Data Representation Synthesis
PLDI'2011*, ESOP'12, PLDI'12*
CACM'12

Peter Hawkins, Stanford University
Alex Aiken, Stanford University
Kathleen Fisher, DARPA
Martin Rinard, MIT
Mooly Sagiv, TAU

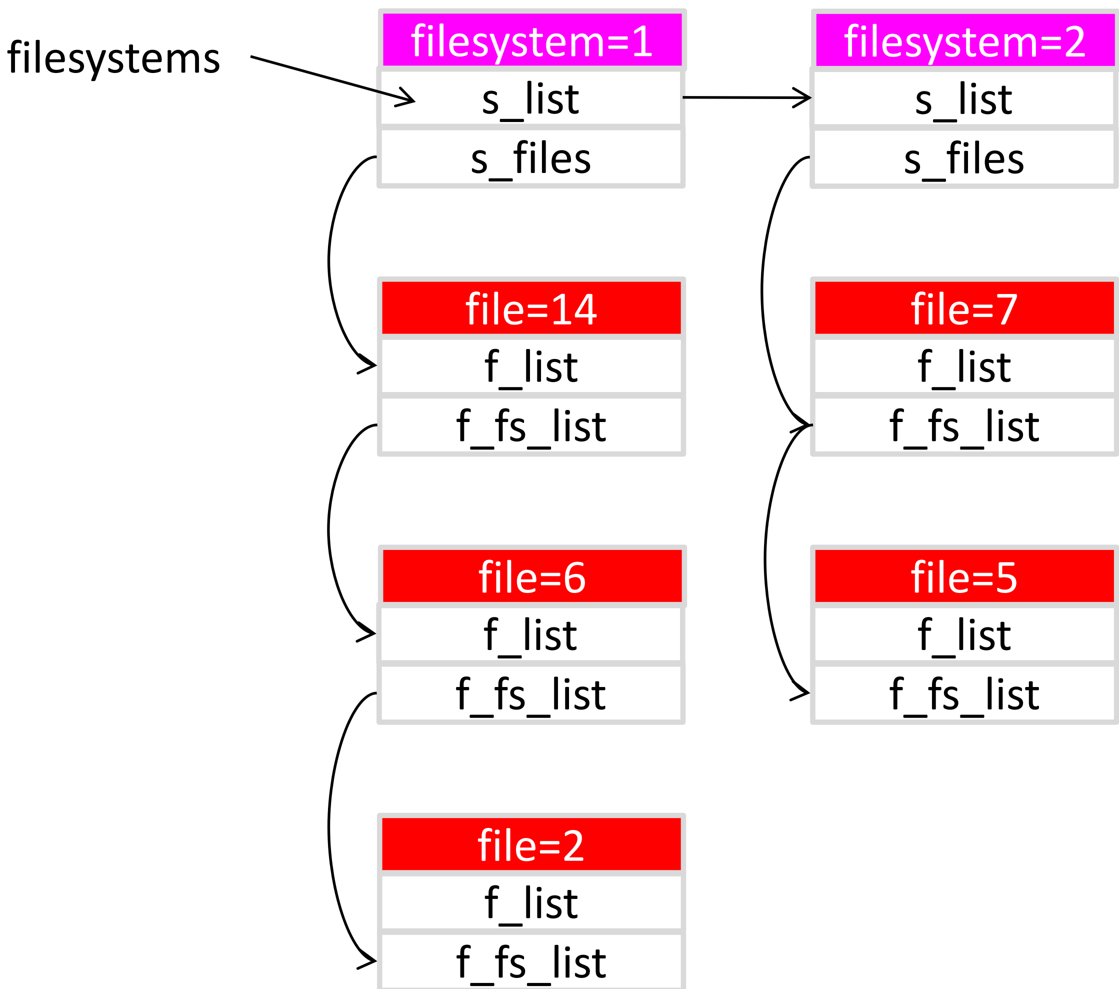
http://theory.stanford.edu/~hawkinsp/

* Best Paper Award

Background

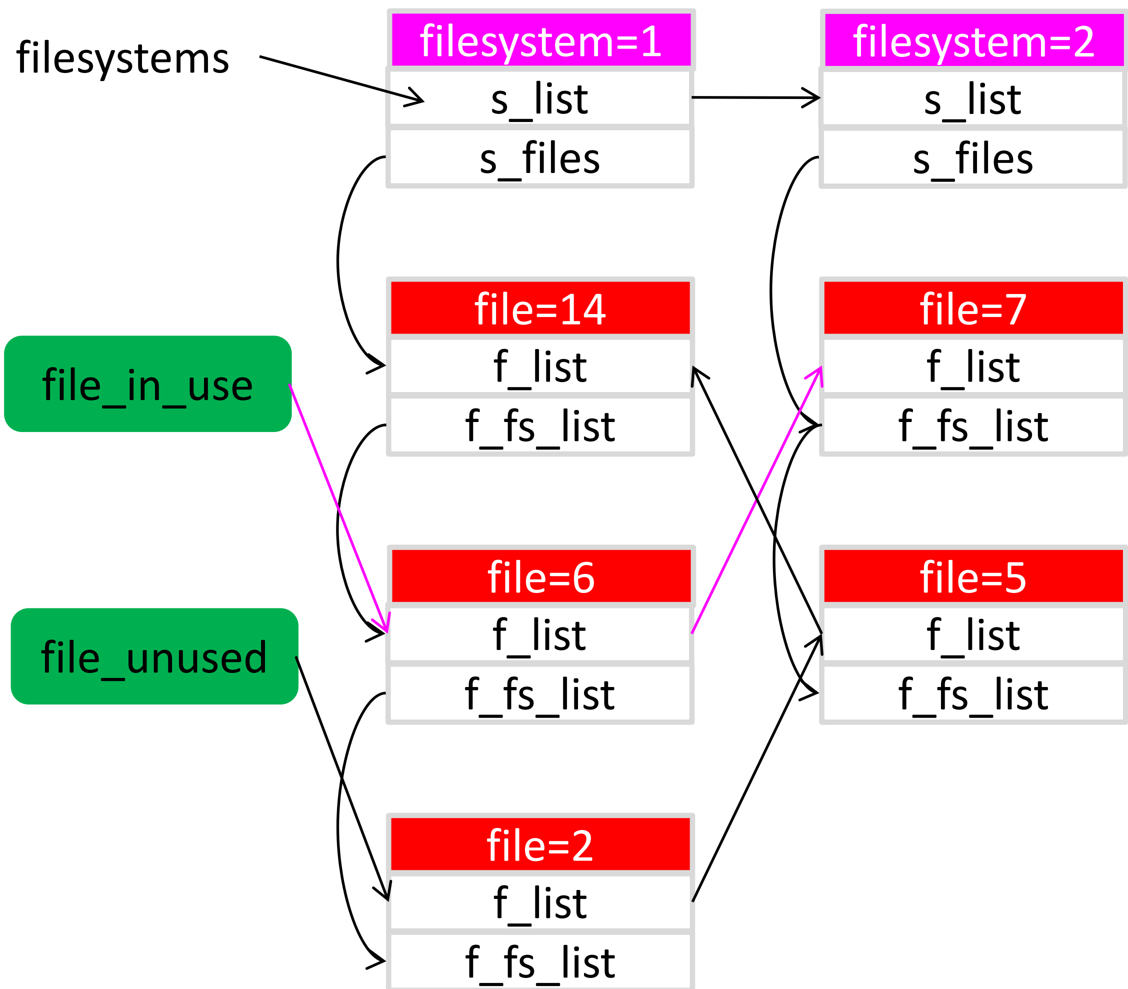
- High level formalisms for static program analysis
  - Circular attribute grammars
  - Horn clauses
- Interprocedural Analysis
  - Context free reachability
    - Implemented in SLAM/SDV
- Shape Analysis
  - Low level pointer data structures

Composing Data Structures



Problem: Multiple Indexes

+Concurency



Access Patterns

- Find all mounted filesystems
- Find cached files on each filesystem
- Iterate over all used or unused cached files in Least-Recently-Used order

Disadvantages of linked shared data structures

- Error prone
- Hard to change
- Performance may depend on the machine and workload
- Hard to reason about correctness

- Concurrency makes it harder
  - Lock granularity
  - Aliasing

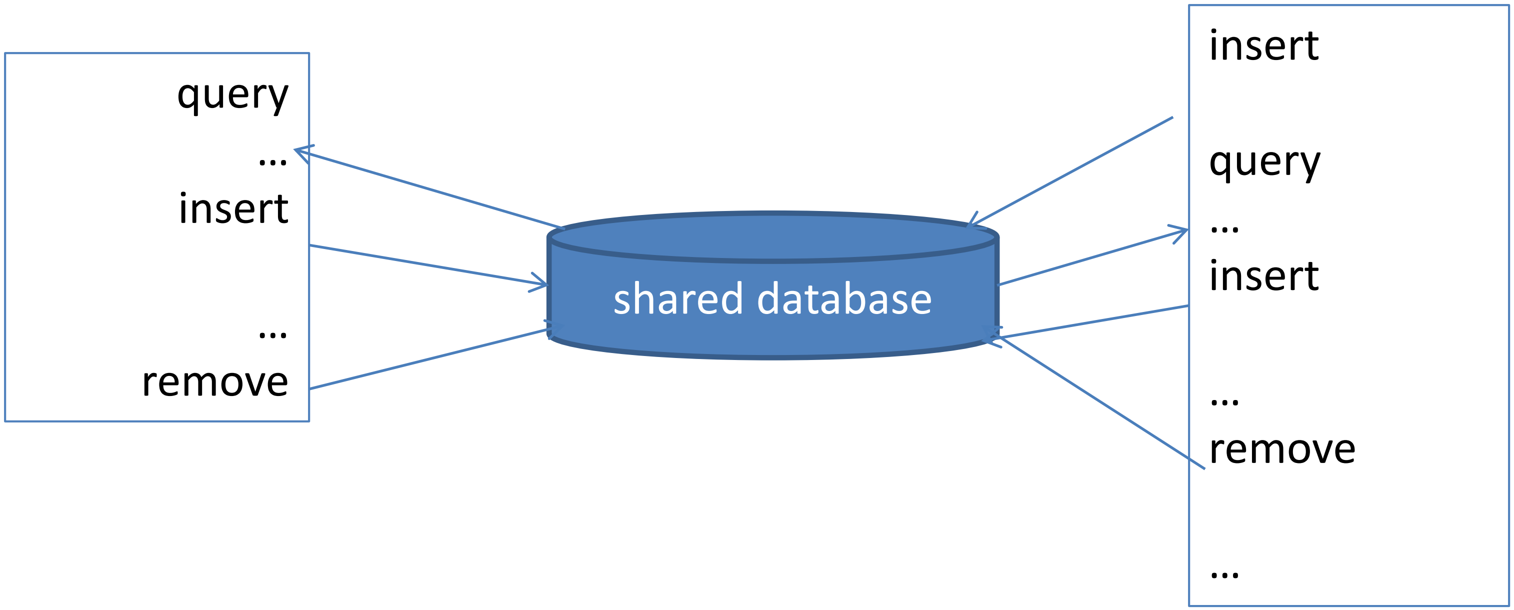
Our thesis

- Very high level programs
  - No pointers and shared data structures
  - Easier programming
  - Simpler reasoning
  - Machine independent
- The compiler generates pointers and multiple concurrent shared data structures
- Performance comparable to manually written code

Our Approach

- Program with “database”
  - States are tables
  - Uniform relational operations
    - Hide data structures from the program
  - Functional dependencies express program invariants
- The compiler generates low level shared pointer data structures with concurrent operations
  - Correct by construction
- The programmer can tune efficiency
- Autotuning for a given workload

Conceptual Programming Model



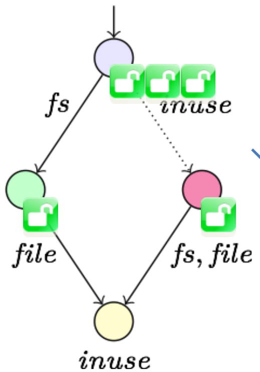
Relational Specification

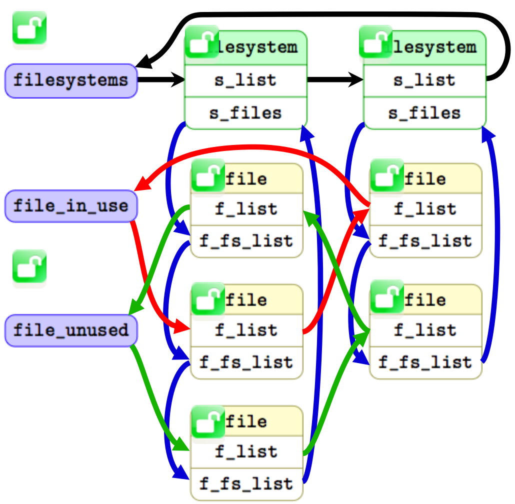
- Program states as relations
  - Columns correspond to properties
  - Functional dependencies define global invariants

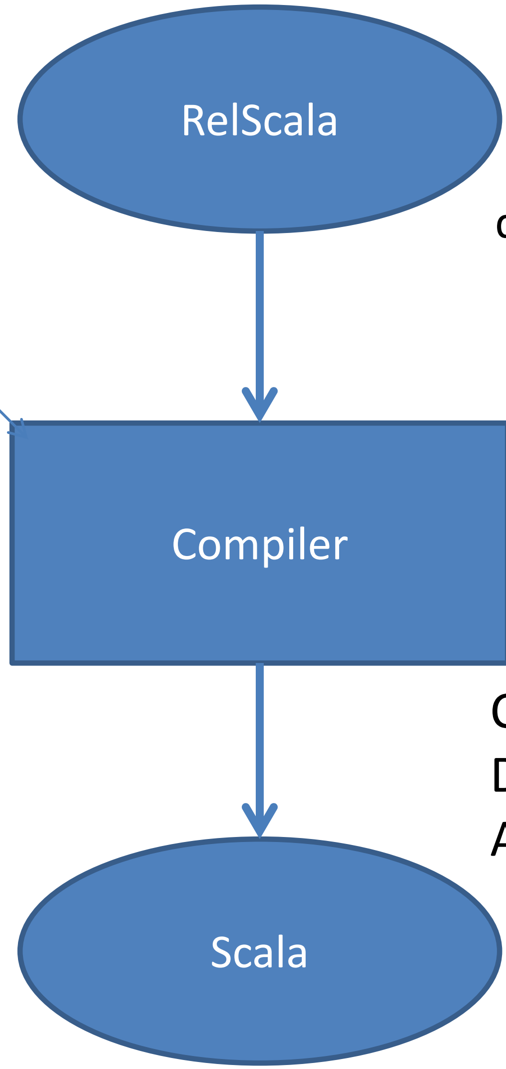
Atomic Operation	meaning
r= empty	r := {}
insert r s t	if s $\notin$ r then r = r $\cup$ {<s.t>}
query r S C	The C of all the tuples in r matching tuple
remove r s	remove from r all the tuples which match s

The High Level Idea

Decomposition







$$\{fs, file, inuse\}$$

$$fs, file \rightarrow inuse$$

query <inuse:T> {fs, file}

Concurrent Compositions of
Data Structures,
Atomic Transactions

List * query(FS* fs, File* file) {
lock(fs) ; for (q= file_in_use; …)
….

