A Parallel Abstract Interpreter for JavaScript

Kyle Dewey Ben Hardekopf

University of California, Santa Barbara

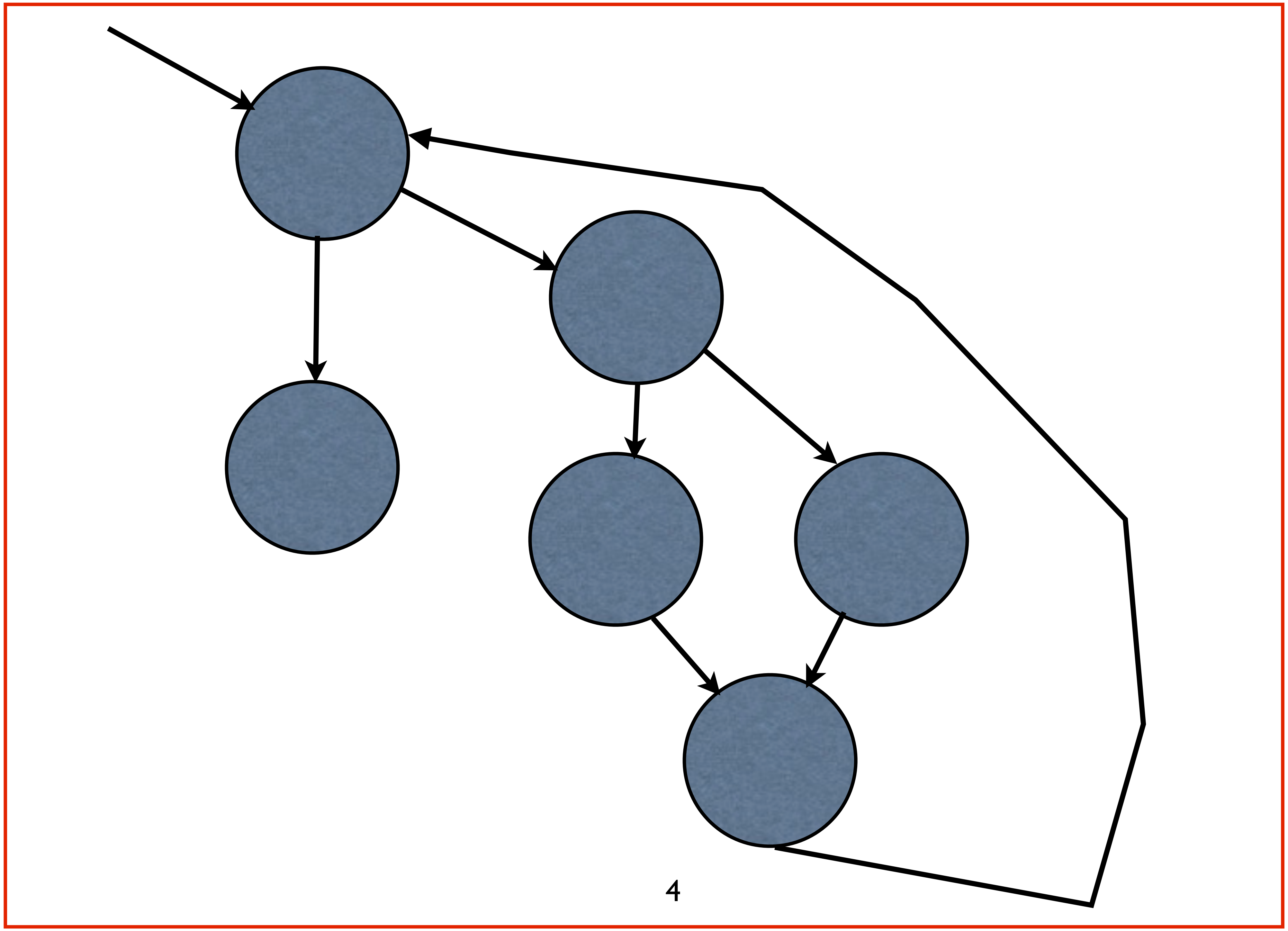
Overall Contributions

- New analysis perspective established in prior work is far more amenable to parallelization than dataflow analysis
- A parallel abstract interpreter for JavaScript based on this new perspective
- Speedups usually better than those of the most closely related program analyses (typically between 2-4X on 12 threads)

