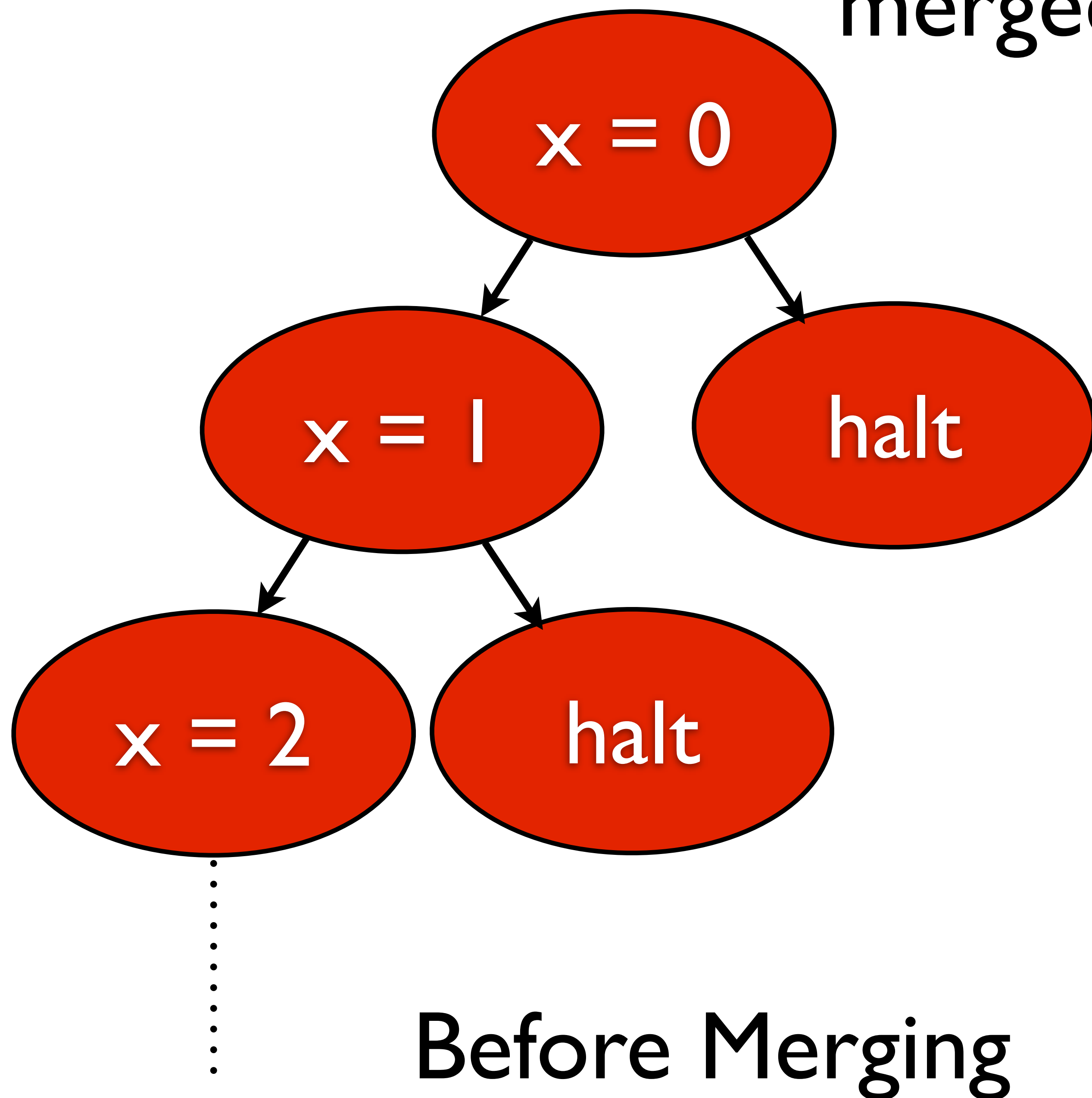


```
1: int x = 0;
2: while (randBool()) {
3:   x++;
4: }
```

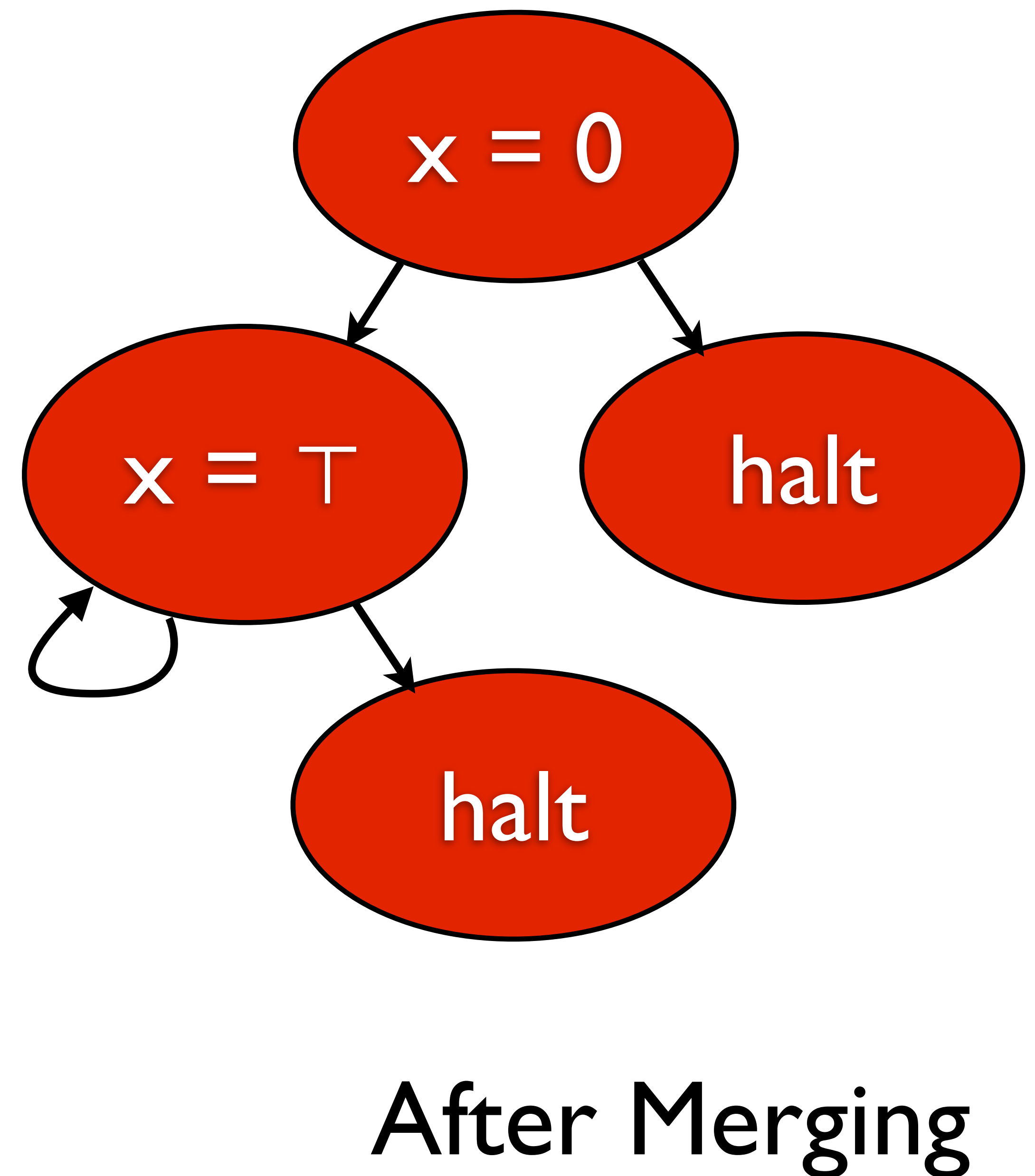
Merging strategy: all states at the same line of code are merged together

```
1: int x = 0;
2: while (randBool()) {
3:   x++;
4: }
```