Filesystem

- Three columns {fs, file, inuse}
- fs:int × file:int × inuse:Bool
- Functional dependencies
  - {fs, file} → { inuse}

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F
1	2	T

Filesystem (operations)

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

query <inuse:T> {fs, file }=

[<fs:2, file:7>, <fs:1, file:6>]

Filesystem (operations)

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

insert <fs:1, file:15> <inuse:T>

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F
1	15	T

Filesystem (operations)

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F
1	15	T

remove <fs:1>

fs	file	inuse
2	7	T
2	5	F

Directed Graph Data Structure

- Three columns {src, dst, weight}
- src $\times$ dst $\times$ weight
- Functional dependencies
  - {src, dst} $\rightarrow$ { weight}
- Operations
  - query <src:1> {dst, weight}
  - query <dst:5> {src, weight}

Plan

- Compiling into sequential code (PLDI’11)
- Adding concurrency (PLDI’12)

Mapping Relations into Low Level Data Structures

- Many mappings exist
- How to combine several existing data structures
  - Support sharing
- Maintain the relational abstraction
- Reasonable performance
- Parametric mappings of relations into shared combination of data structures
  - Guaranteed correctness

The RelC Compiler

Relational Specification

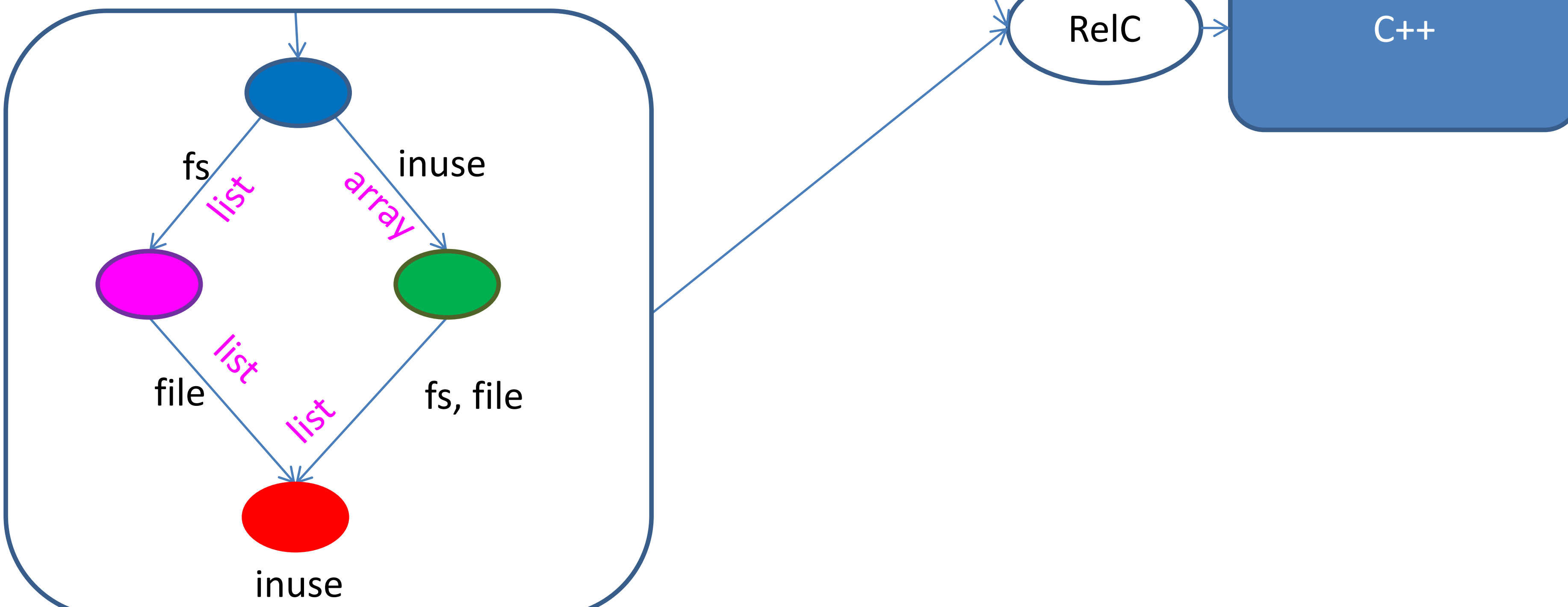
$fs \times file \times inuse$

$\{fs, file\} \to \{inuse\}$

foreach <fs, file, inuse>$\in$ filesystems s.t. fs= 5

do …

Graph decomposition



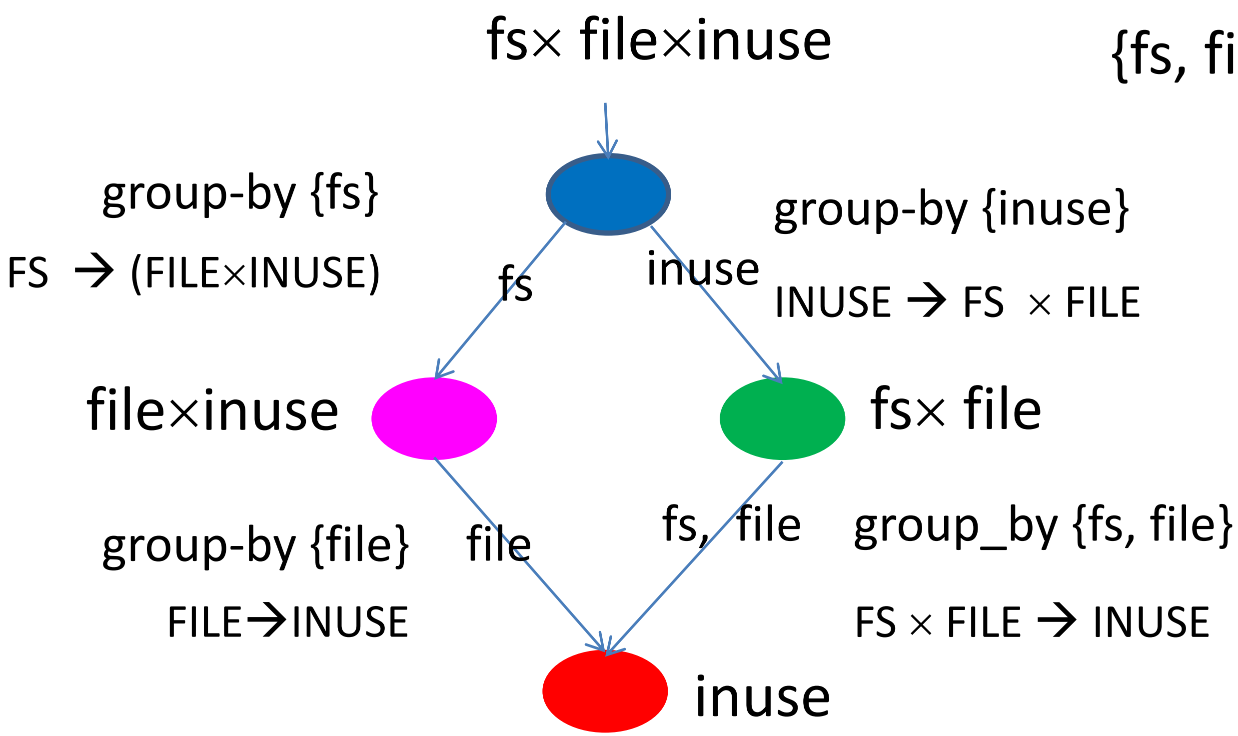
Decomposing Relations

- Represents subrelations using container data structures
- A directed acyclic graph(DAG)
  - Each node is a sub-relation
  - The root represents the whole relation
  - Edges map columns into the remaining sub-relations
  - Shared node=shared representation

Decomposing Relations into Functions
Currying

$fs \times file \times inuse$

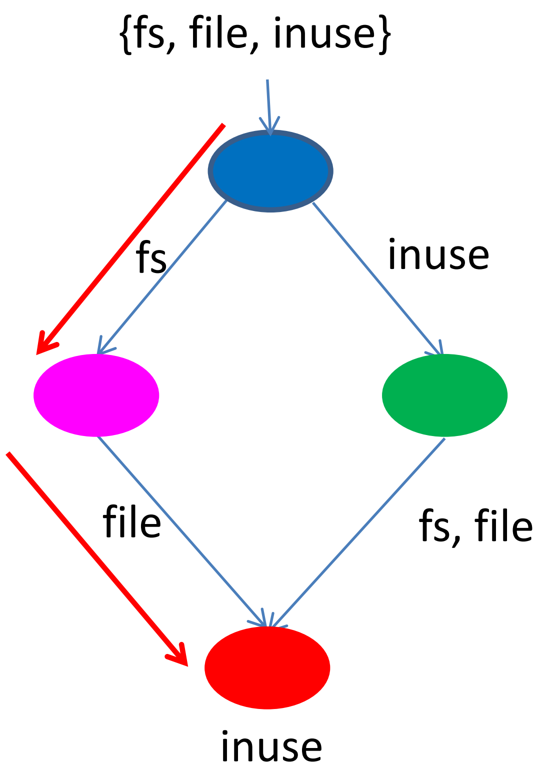
$\{fs, file\} \rightarrow \{inuse\}$



FS → (FILE→INUSE)

INUSE → (FS × FILE → INUSE)

Filesystem Example



fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

fs:1

file	inuse
14	F
6	T
2	F

fs:2

file	inuse
7	T
5	F

file:14

inuse
F

file:6

inuse
T

file:2

inuse
F

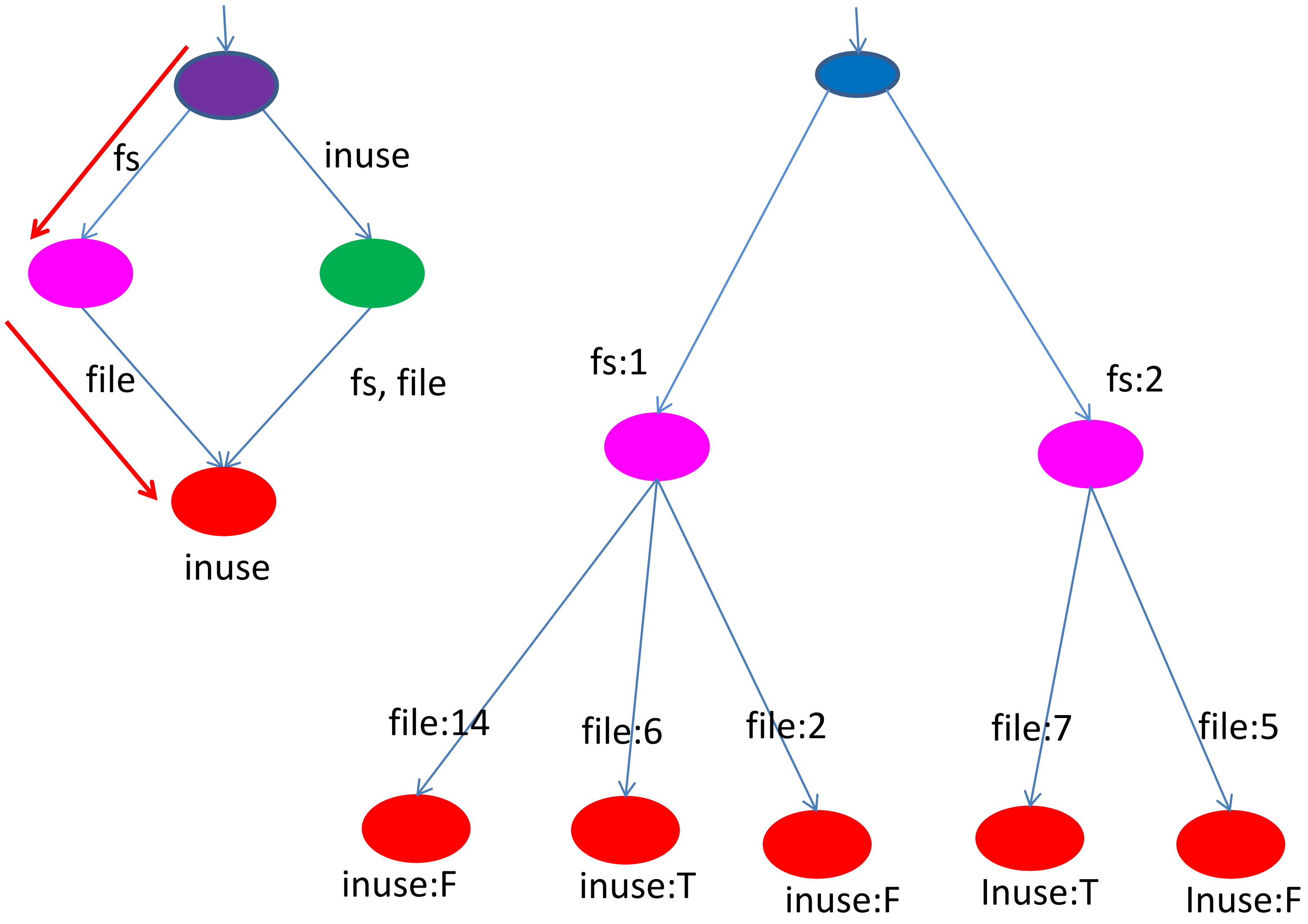
file:7

inuse
T

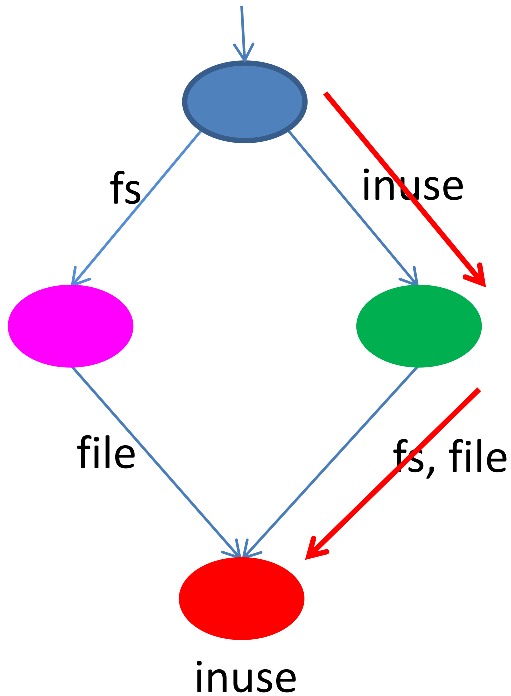
file:5

inuse
F

Memory Decomposition(Left)