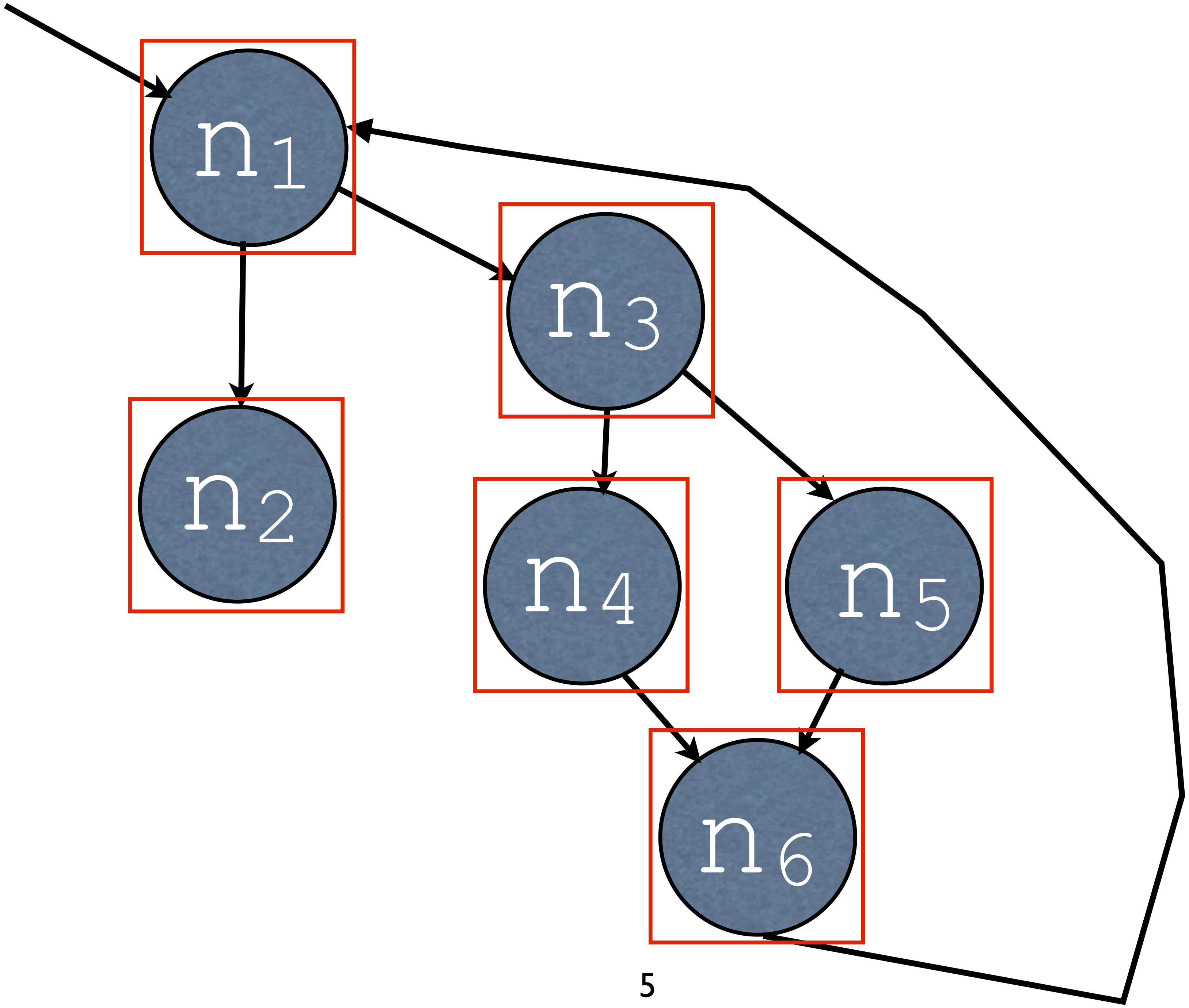
Dataflow Analysis

- 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

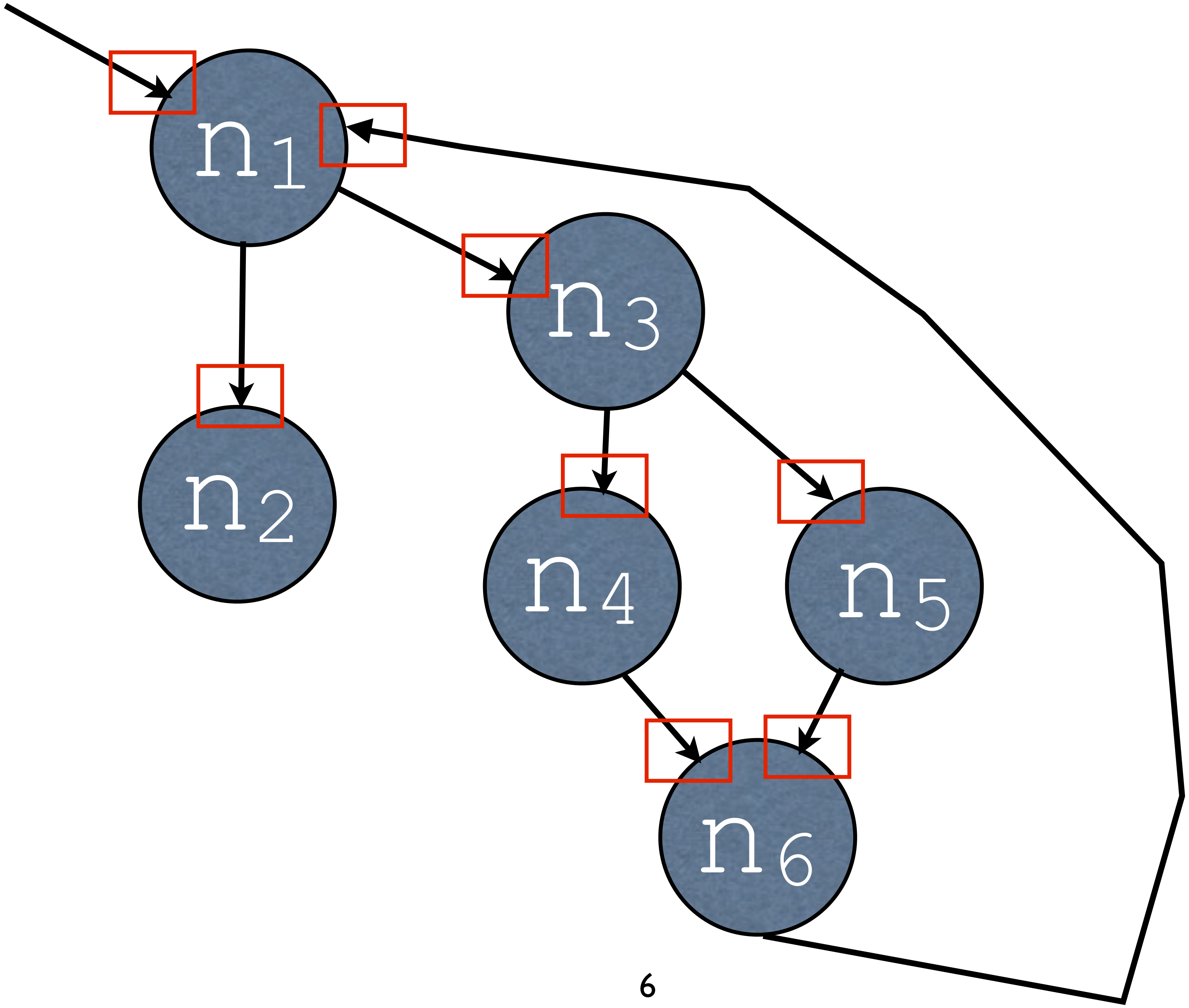
Traditional Dataflow Analysis



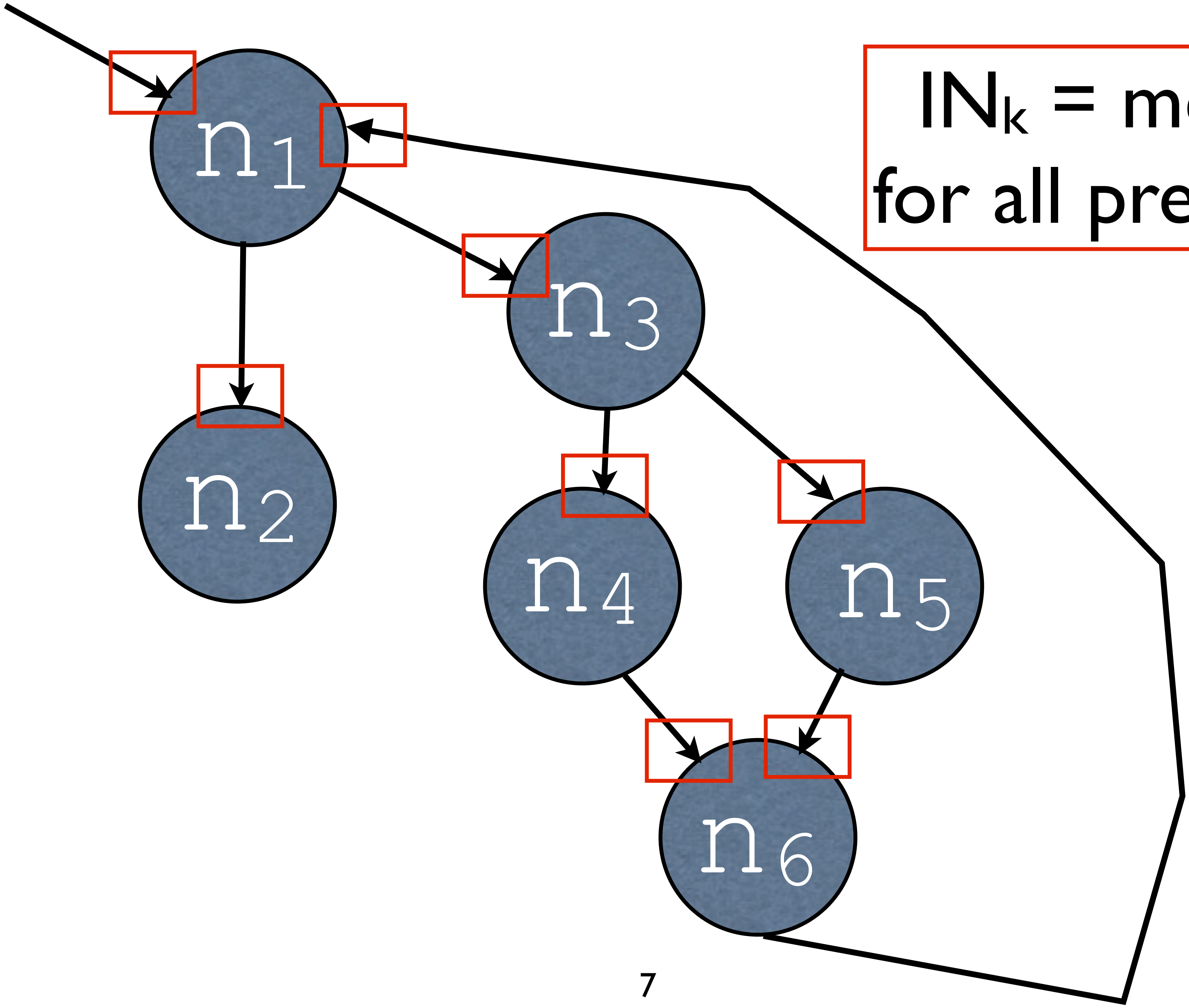
Traditional Dataflow Analysis



Traditional Dataflow Analysis