Merging strategy: all states at the same line of code are merged together



Before Merging



After Merging

Core Insight: This Parallelizes Well

- State reachability over a tree is a massively parallel problem
- We can reason about the analysis separately from the state merging component
  - The analysis itself need not change

Core Insight: This Parallelizes Well

- State merging strategies selectively impart sequential dependencies
  - Dependencies are specific to a strategy
  - Much smaller component than the whole analysis

Assigning Threads

- A separate problem from defining the analysis and where sequential dependencies lie
  - Many possible assignment policies
  - All three problems can vary independently in this definition

Dataflow Analysis as an Instantiation

- Traditional merging strategy: merge at every operation performed by the program
  - For precision this is fine - not necessarily all states will be merged together
  - For parallelism, this is poor - lots of synchronization is needed

Outline

- Background
- Prior work and core insight
- Evaluation
- Conclusions

Application: JavaScript

- Parallelized a sequential JavaScript analysis defined in prior work
- The feature set of JavaScript makes deriving a precise control flow graph unrealistic
  - Traditional dataflow analysis is impossible

Merging Strategy

- Two states are merged together if:
  - They occur at the same point of the program (e.g., line 10)
  - The top `k` functions on the call stack are the same
- The program point along with the call stack snippet is a ***context***

```
function foo() {
  foo(); // P
}
```

```
k = 3
```

Program Point `P`
`foo:main`
`foo:foo:main`
`foo:foo:foo`

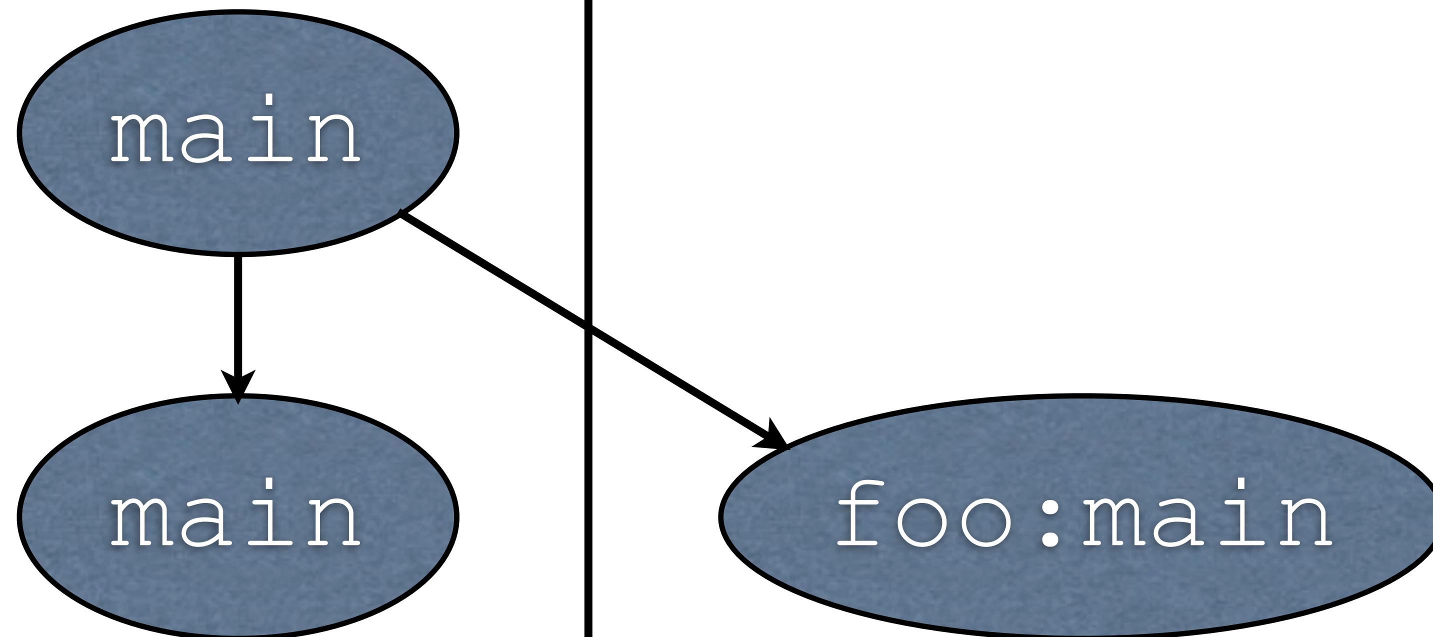
Thread Assignment Strategy

- Program states in distinct contexts are assigned distinct threads
- Program states in the same context are uniformly assigned to the same thread