Filesystem Example



$\{fs, file\} \rightarrow \{ inuse\}$

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

inuse:T

fs	file
2	7
1	6

inuse:F

fs	file
1	14
2	5
1	2

fs:2
file:7

inuse
T

fs:1
file:6

inuse
T

fs:1
file:14

inuse
F

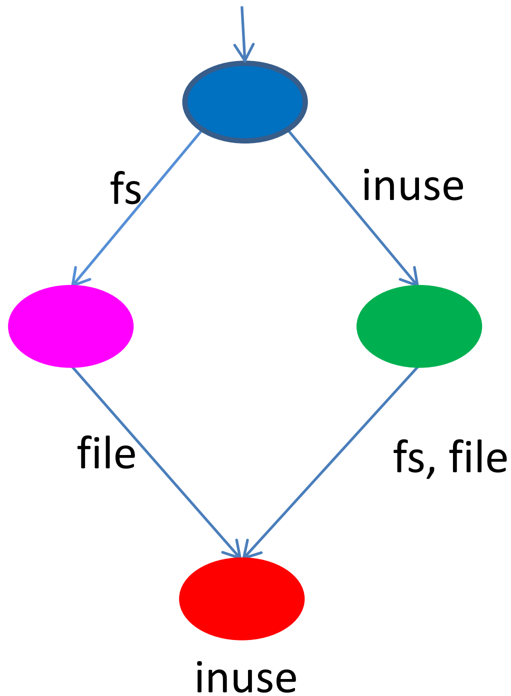
fs:2
file:5

inuse
F

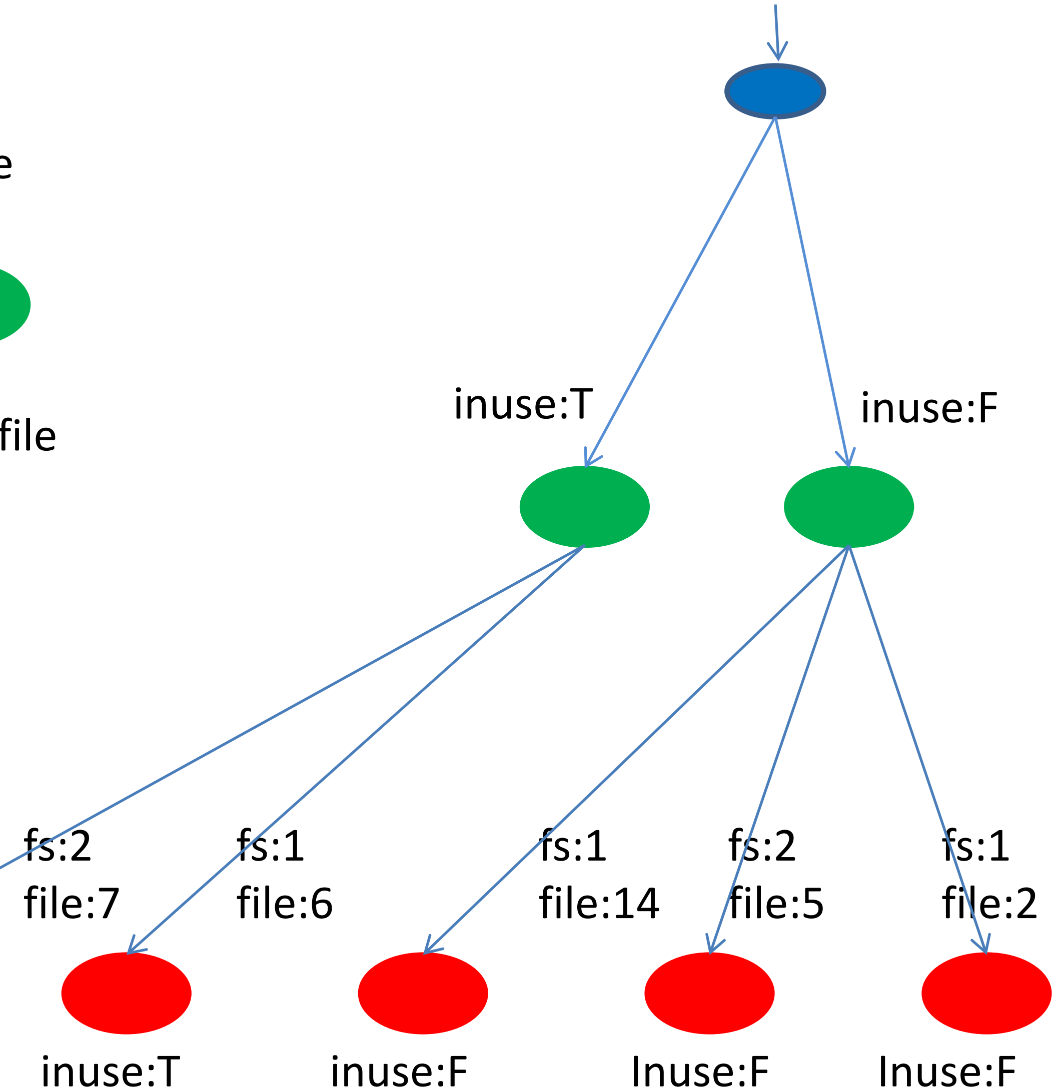
fs:1
file:2

inuse
F

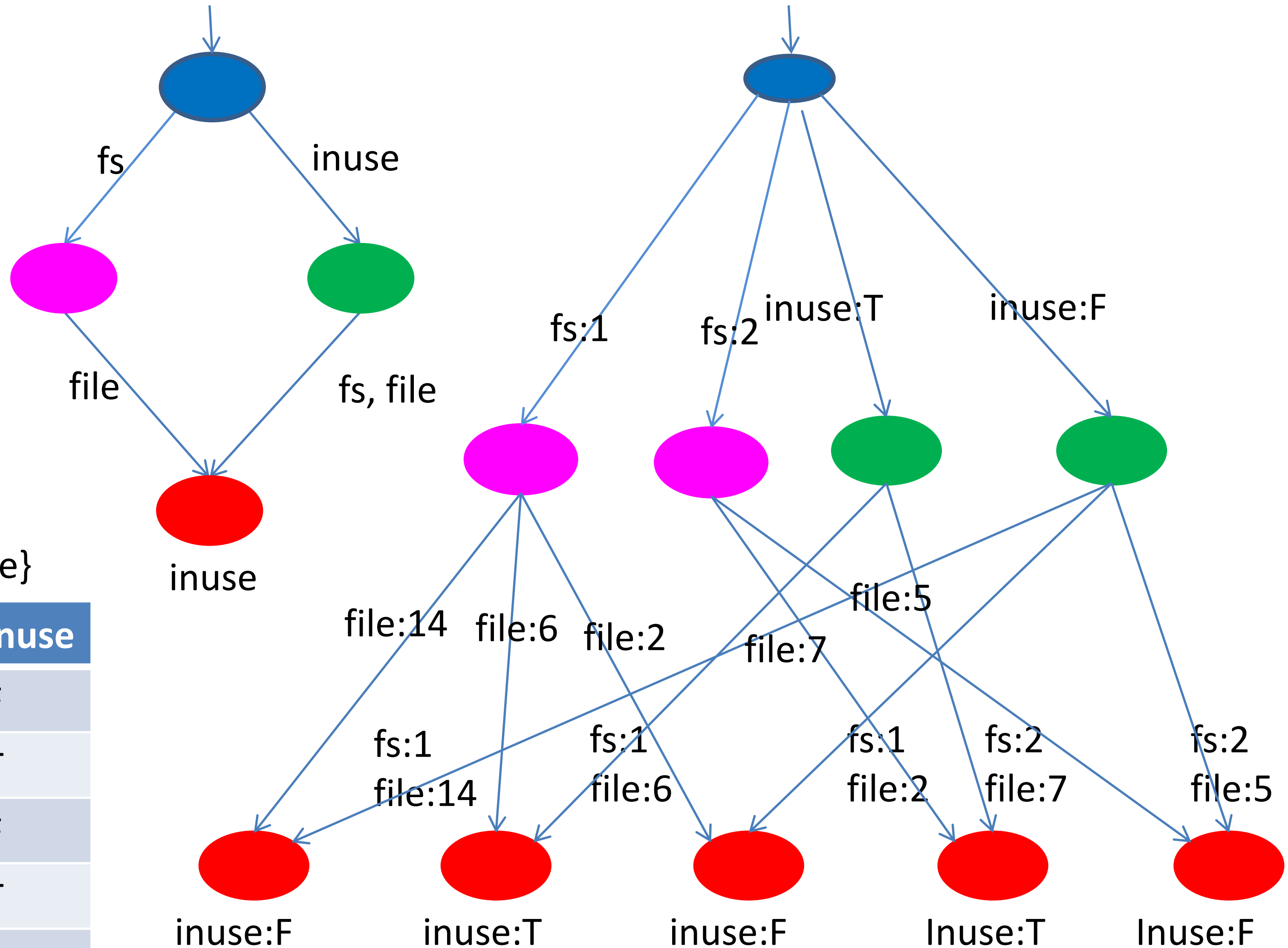
Memory Decomposition(Right)



{fs, file} $\rightarrow$ { inuse}



Decomposition Instance

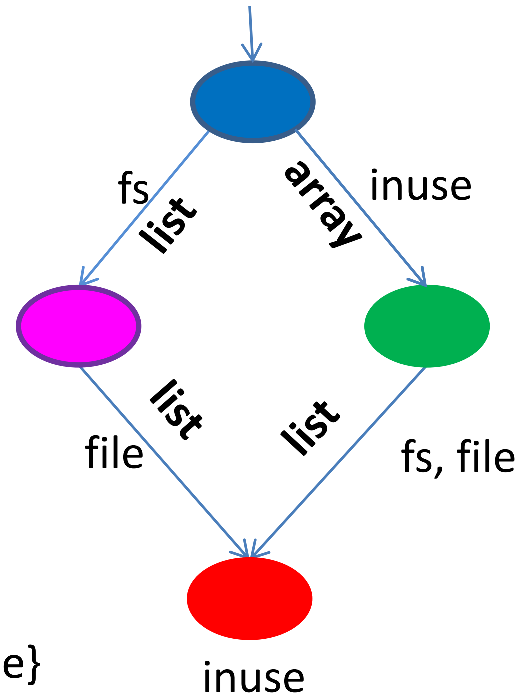


fs $\times$ file $\times$ inuse

{fs, file} $\rightarrow$ { inuse}

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

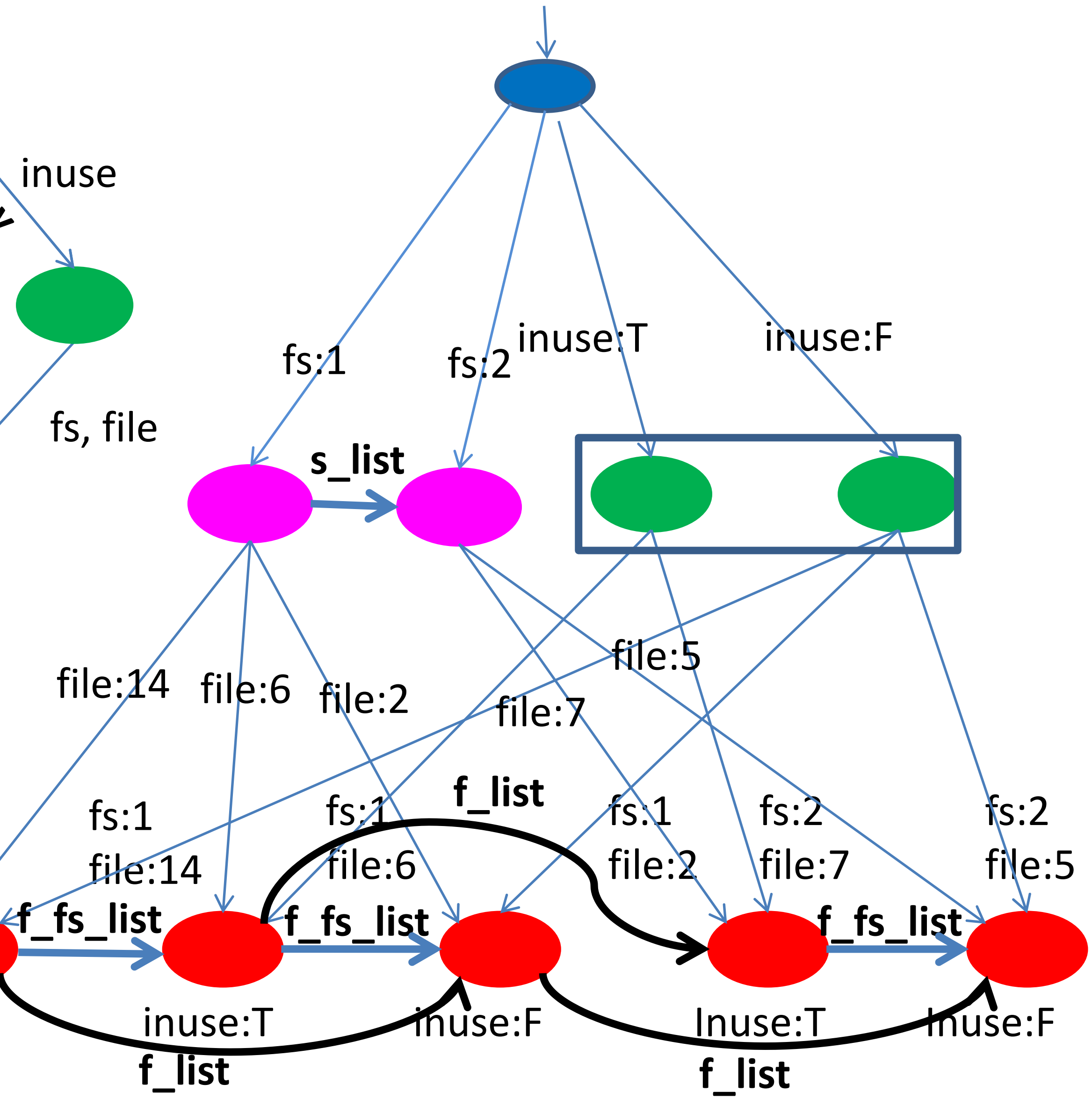
Decomposition Instance



fs × file × inuse

{fs, file} →{ inuse}

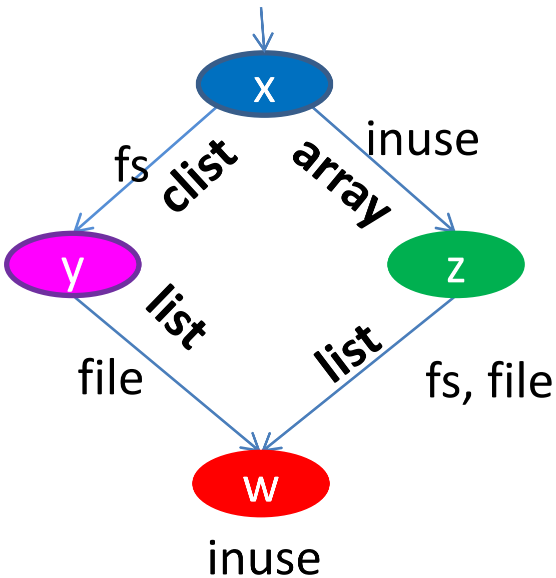
fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F