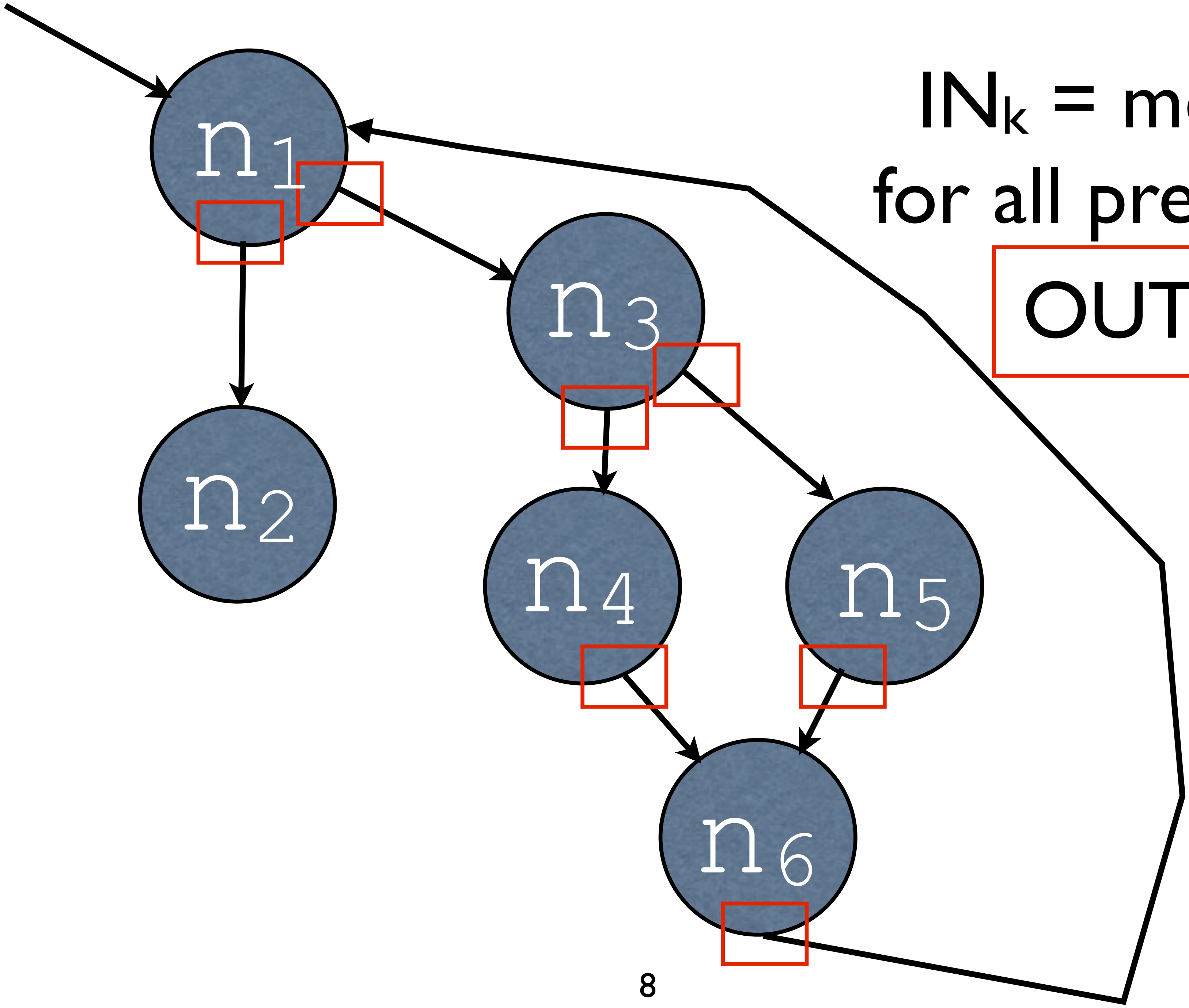


Traditional Dataflow Analysis



IN_k = meet(OUT_x)
for all predecessors x

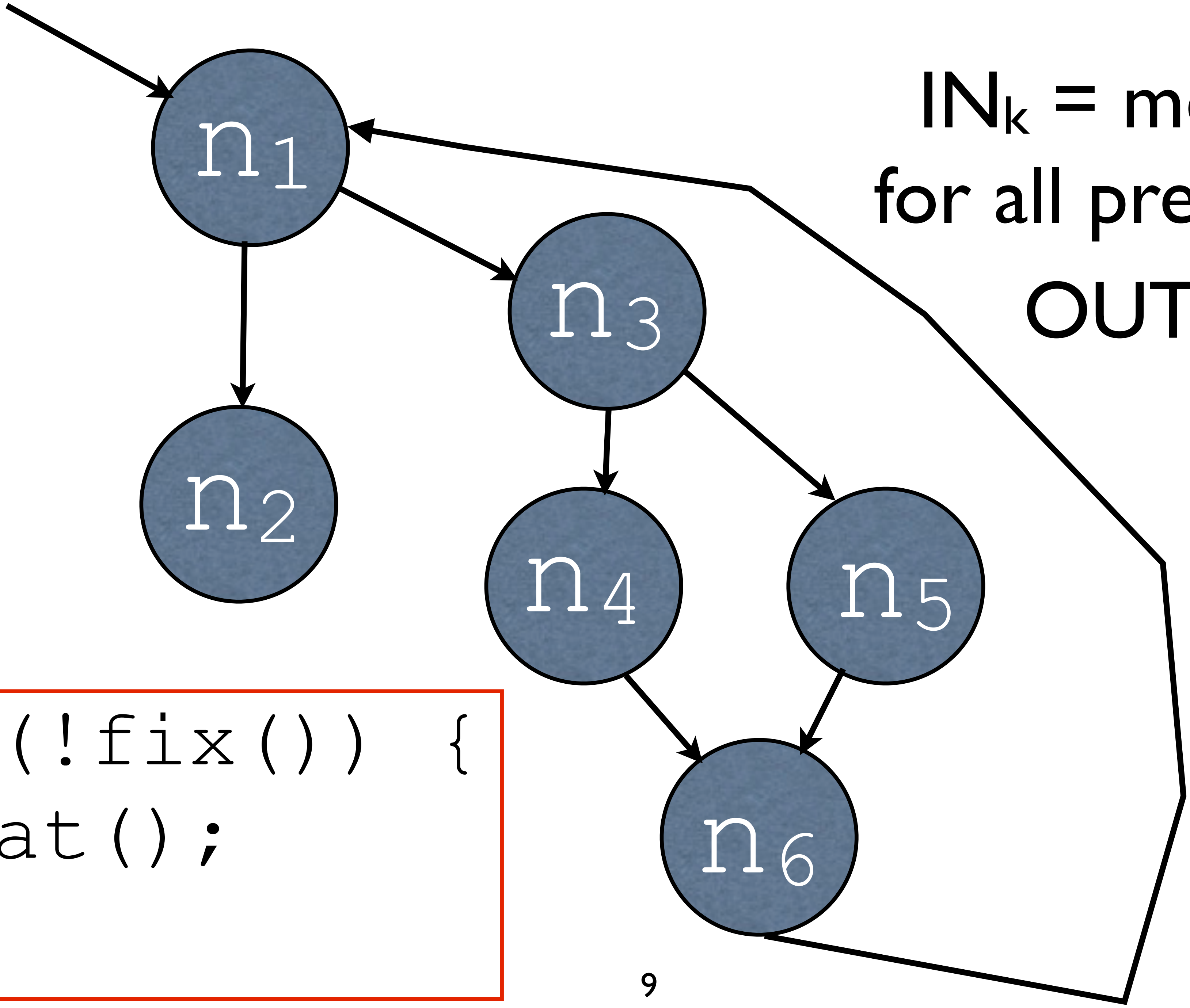
Traditional Dataflow Analysis



IN_k = meet(OUT_x)
for all predecessors x

$\boxed{OUT_k = Xfer(IN_k)}$

Traditional Dataflow Analysis



$\mathrm{IN}_k = \mathrm{meet}(\mathrm{OUT}_x)$
for all predecessors x

$\mathrm{OUT}_k = \mathrm{Xfer}(\mathrm{IN}_k)$

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

Dataflow Analysis Problem #1

- Underlying assumption: deriving the program's control flow graph is cheap
- This is not true for JavaScript
    - Higher-order functions
    - Exceptions
    - Implicit type conversions

Dataflow Analysis Problem #2

- Fundamentally, dataflow analysis' definition assumes sequential behavior
  - Each node acts as a synchronization point
  - Can end up calculating redundant info if nodes are processed in arbitrary order