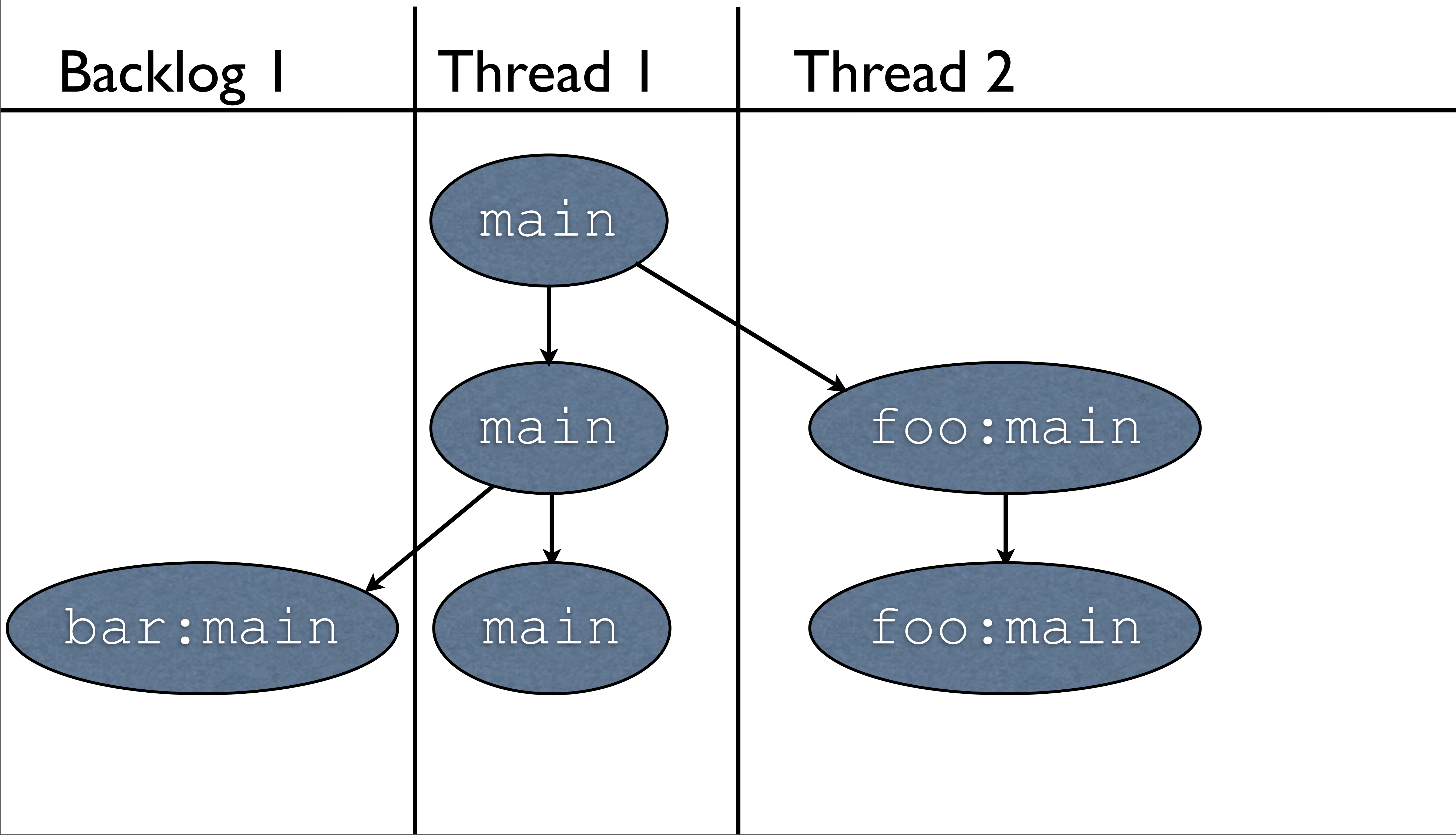
Thread Assignment

Thread 1	Thread 2
main	

Thread Assignment

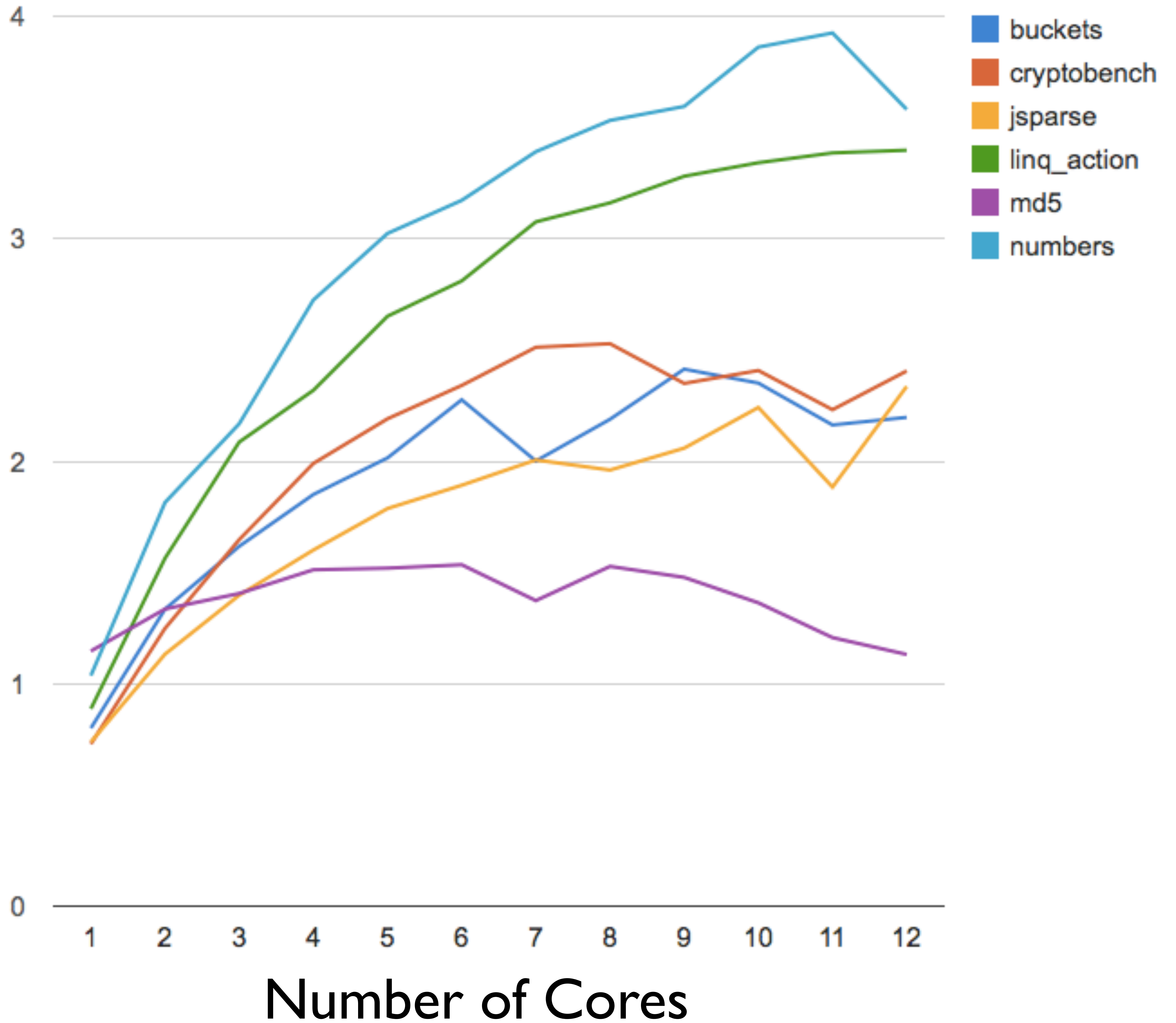


Thread Assignment



Evaluation

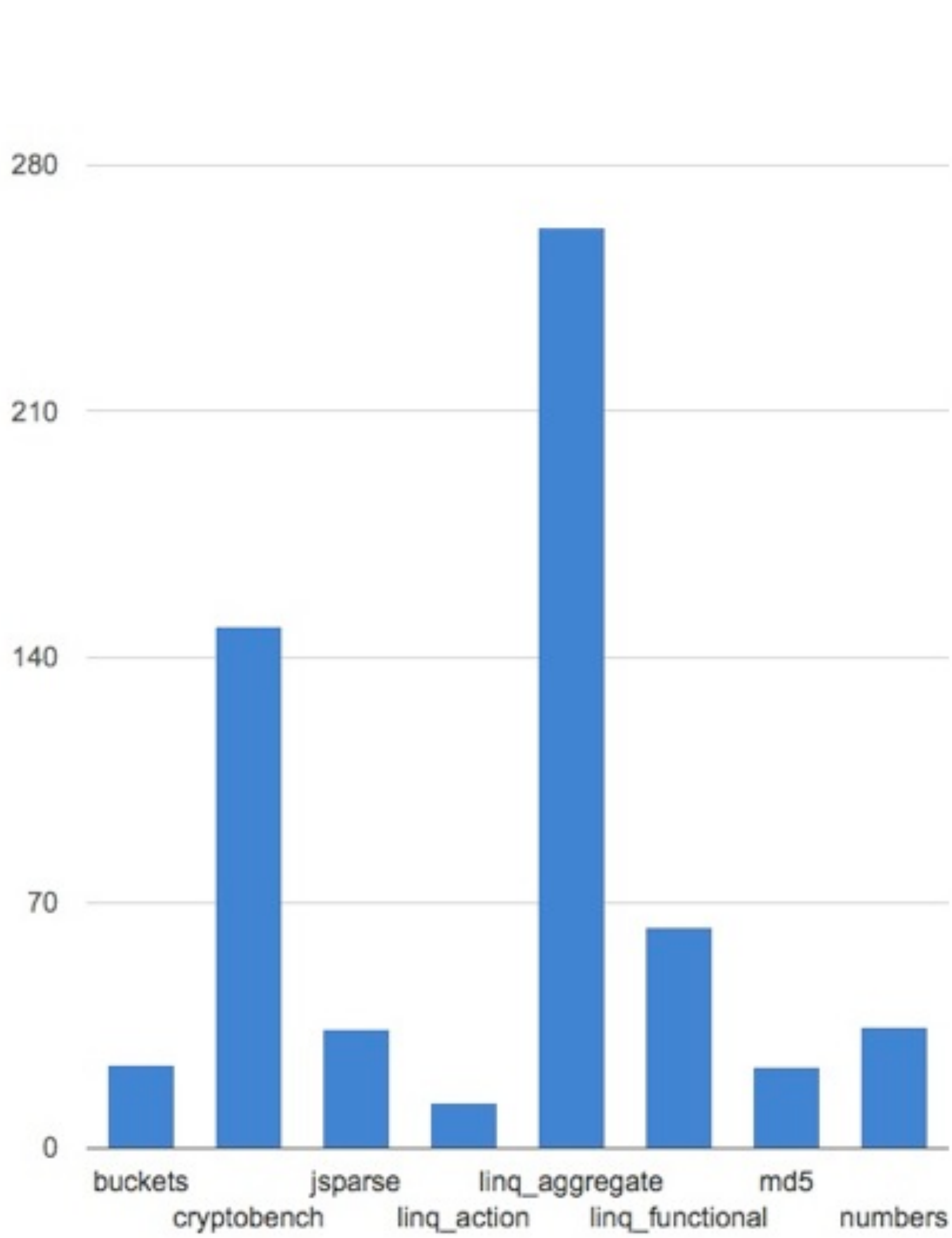
- On a series of open source real-world benchmarks taking between 30s and 20m
- Recording true speedups (i.e., relative to the preexisting sequential framework)
  - Measure of scale and performance