Decomposing Relations Formally(PLDI'11)

fs$\times$ file$\times$inuse

$\{fs, file\} \rightarrow \{inuse\}$



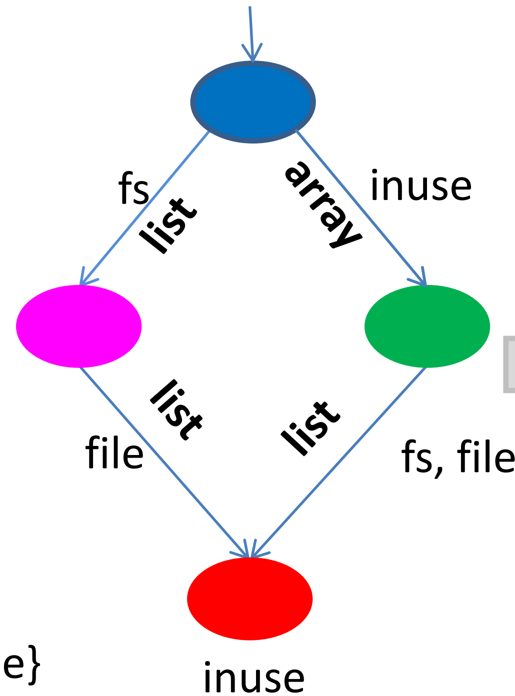
let w: {fs, file,inuse} $\triangleright$ {inuse} = {inuse} in

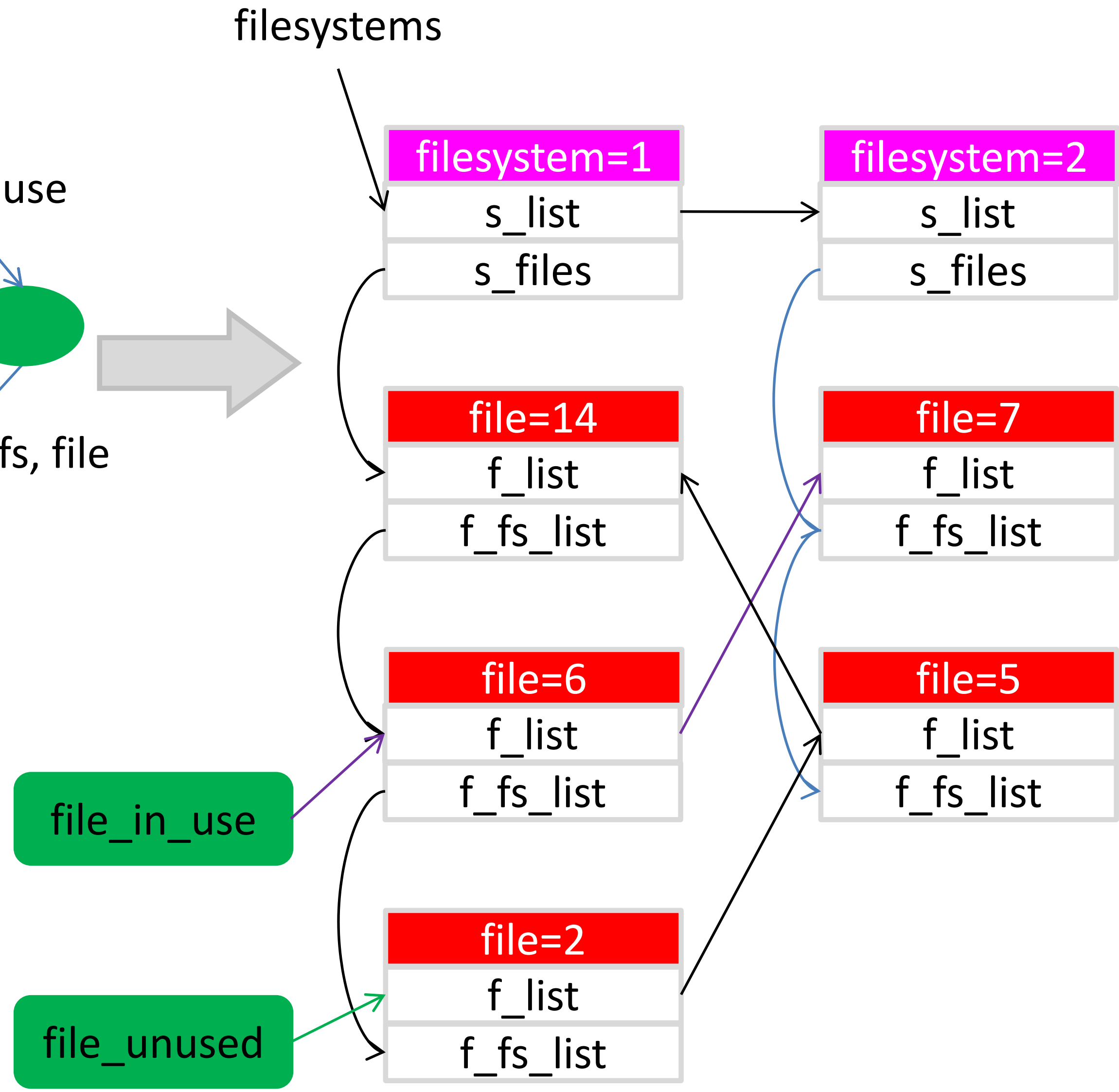
let y : {fs} $\triangleright$ {file, inuse} = {file} $\rightarrow^{list}$ {w} in

let z : {inuse } $\triangleright$ {fs, file, inuse} = {fs,file} $\rightarrow^{list}$ {w} in

let x: {} $\triangleright$ {fs, file, inuse} = {fs} $\rightarrow^{clist}$ {y} $\bowtie$ {inuse} $\rightarrow^{array}$ {z}

Memory State





fs × file × inuse

{fs, file} →{ inuse}

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

Adequacy

Not every decomposition is a good representation of a relation

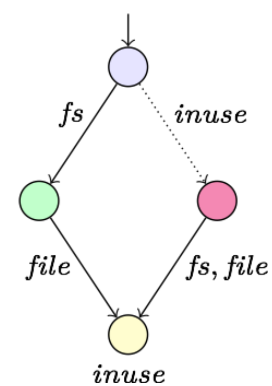
A decomposition is *adequate* if it can represent every possible relation matching a relational specification

enforces sufficient conditions for adequacy

$$\{fs, file, inuse\}$$

$$fs, file \rightarrow inuse$$

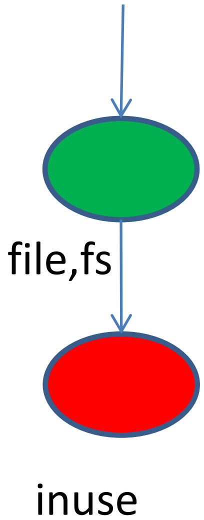
Adequacy



Adequacy of Decompositions

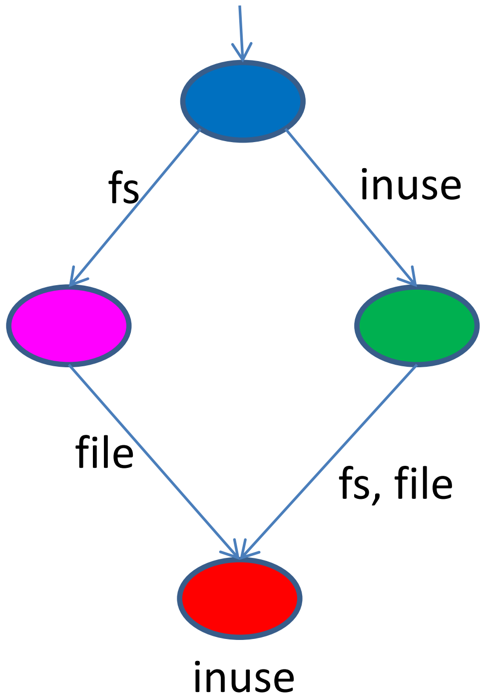
- All columns are represented
- Nodes are consistent with functional dependencies
  - Columns bound to paths leading to a common node must functionally determine each other

Respect Functional Dependencies



✓ $\{\text{file, fs}\} \rightarrow \{\text{inuse}\}$

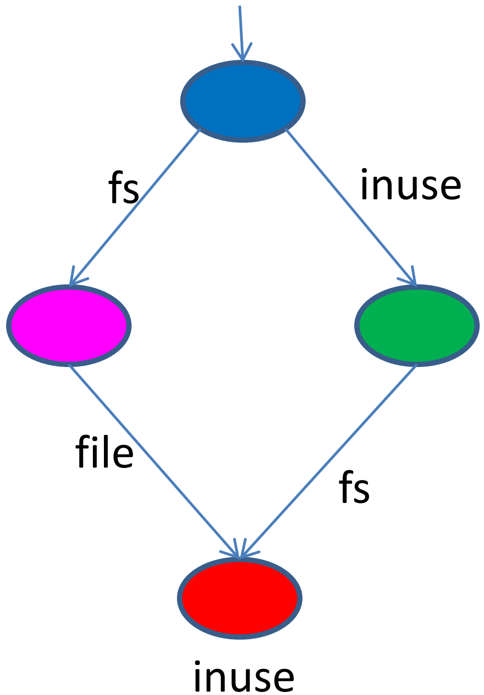
Adequacy and Sharing



Columns bound on a path to an object *x must functionally* determine columns bound on any other path to *x*

✓ {fs, file}↔{inuse, fs, file}

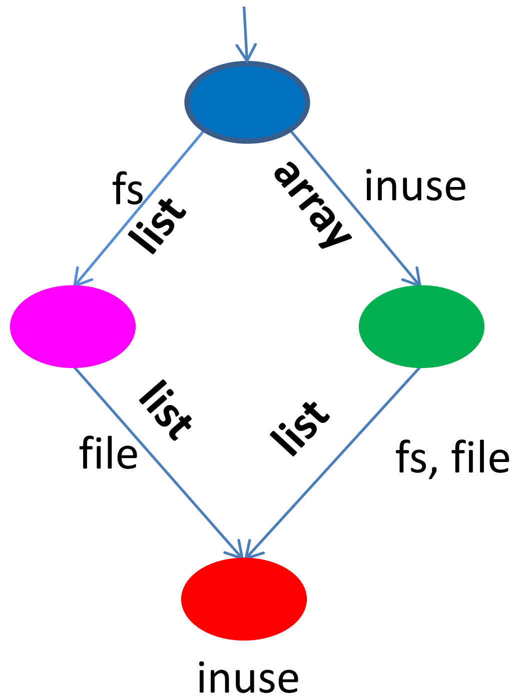
Adequacy and Sharing

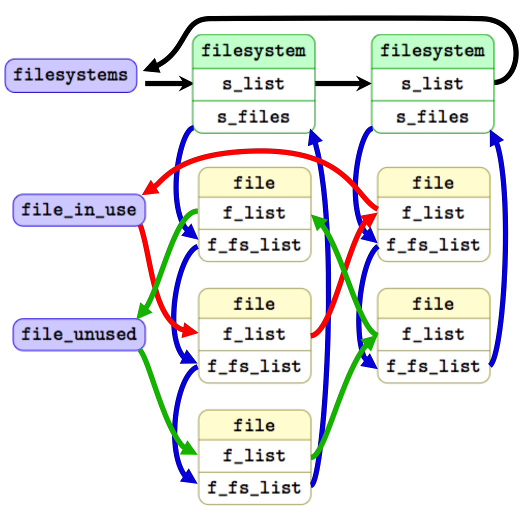


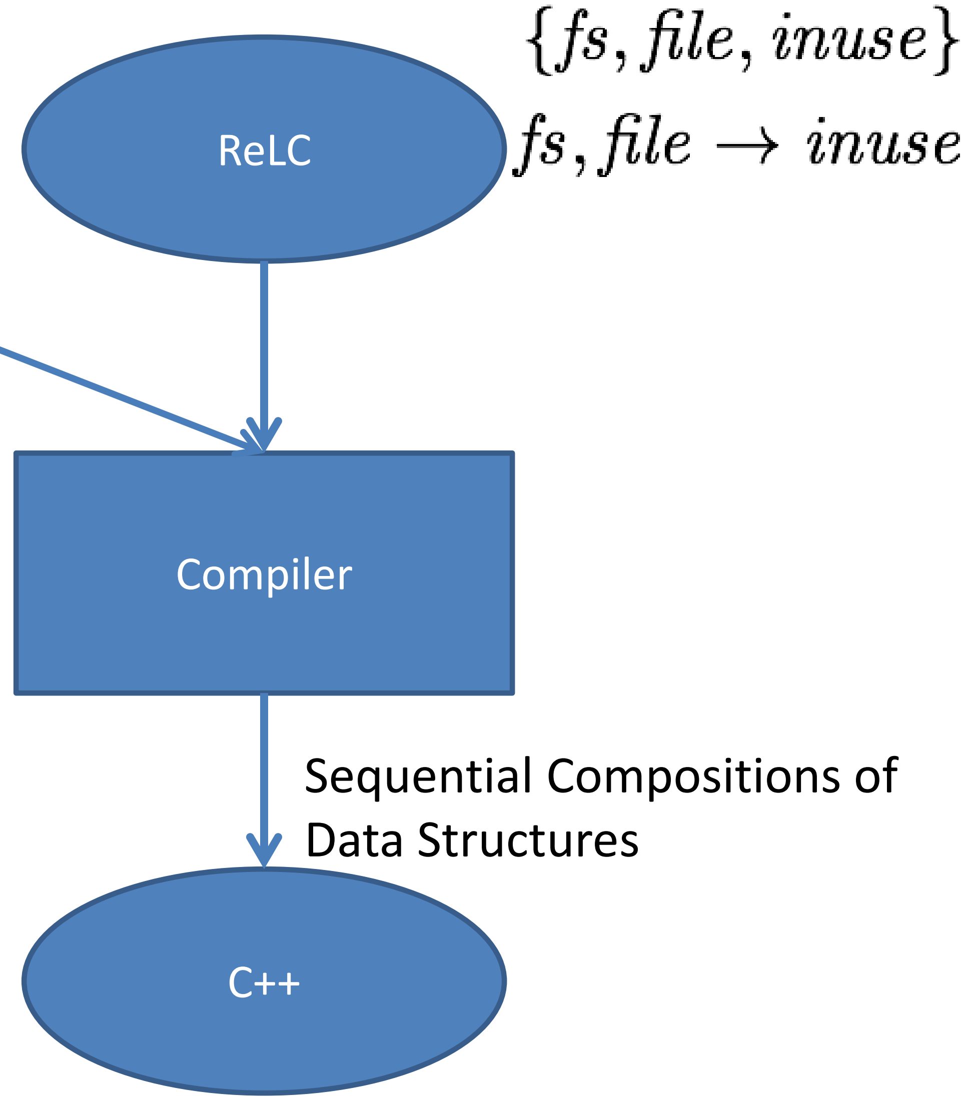
Columns bound on a path to an object *x must functionally* determine columns bound on any other path to *x*