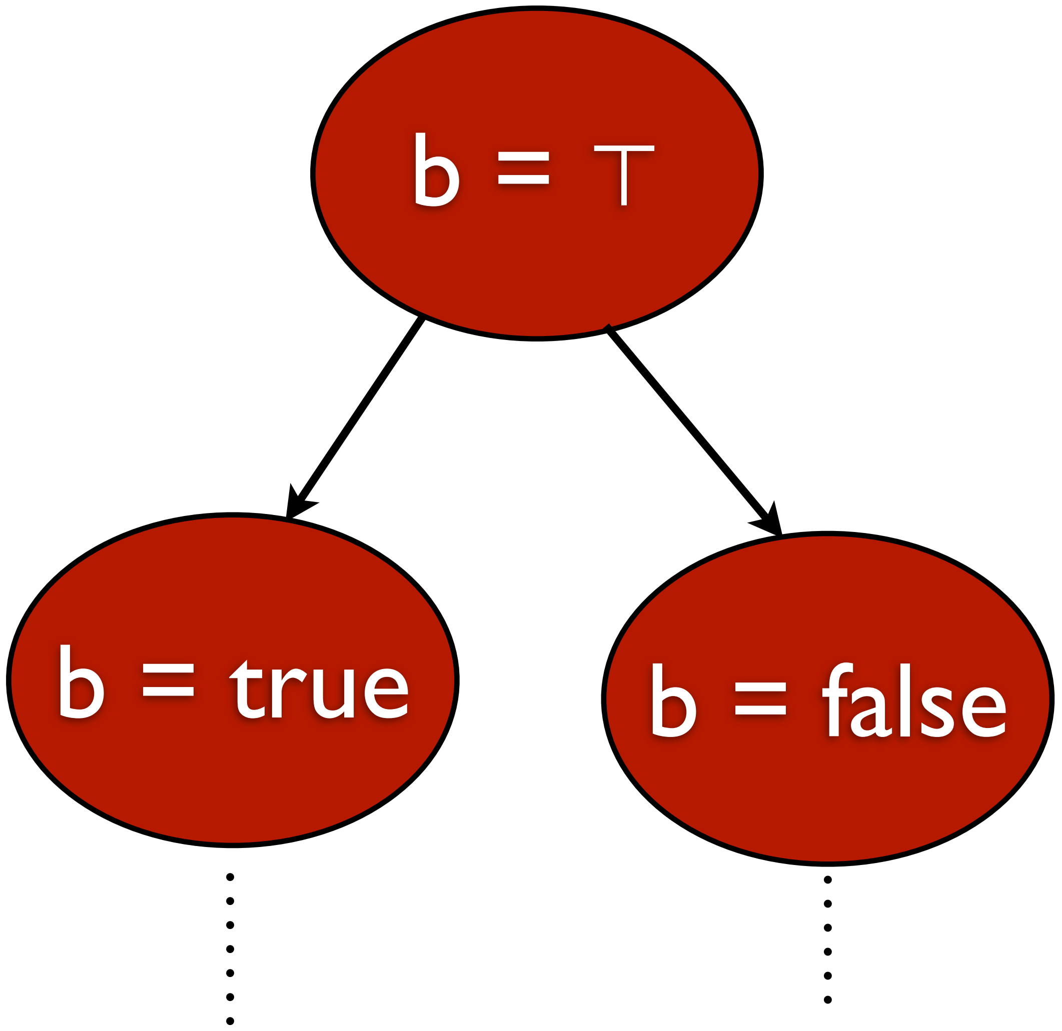
Our Approach

State Transition Representation

- Prior work: utilize abstract interpretation to form a *widened state transition system*
  - Represent program execution as a nondeterministic infinite state transition system
  - Analyze which states are reachable
  - Representable as a tree

State Transition Example

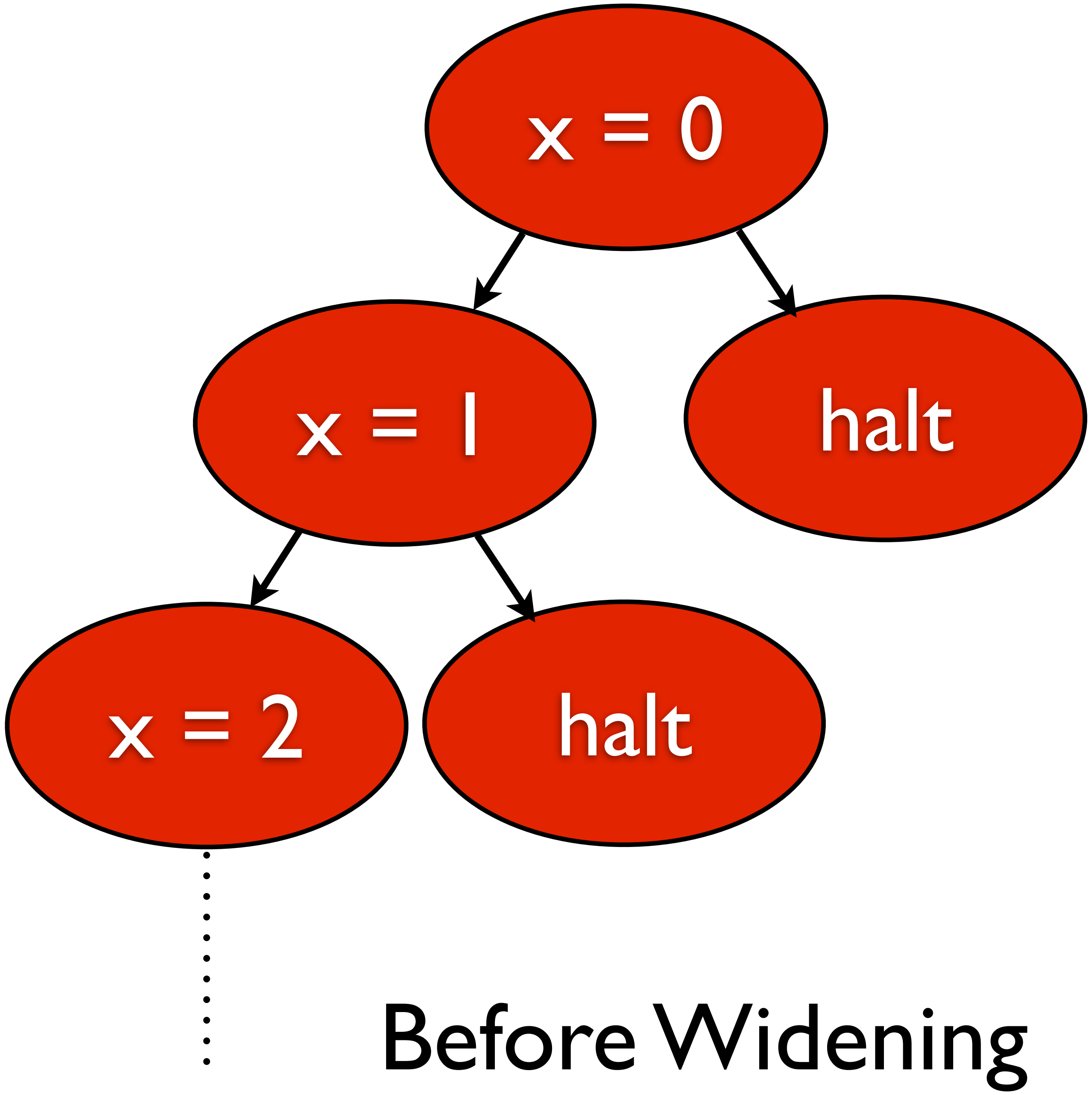
```
bool b = randBool();
if (b) {
   ...
} else {
   ...
}
```