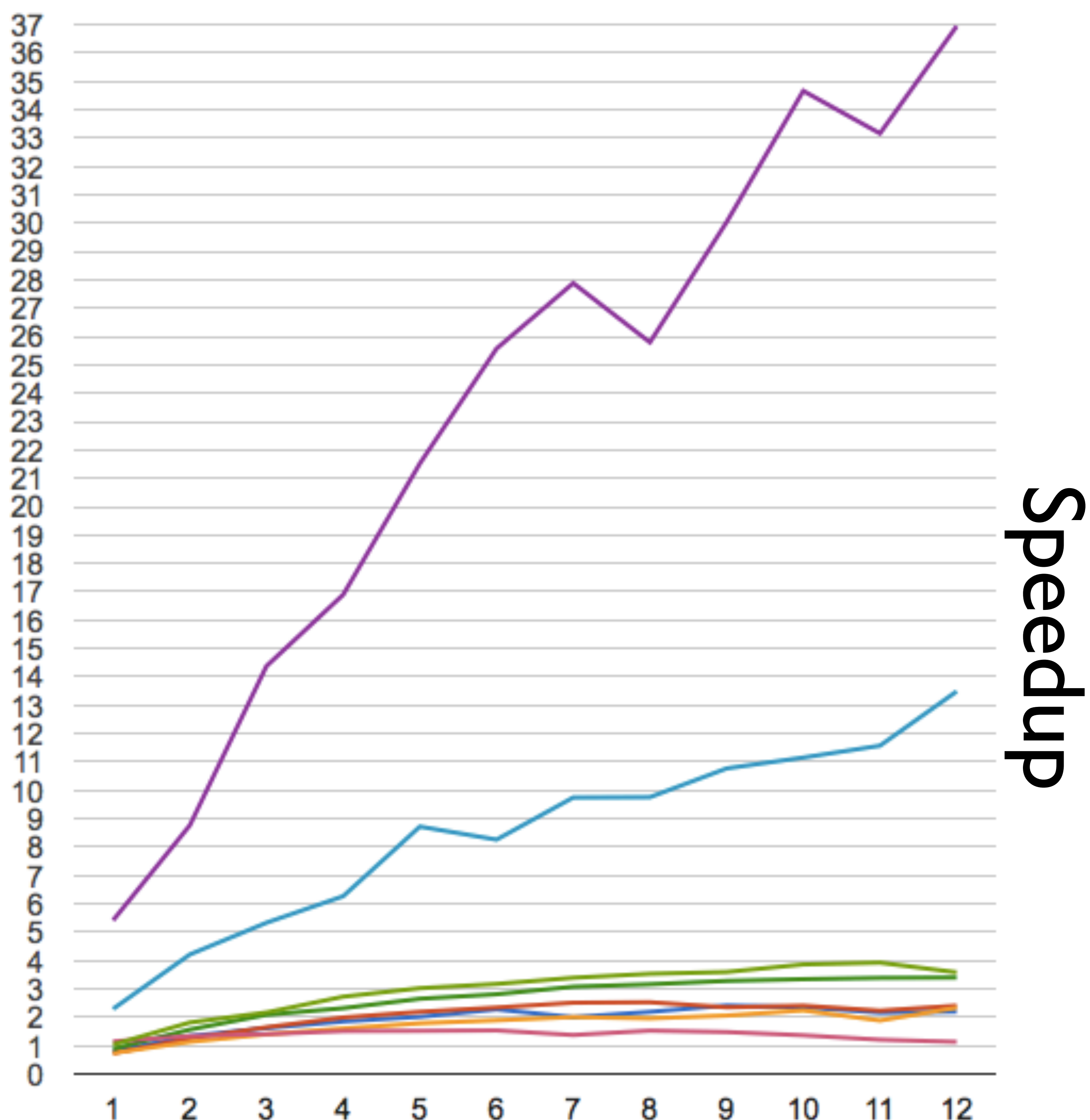

Speedup
4
3
2
1
0
1
2
3
4
5
6
7
8
9
10
11
12
Number of Cores
buckets
cryptobench
jsparse
linq_action
md5
numbers

On Using One Thread Per Context

- Recall: we assign one thread per context
- If this is optimal, then more contexts should mean better scalability and performance