☒ {fs, file} ↮ {inuse, fs}

The RelC Compiler PLDI'11

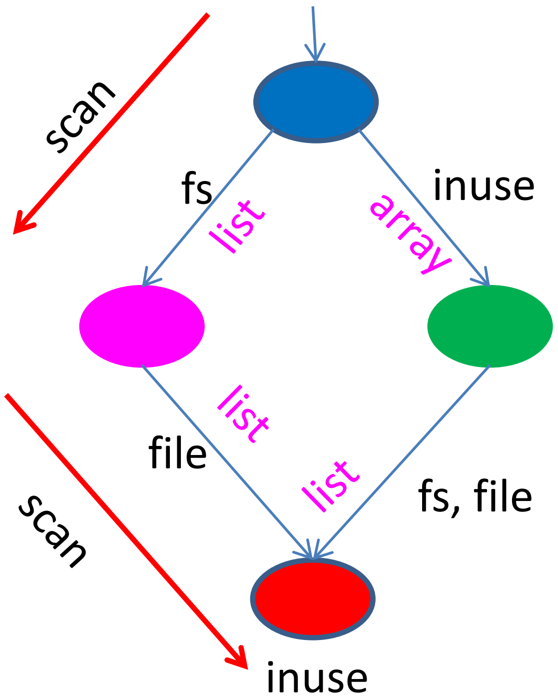






Query Plans

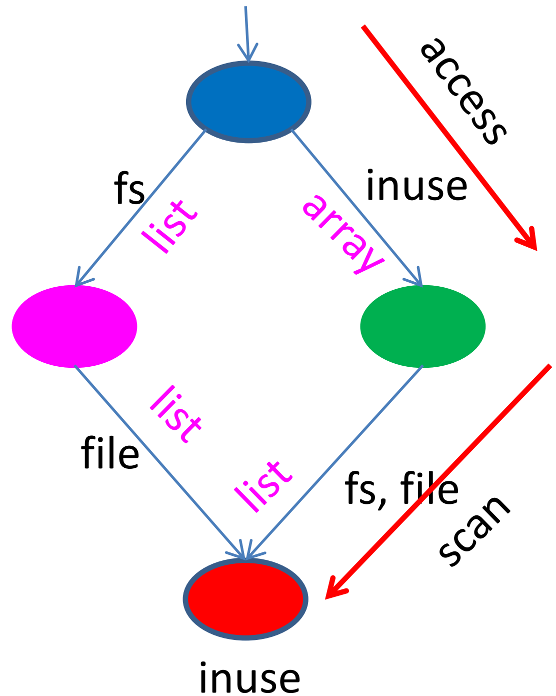
foreach <fs, file, inuse>$\in$ filesystems
  if inuse=T do …



Cost proportional to the number of files

Query Plans

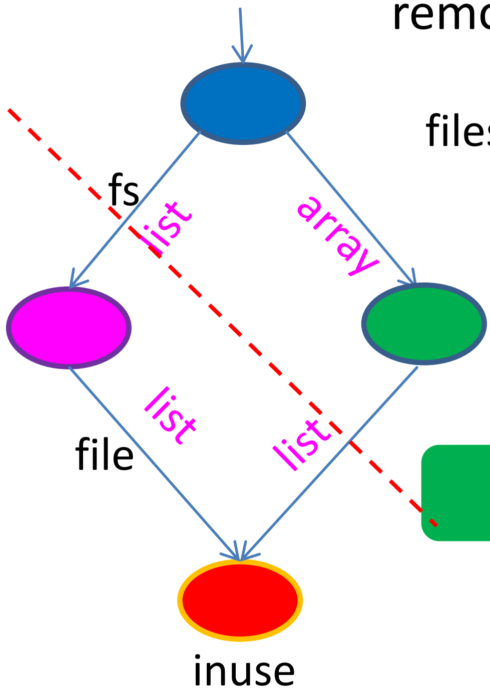
foreach <fs, file, inuse>∈ filesystems
  if inuse=T do …

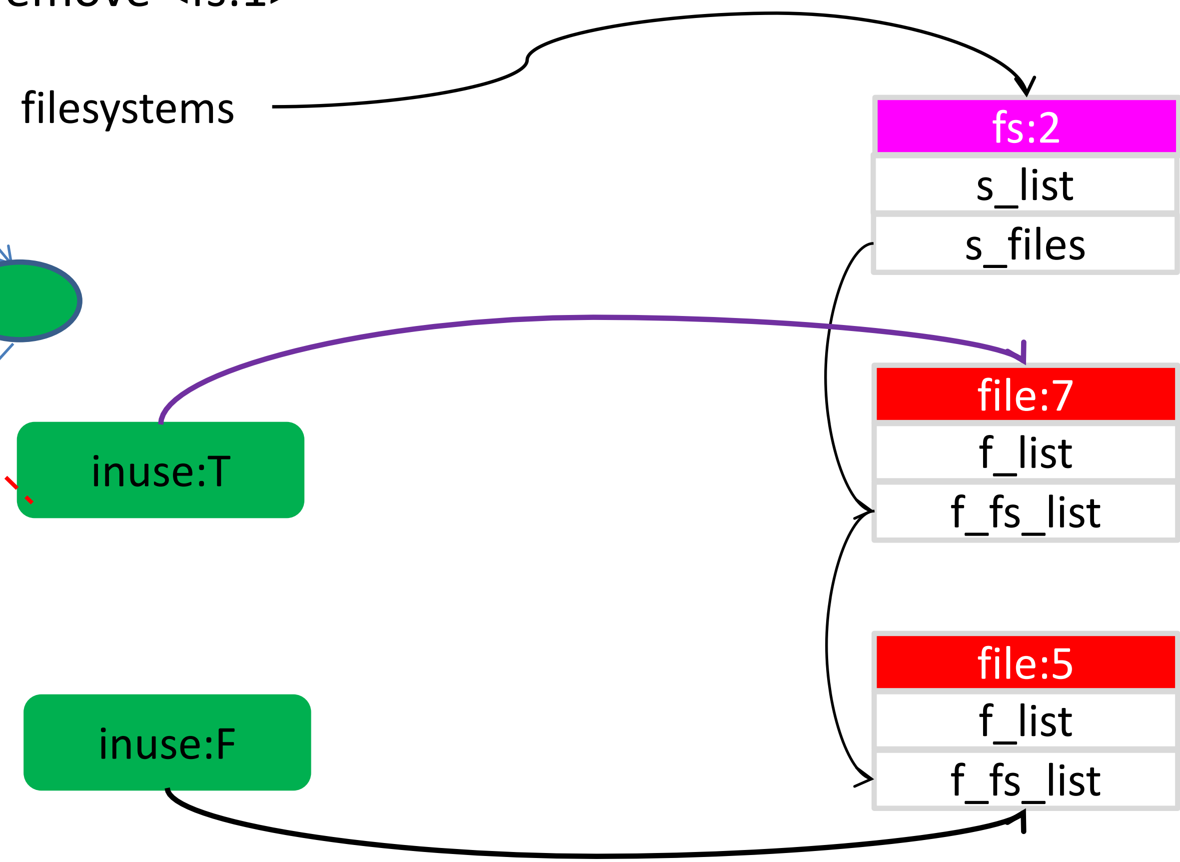


Cost proportional to the number of files in use

Removal and graph cuts

remove <fs:1>

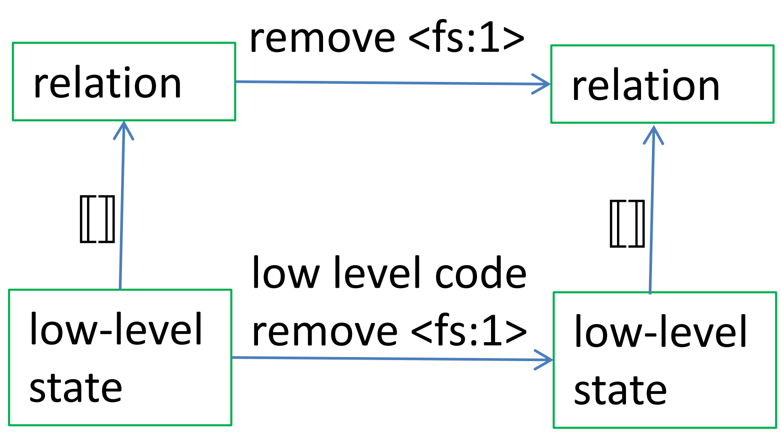




fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

Abstraction Theorem

- If the programmer obeys the relational specification and the decomposition is **adequate** and if the individual containers are correct
- Then the generated low-level code maintains the relational abstraction

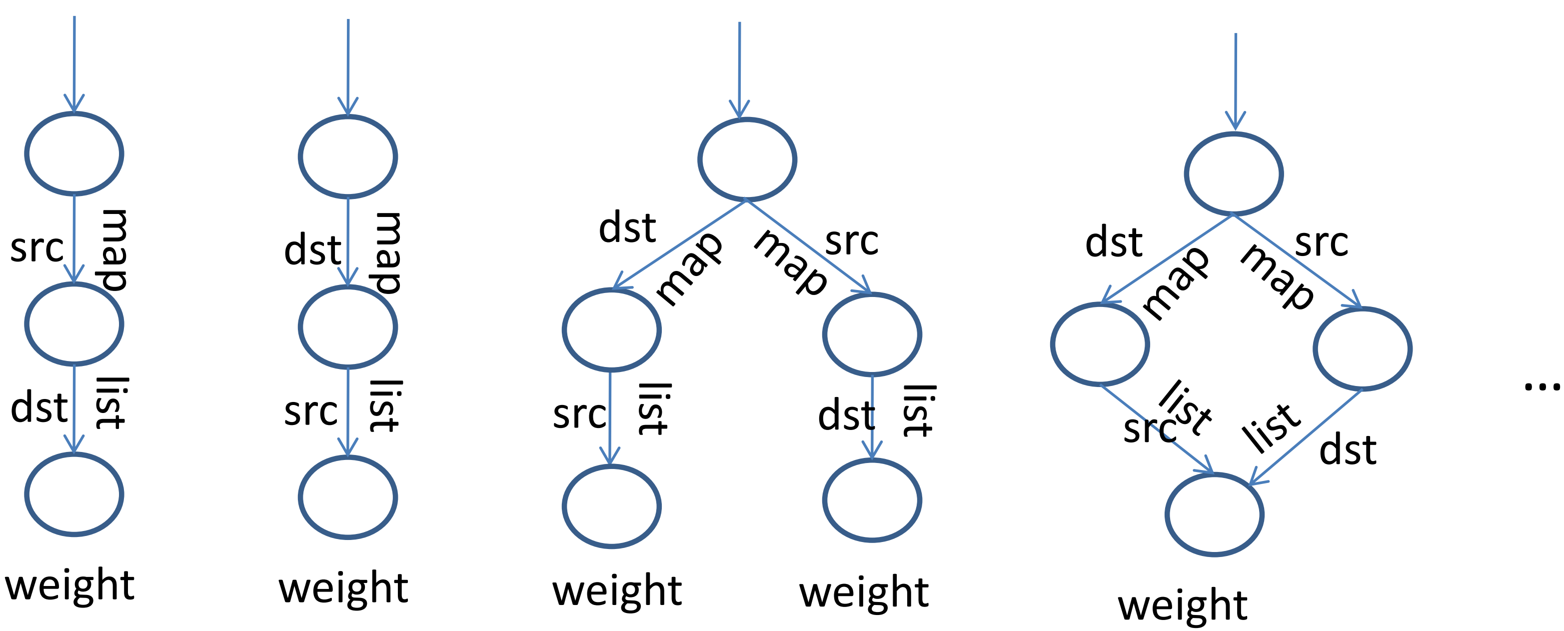


Autotuner

- Given a fixed set of primitive types
  - list, circular list, doubly-linked list, array, map, …
- A workload
- Exhaustively enumerate all the adequate decompositions up to certain size
- The compiler can automatically pick the best performing representation for the workload

Directed Graph Example (DFS)

- Columns
  src × dst × weight
- Functional Dependencies
  - {src, dst} → {weight}
- Primitive data types
  - map, list