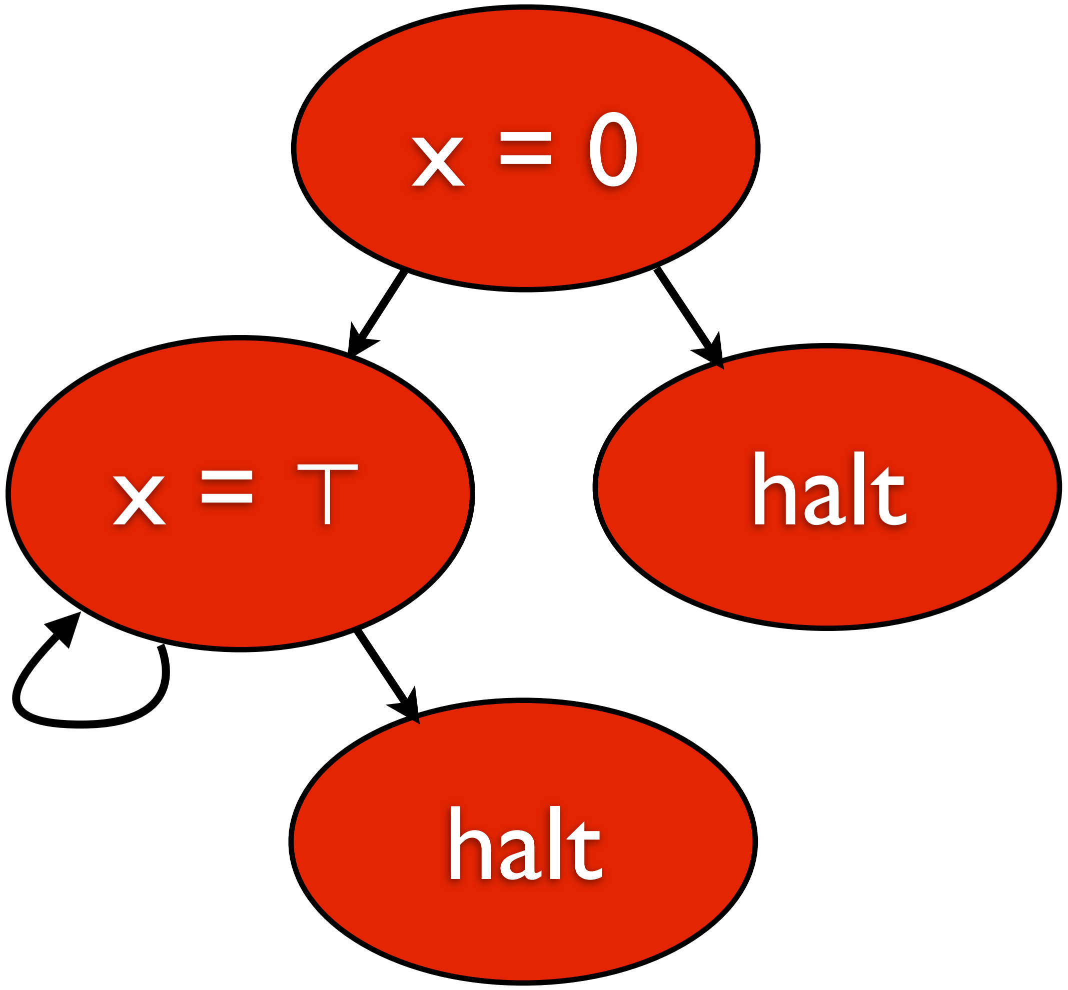
Computability and Tractability

- Trees can be of exponential or infinite size
  - Infinite loops can mean trees of infinite depth
- To ensure a reasonably finite tree size, a *widening operator* is employed
  - Allows for states to be selectively merged with each other

```
int x = 0;
while (randBool()) {
   x++;
}
```