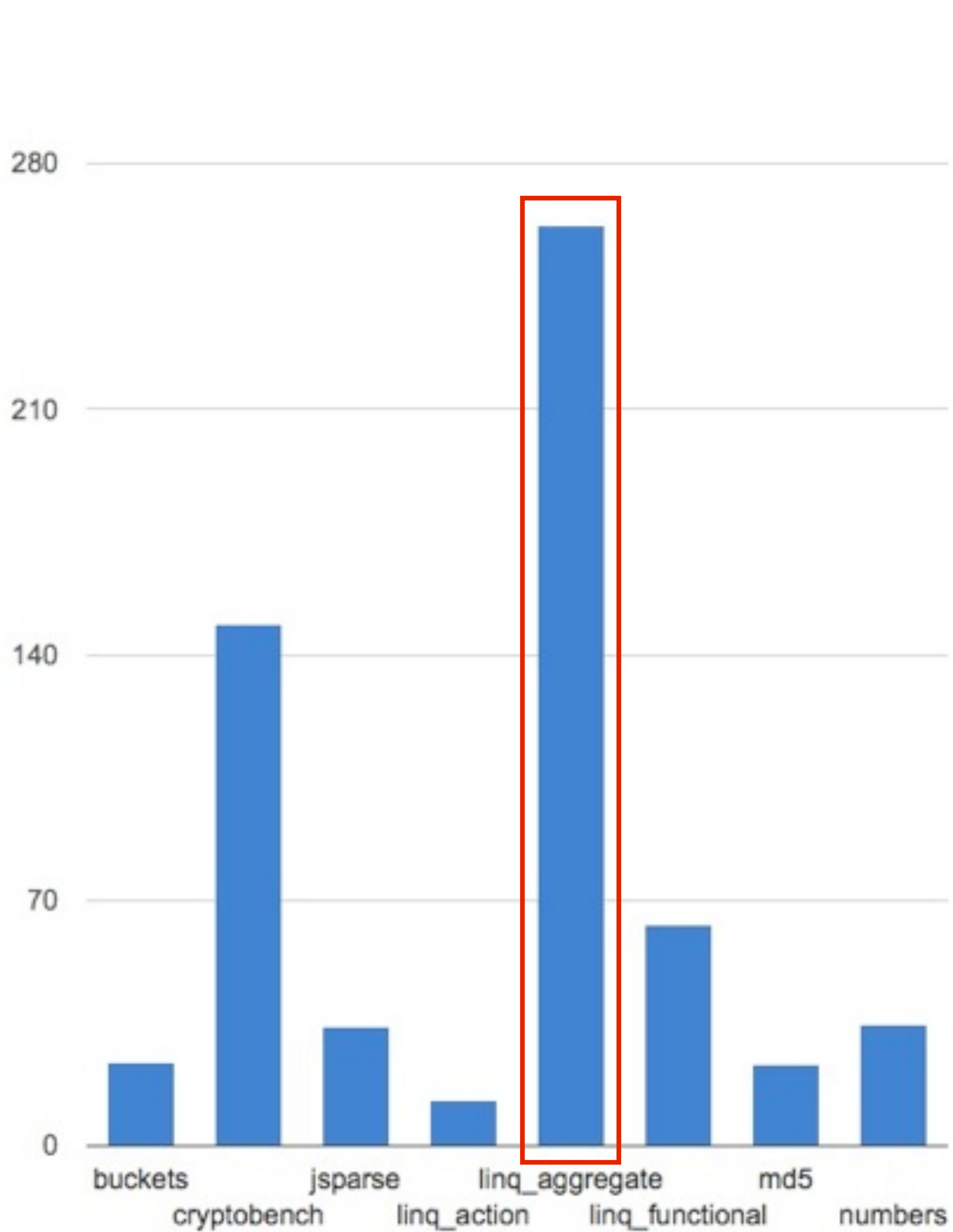
Average Number of Available Contexts

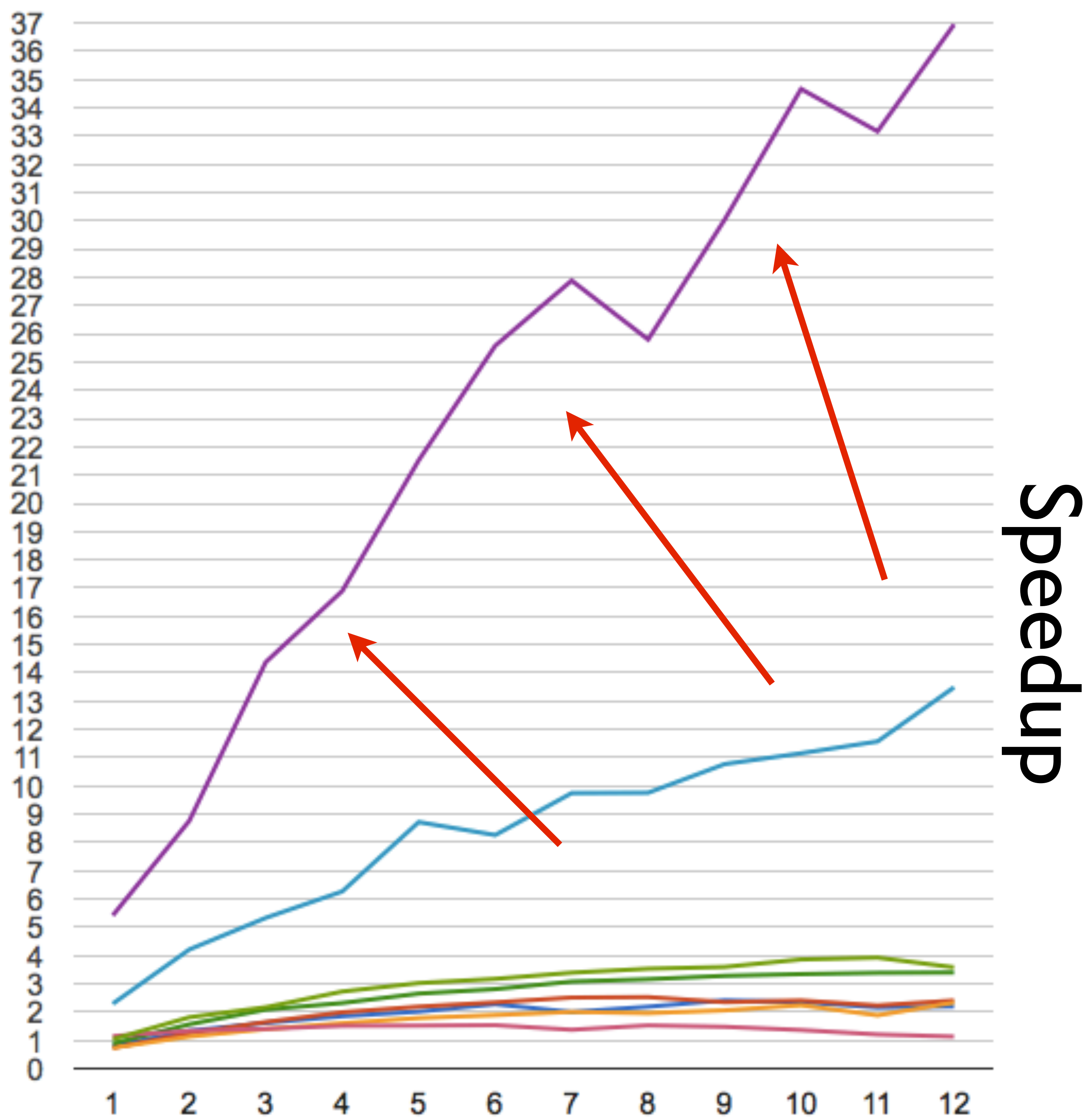




buckets cryptobench jsparse linq_action linq_aggregate linq_functional md5 numbers