Synthesizing Concurrent Programs

PLDI'12

Multiple ADTs

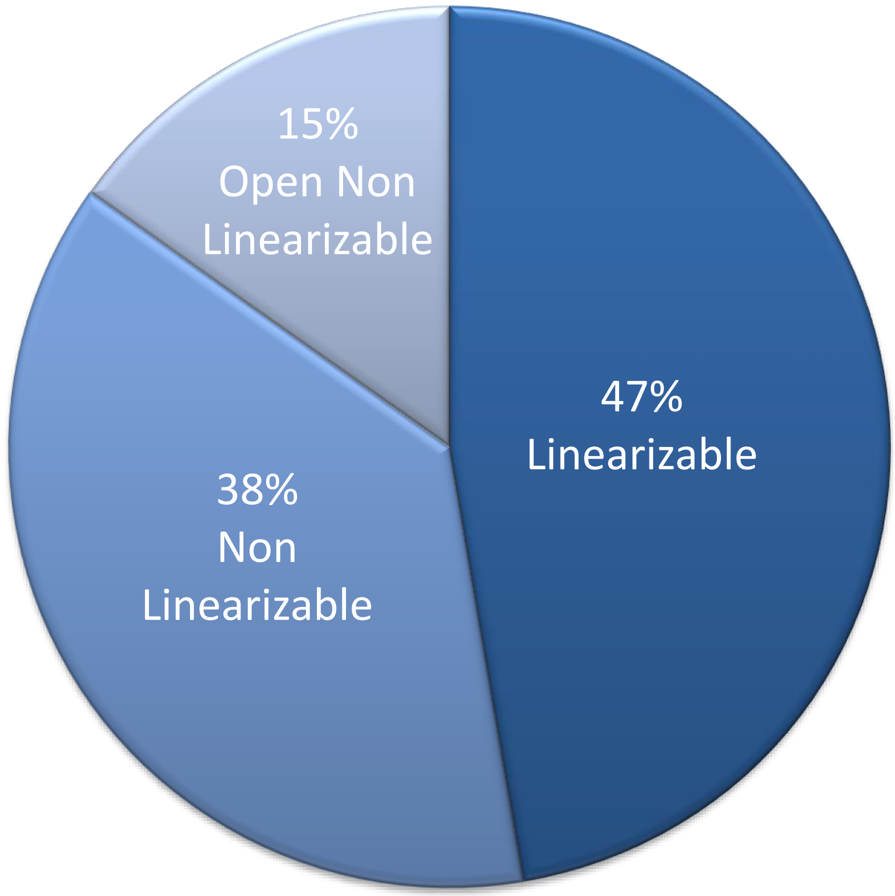
```
public void put(K k, V v) {
      if (this.eden.size() >= size) {
          this.longterm.putAll(this.eden);
          this.eden.clear();
       }
       this.eden.put(k, v);
}
```

Invariant: Every element that added to eden is either in eden or in longterm

OOPSLA’11 Shacham

- Search for all public domain collection operations methods with at least two operations
- Used simple static analysis to extract composed operations
  - Two or more API calls
- Extracted **112** composed operations from **55** applications
  - Apache Tomcat, Cassandra, MyFaces – Trinidad, …
- Check Linearizability of **all** public domain composed operations

Motivation: OOPSLA'11 Shacham



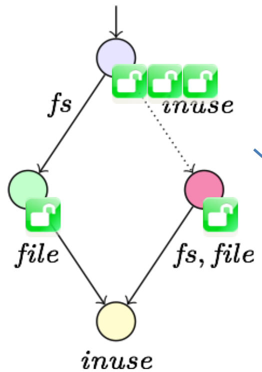
Relational Specification

- Program states as relations
  - Columns correspond to properties
  - Functional dependencies define global invariants

Atomic operation	meaning
r= empty	r := {}
insert r s t	if s $\notin$ r then r := r $\cup$ {<s.t>}
query r S C	The C of all the tuples in r matching tuple
remove r s	remove from r all the tuples which match s

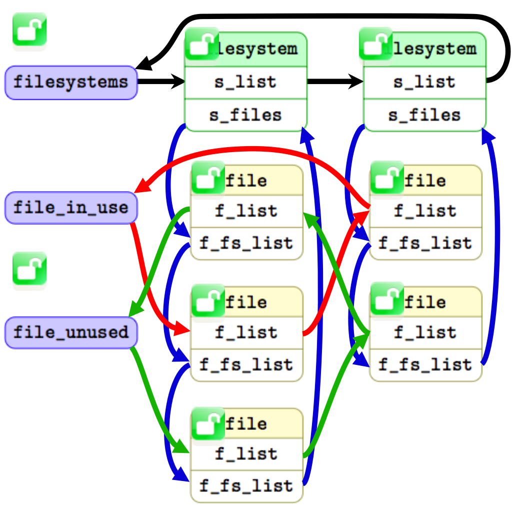
The High Level Idea

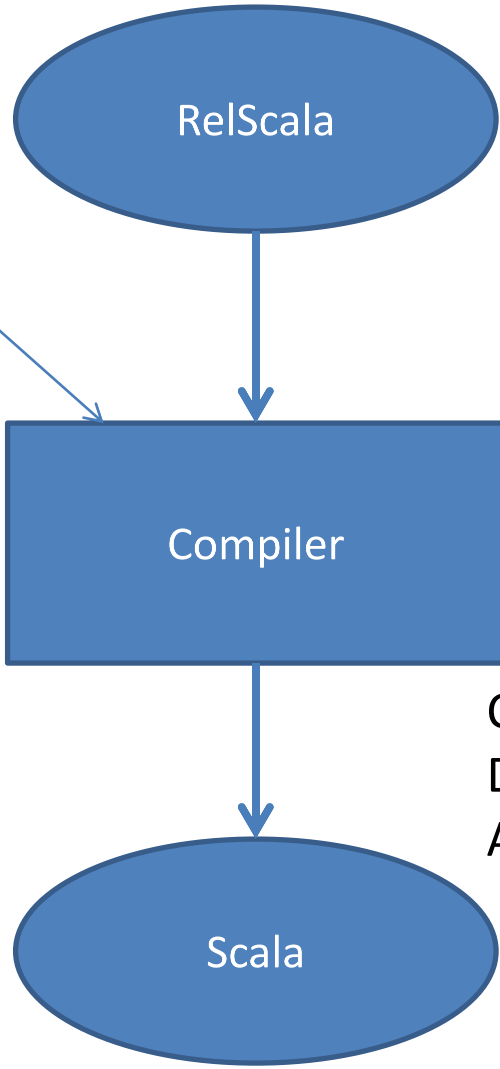
Concurrent Decomposition



ConcurrentHashMap

HashMap





$$\{fs, file, inuse\}$$

$$fs, file \to inuse$$

query <inuse:T> {fs, file}

Concurrent Compositions of
Data Structures,
Atomic Transactions

```
List * query(FS* fs, File* file) {
    lock(…) for (q= file_in_use; …)
    ….
```

Two-Phase Locking

Attach a lock to each piece of data

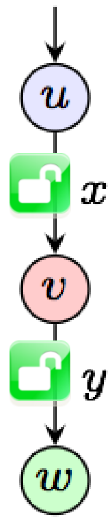
Two phase locking protocol:

- **Well-locked:** To perform a read or write, a thread must hold the corresponding lock
- **Two-phase:** All lock acquisitions must precede all lock releases

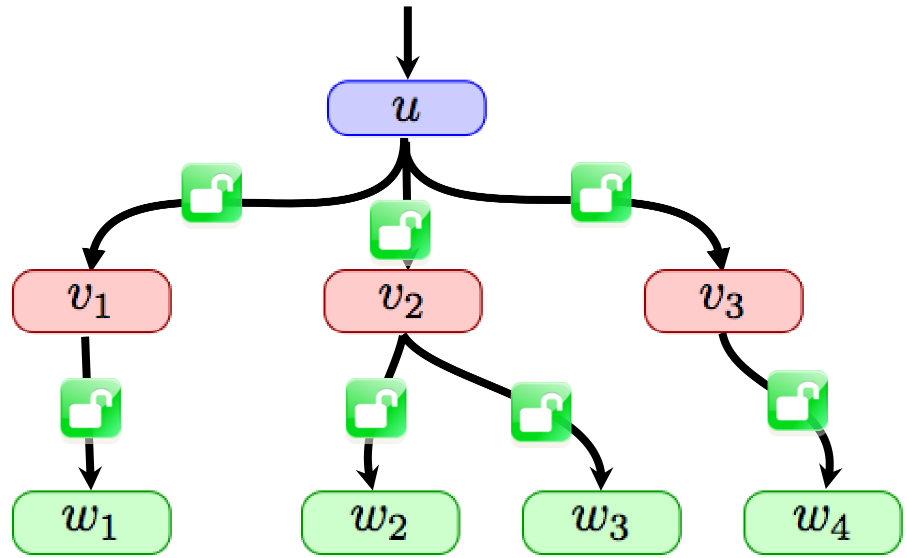
Theorem [Eswaran et al., 1976]: Well-locked, two-phase transactions are serializable

Two Phase Locking

Decomposition



Decomposition Instance



Attach a lock to every edge

Two Phase Locking ➔ Serialiazability

We're done!

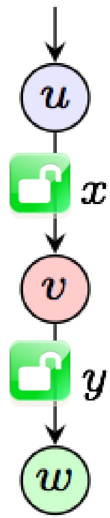
Problem 1: Can't attach locks to container entries

Problem 2: Too many locks

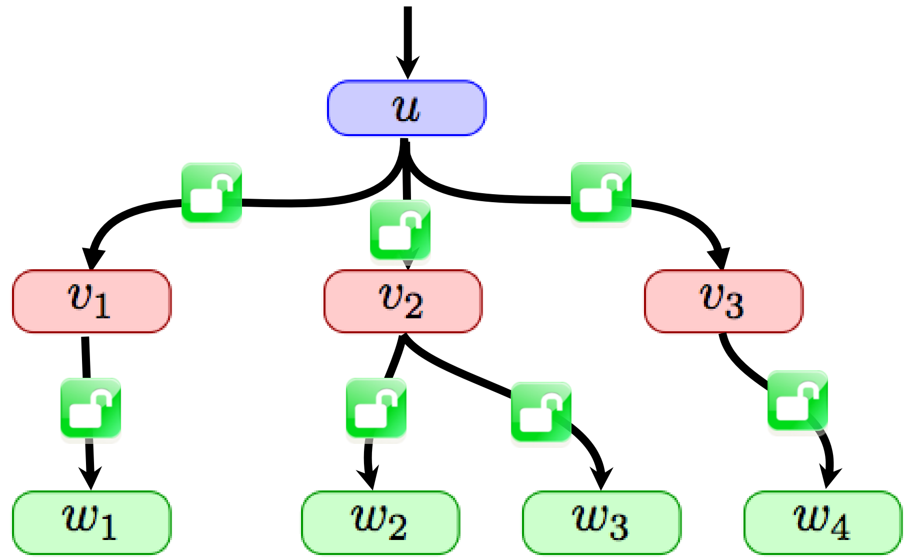
Butler Lampson/David J. Wheeler: "Any problem in computer science can be solved with another level of indirection."

Two Phase Locking

Decomposition



Decomposition Instance



Attach a lock to every edge

Two Phase Locking ➔ Serialiazability

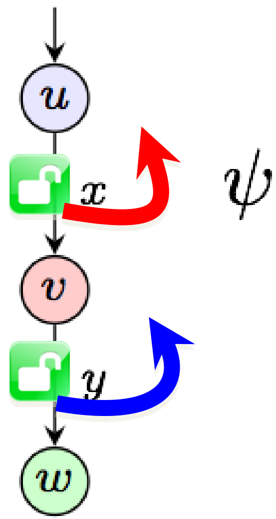
We're done!

Problem 1: Can't attach locks to container entries

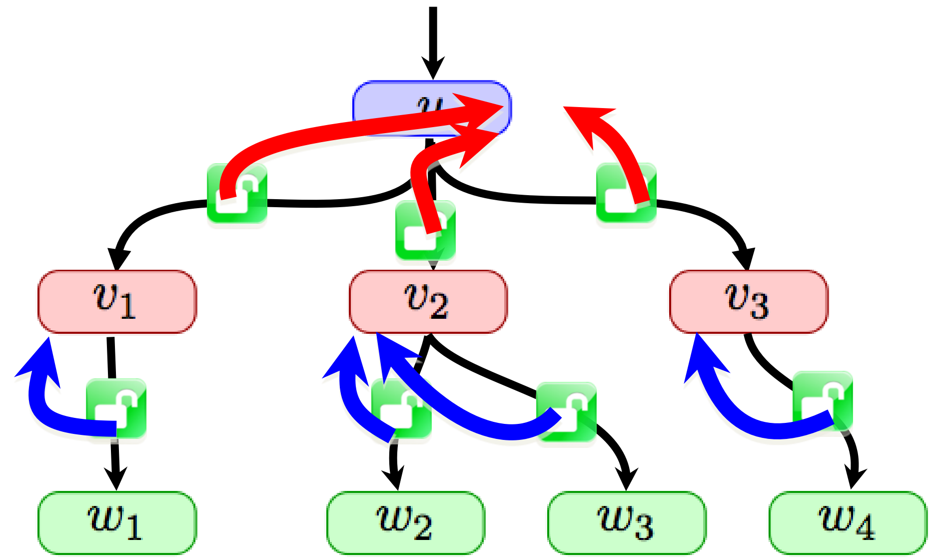
Problem 2: Too many locks

Lock Placements

Decomposition



Decomposition Instance

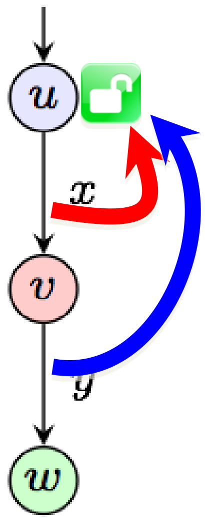


1. Attach locks to nodes

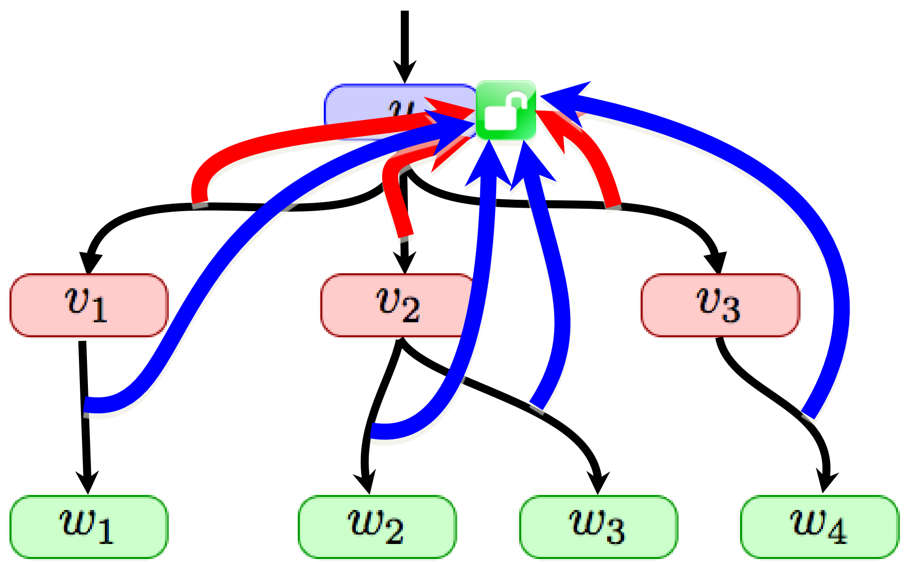
2. Use a *lock placement* ψ to map data (on edges)
to locks (on nodes)