Before Widening



After Widening

New Insight: This Parallelizes Well

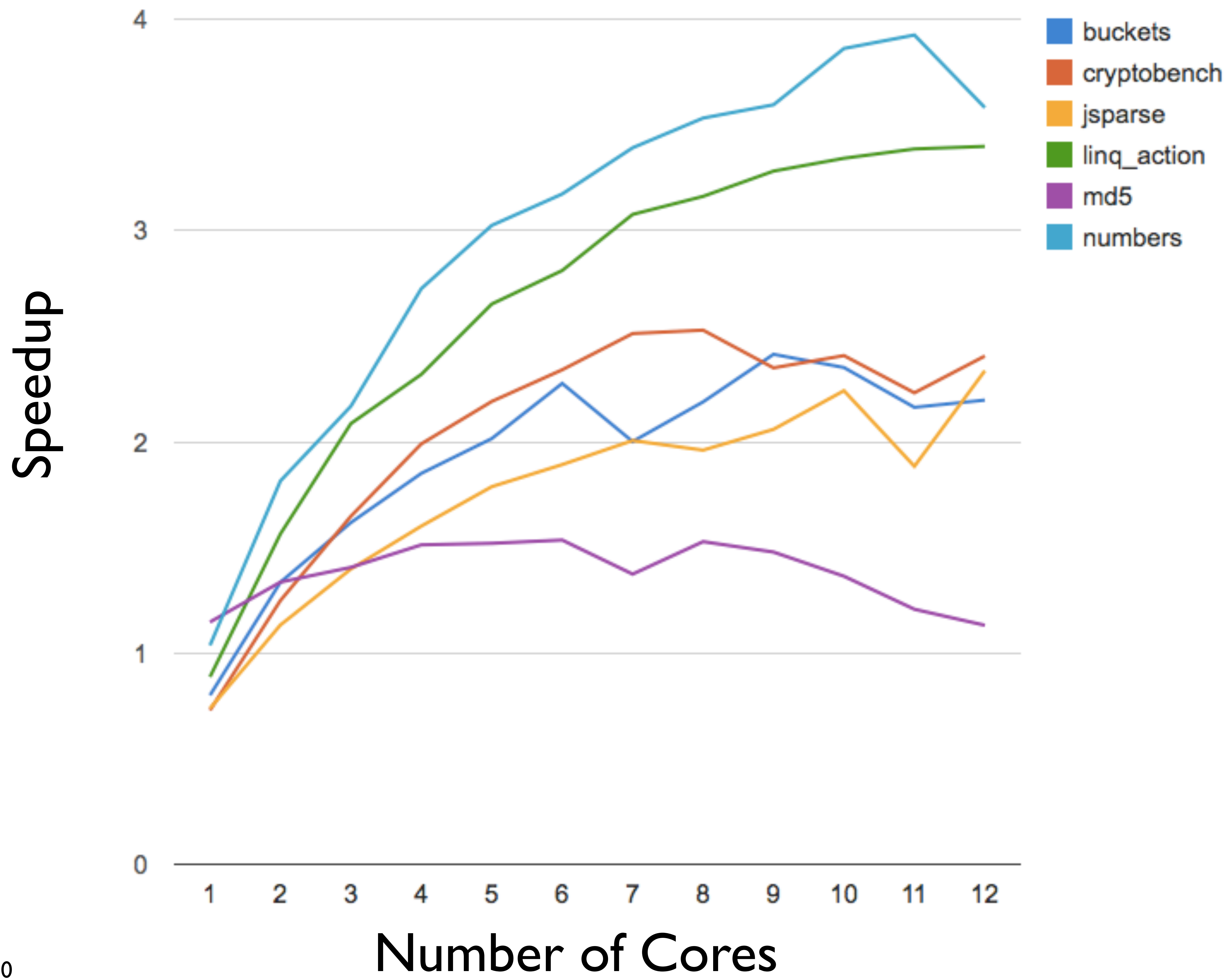
- The analysis and the widening component are separate
- The analysis is an inherently massively parallel tree-like state exploration
- The widening component selectively injects sequential dependencies

Analysis Parameters

Parameter	Our Instantiation for JavaScript
When are states merged? (existing)	Based on `k`-bounded call strings (CFA)
Where are threads placed? (new)	Each distinct context is assigned its own thread

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

Comparison to Related Work

- Most existing work deals with C
  - Our speedups are generally equal or better, despite additional JS complexity
- Most existing parallel frameworks are based on dataflow analysis
  - Ad-hoc
  - Requires control flow graphs
- Evaluation issues are common

Future Work

- Parallel experimentation with other merging strategies and thread granularity levels
  - Preliminary data shows there is progress that can be made
- Application to C and other languages
  - Would allow for direct comparison to related work