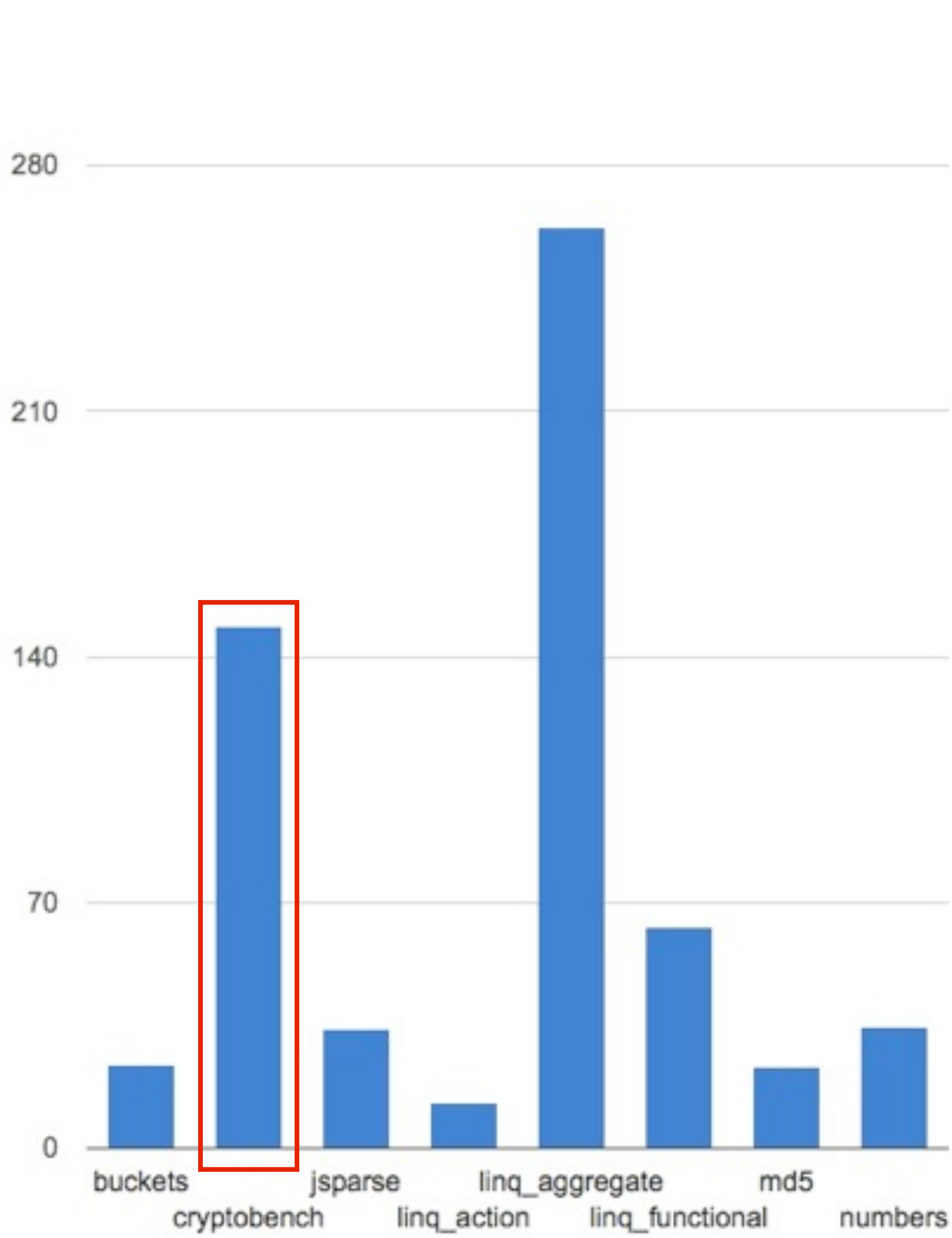
Average Number of Available Contexts

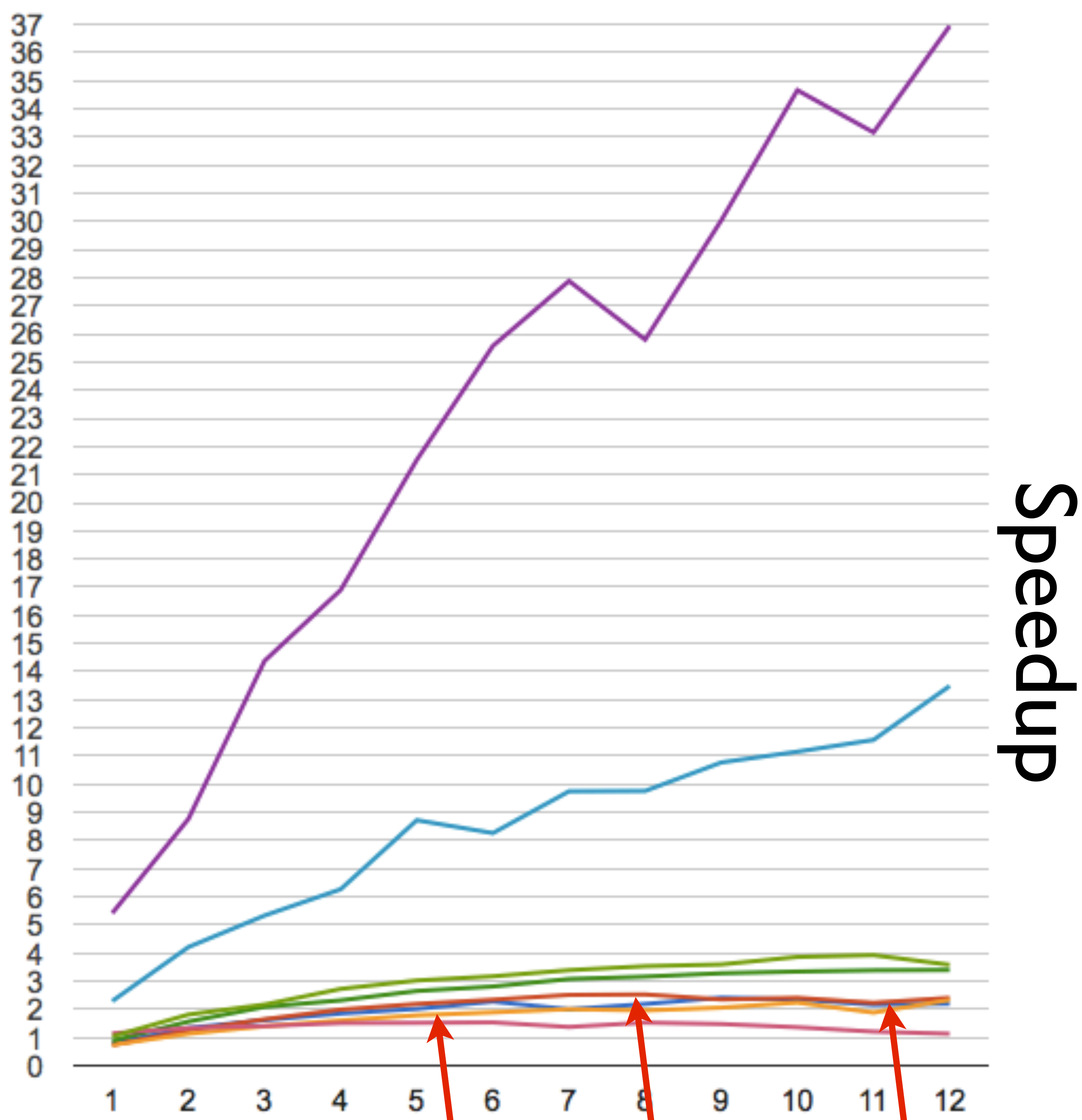




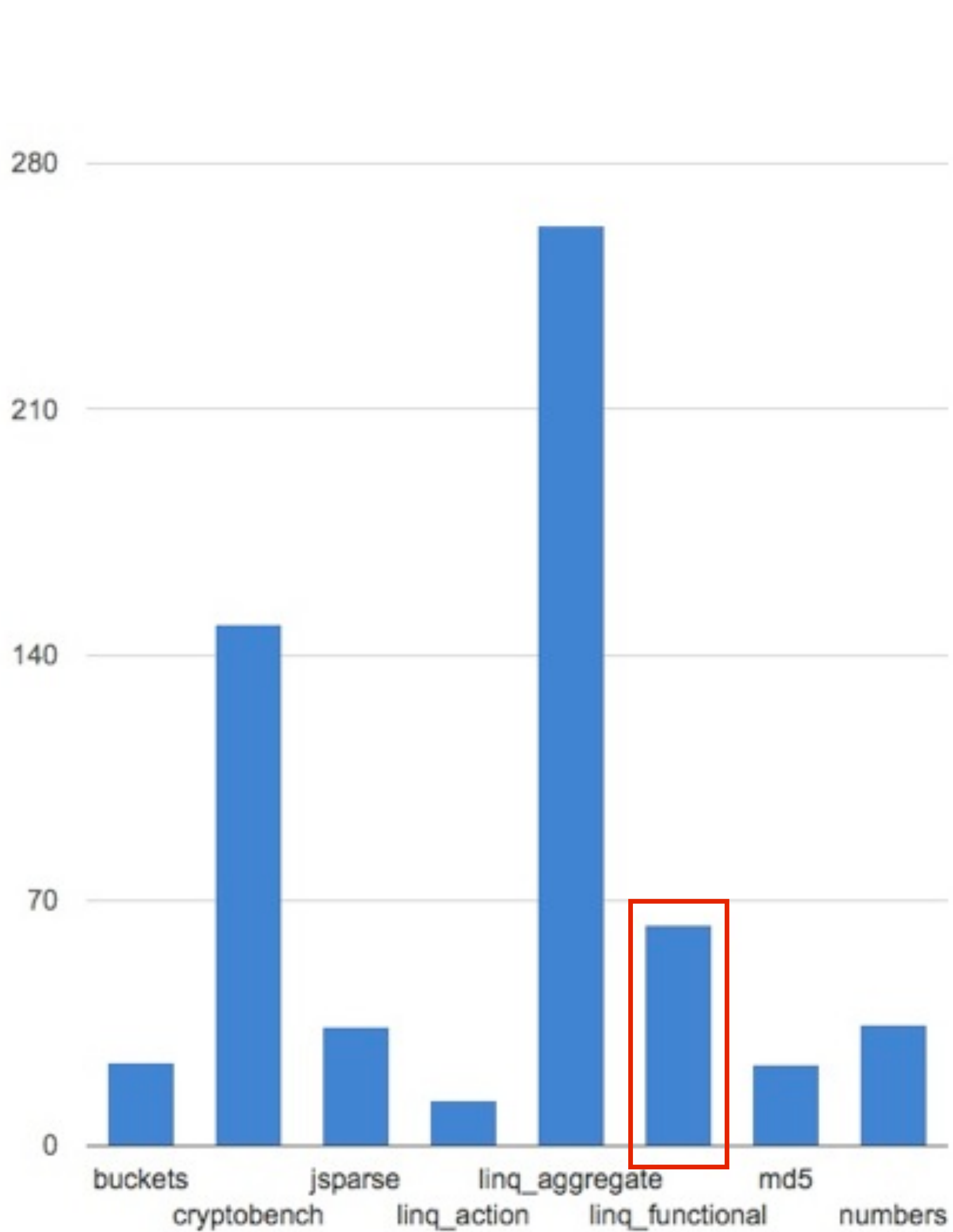
buckets cryptobench jsparse linq_action linq_aggregate linq_functional md5 numbers