Coarse-Grained Locking

Decomposition



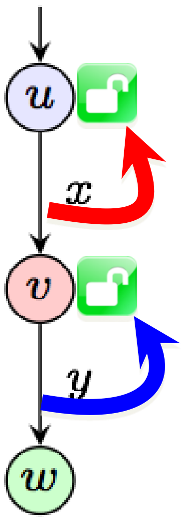
Decomposition Instance



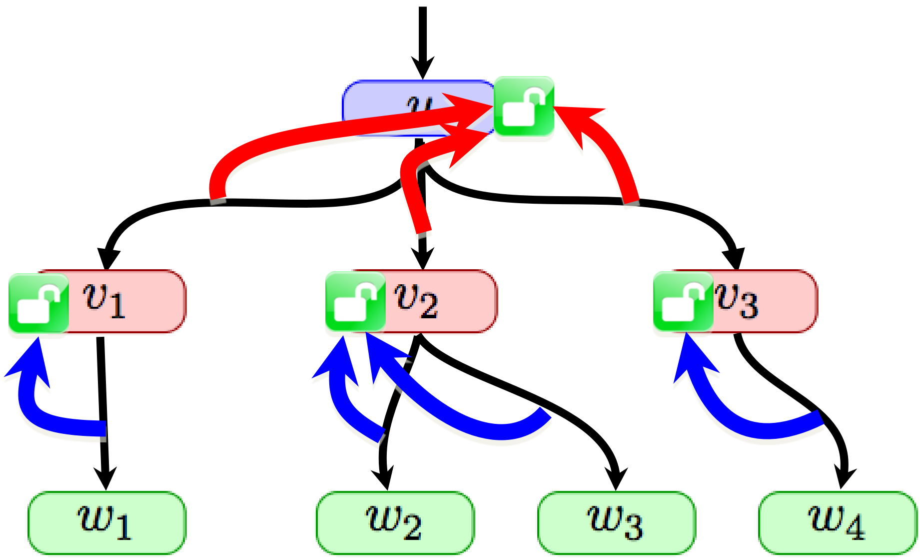
$$\psi = \{uv \mapsto u,\ vw \mapsto u\}$$

Finer-Grained Locking

Decomposition



Decomposition Instance

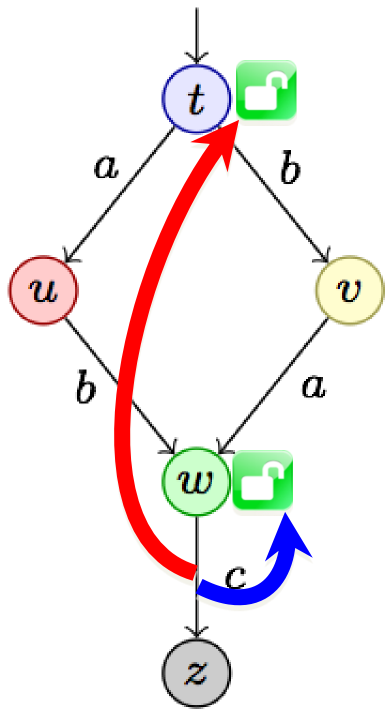


$$\psi = \{uv \mapsto u,\ vw \mapsto v\}$$

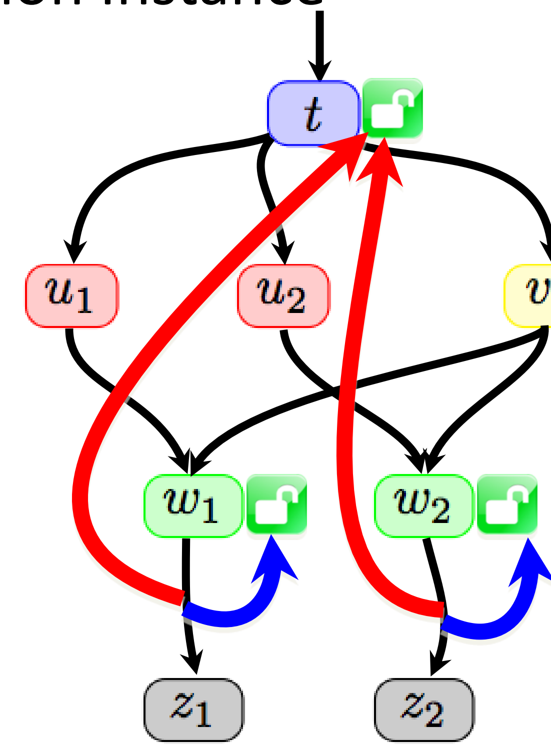
Lock Placements: Domination

Locks must dominate the edges they protect

Decomposition

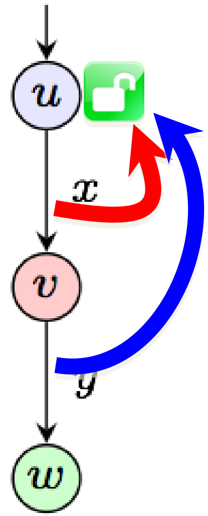


Decomposition Instance



Lock Placements: Path-Closure

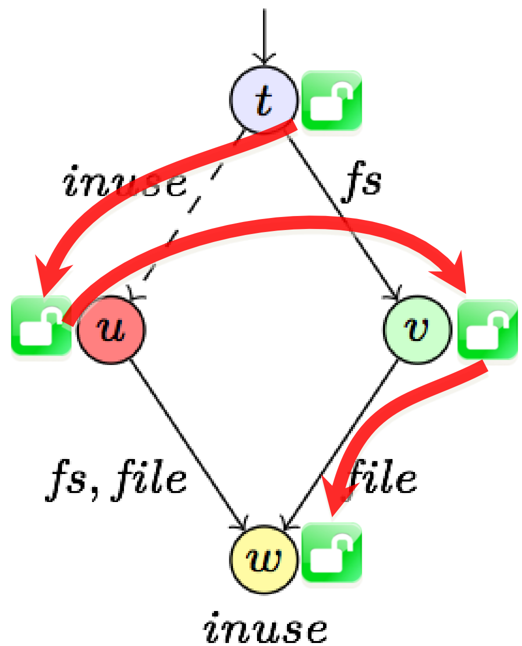
All edges on a path between an edge and its lock must share the same lock



If $\psi(vw) = u$, then $\psi(uv) = u$ also.

Lock Ordering

Prevent deadlock via a topological order on locks

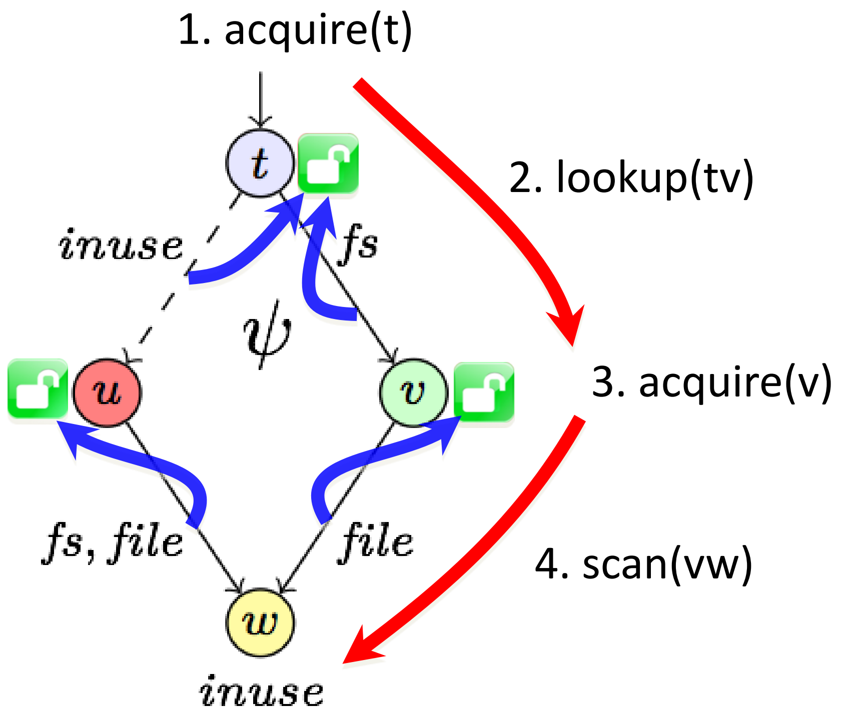


$$t \prec u \prec v \prec w$$

Queries and Deadlock

Query plans must acquire the correct locks in the correct order

$$t \prec u \prec v \prec w$$

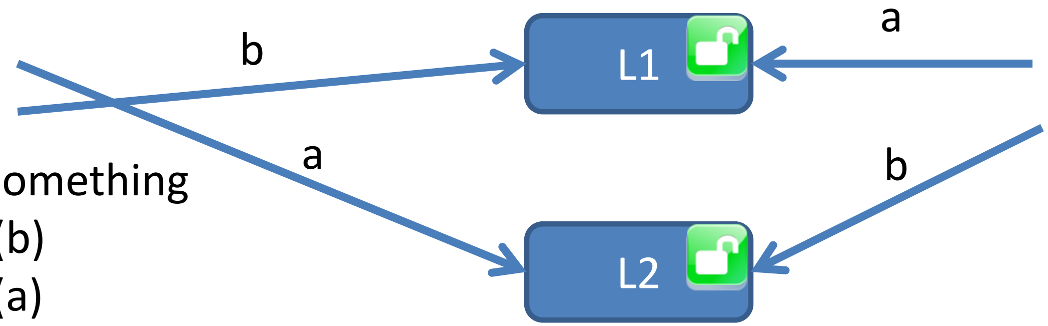


Example: find files on a particular filesystem

Deadlock and Aliasing

```
{
lock(a)
lock(b)
// do  something
unlock(b)
unlock(a)
}
```

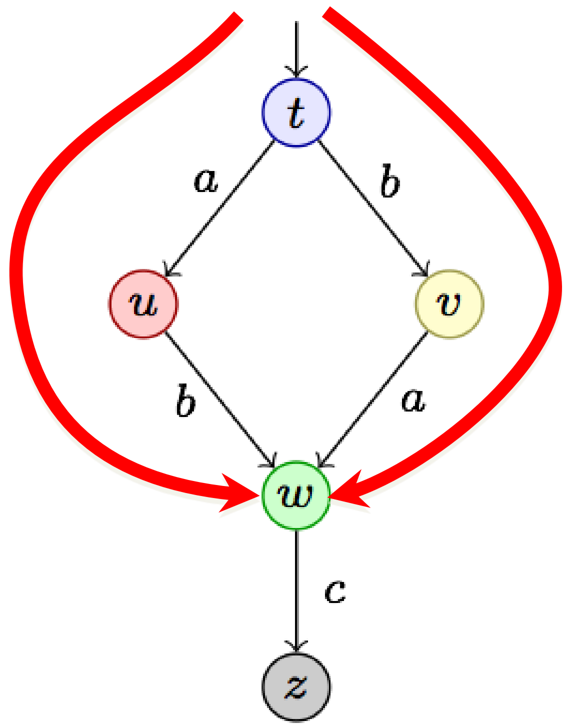
||




```
{
lock(a)
lock(b)
// do  something
unlock(b)
unlock(a)
}
```

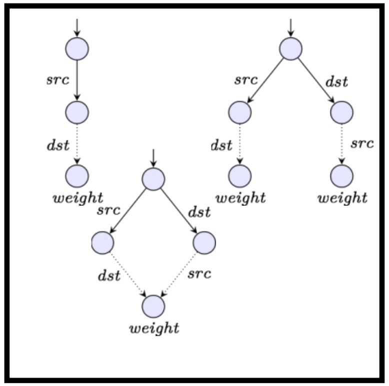
Decompositions and Aliasing

- A decomposition is an abstraction of the set of potential aliases
- Example: there are *exactly* two paths to any instance of node *w*



Concurrent Synthesis (Autotuner)