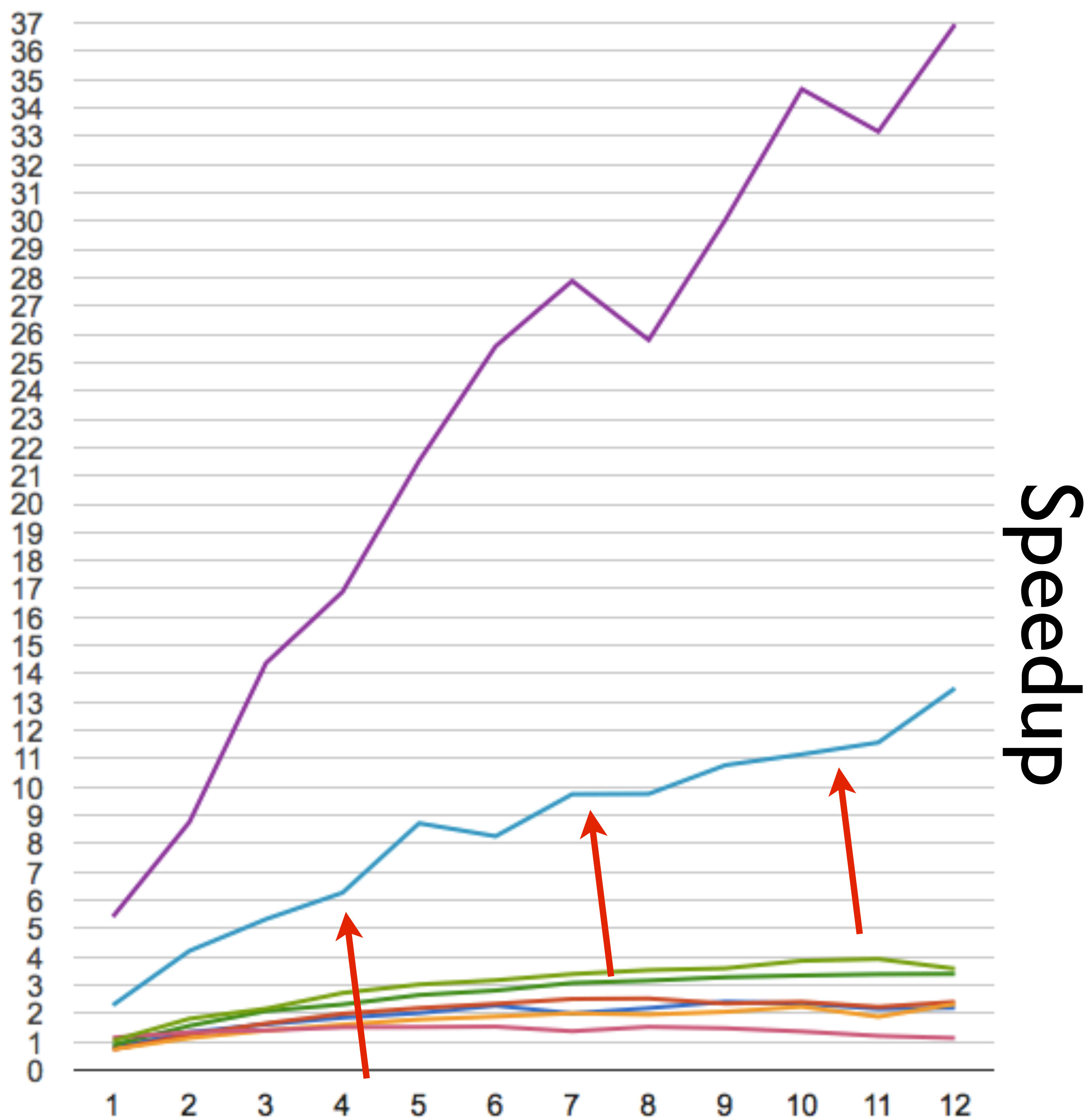
Average Number of Available Contexts





buckets cryptobench jsparse linq_action linq_aggregate linq_functional md5 numbers


Average Number of Available Contexts
280
210
140
70
0
buckets
cryptobench
jsparse
linq_action
linq_aggregate
linq_functional
md5
numbers


Speedup
37
36
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
9
10
11
12
buckets
cryptobench
jsparse
linq_action
linq_aggregate
linq_functional
md5
numbers

Outline

- Background
- Prior work and core insight
- Evaluation
- Conclusions

Conclusions

- Our perspective on analysis is inherently parallel, unlike traditional dataflow analysis
- We see performance which is typically superior to related work
- Much improvement can still be made in assigning threads for better performance

Future Work

- Parallel experimentation with other merging and thread assignment strategies
- Application to C
    - Would allow for direct comparison to related work