Find optimal combination of



Decomposition

×

Array TreeMap

HashMap

LinkedList

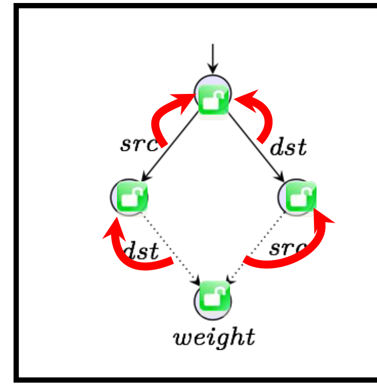
ConcurrentHashMap

ConcurrentSkipListMap

CopyOnWriteArrayList

Container
Data Structures

×



Lock Placement

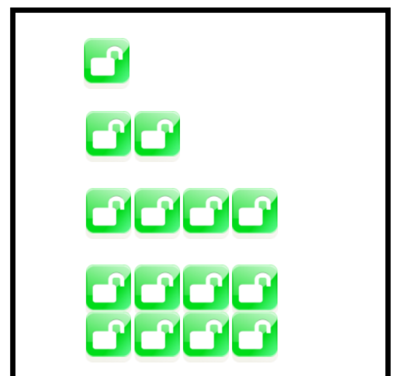
×

ReentrantLock

ReentrantReadWriteLock

Lock Implementations

×



Lock Striping Factors

Based on Herlihy's benchmark of concurrent maps

Concurrent Graph Benchmark

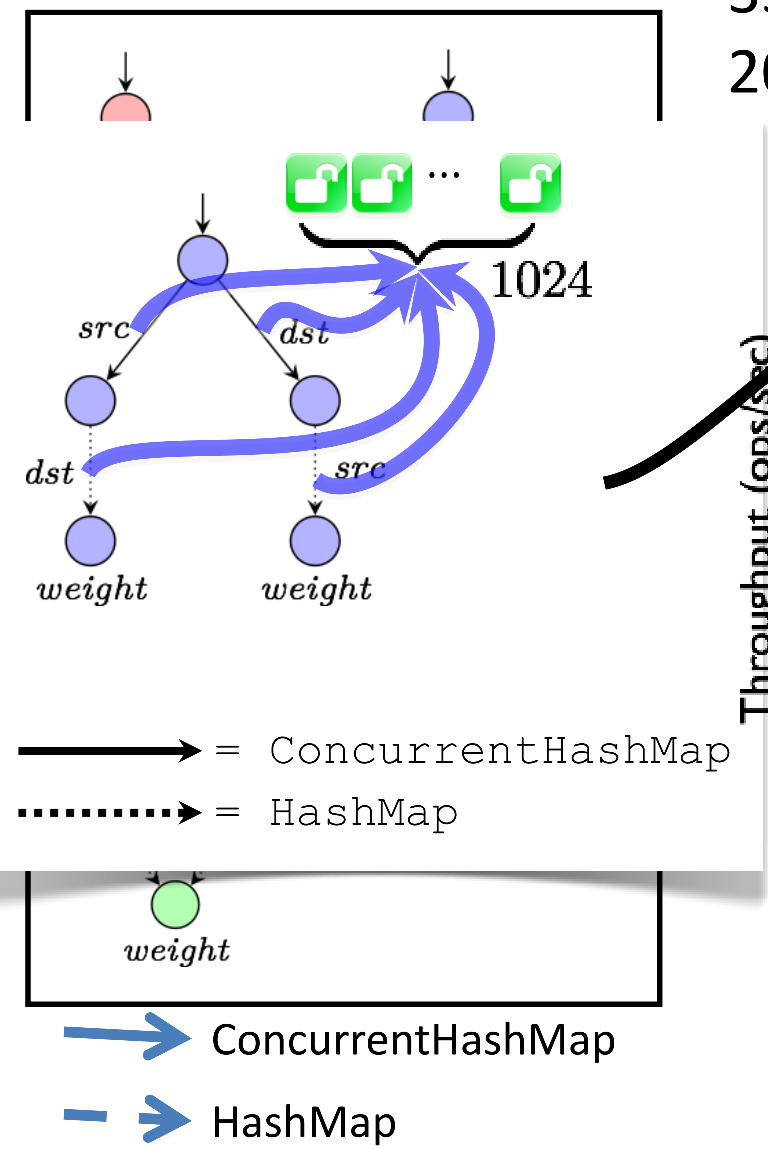
$$\{src, dst, weight\}$$

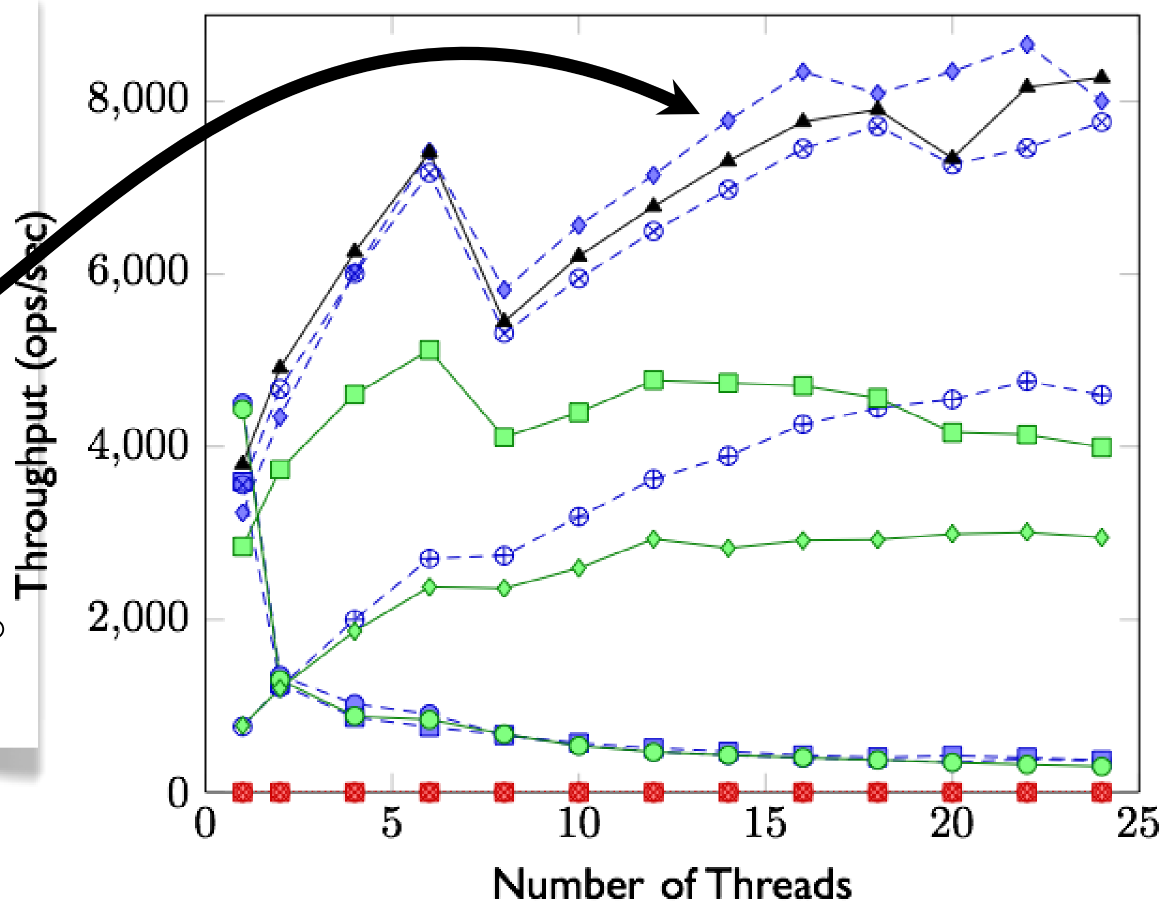
$$src, dst \rightarrow weight$$

- Start with an empty graph
- Each thread performs 5 x 10^5 random operations
- Distribution of operations a-b-c-d (a% find successors, b% find predecessors, c% insert edge, d% remove edge)
- Plot throughput with varying number of threads

Results: 35-35-20-10

35% find successor, 35% find predecessor, 20% insert edge, 10% remove edge





(Some) Related Projects

- SETL
- Relational synthesis: [Cohen & Campbell 1993], [Batory & Thomas 1996], [Smaragdakis & Batory 1997], [Batory et al. 2000] [Manevich, 2012] …
- Two-phase locking and Predicate Locking [Eswaran et al., 1976], Tree and DAG locking protocols [Attiya et al., 2010], Domination Locking [Golan-Gueta et al., 2011]
- Lock Inference for Atomic Sections: [McCloskey et al.,2006], [Hicks, 2006], [Emmi, 2007]

Summary

- Programming with uniform relational abstraction
  - Increase the gap between data abstraction and low level implementation
- Comparable performance to manual code
- Easier to evolve
- Automatic data structure selection
- Easier for program reasoning

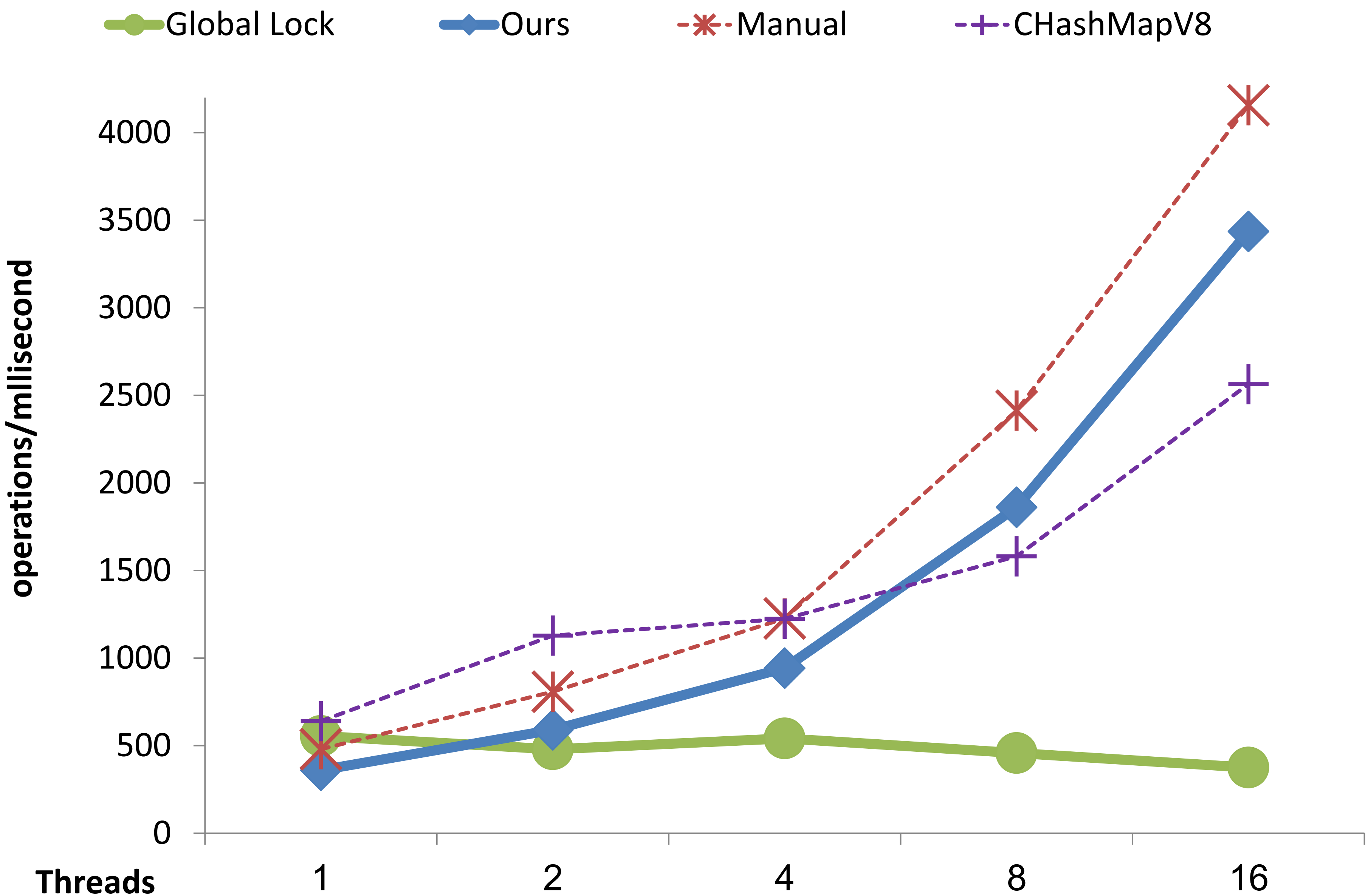
Concurrent Libraries with Foresight PLDI'13

Guy Gueta(TAU)
G. Ramalingam (MSR)
M. Sagiv (TAU)
E. Yahav (Technion)

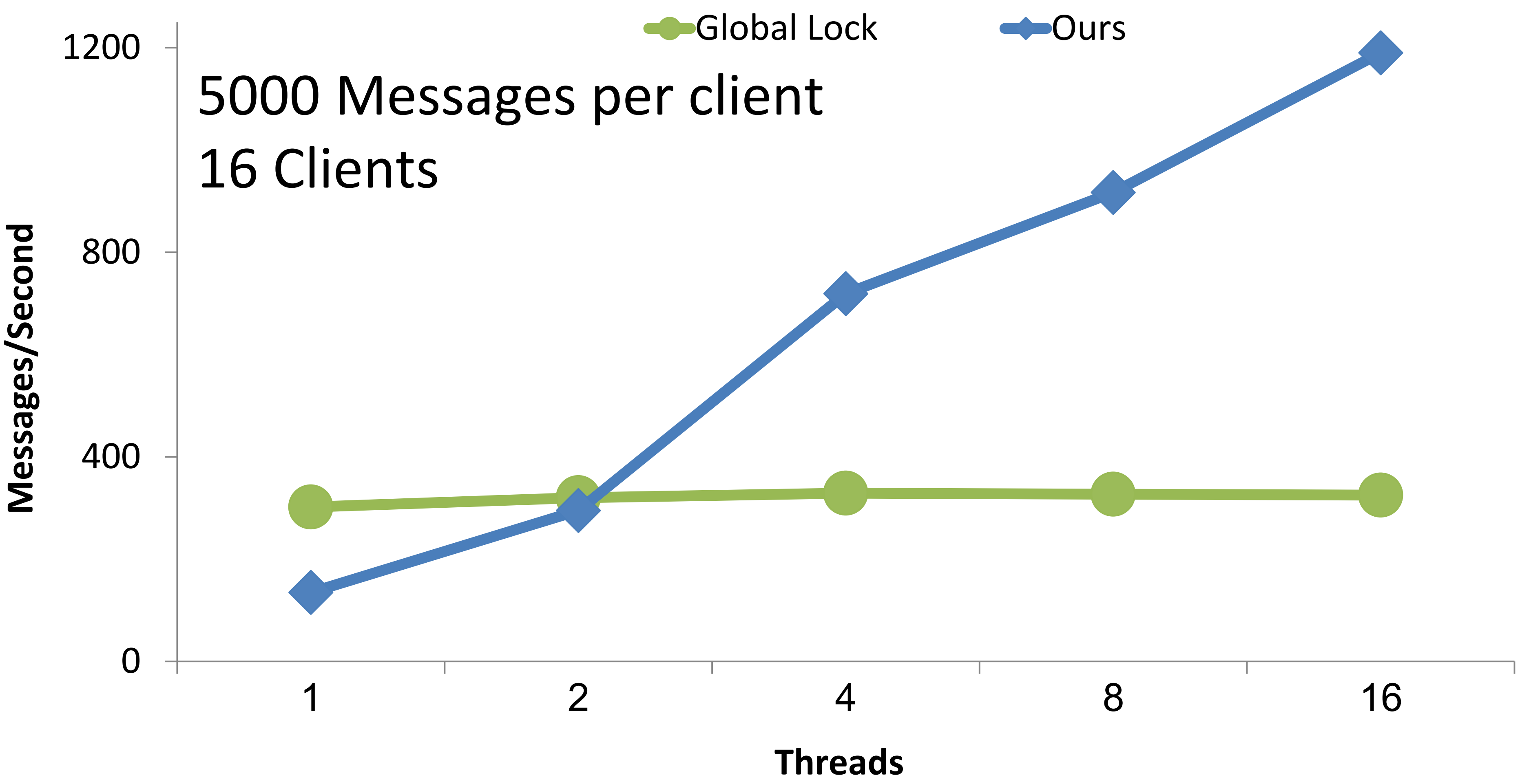
Transactional Libraries with Foresight

- Enforce atomicity of arbitrary sequences
- The client declares intended operations
  - foresight
- The library utilizes the specification
  - Synchronize between operations which do not serialize with foresight
- Methodology for creating libraries with foresights
  - Maps
- Foresight can be automatically inferred by sequential static program analysis

ComputeIfAbsent (single Map)



GossipRouter (multiple Maps)



Summary

- Methods for enforcing atomicity of sequences of operations
- Provably correct
- Simplifies reasoning
  - Sequential reasoning
  - High level data structures & invariants
- Is that efficient enough?
  - Pessimistic concurrency
  - Optimistic concurrency