Lecture 17:

Distinctness Analysis

Guest Lecture by Chris Fallin

Finding and Exploiting Parallelism with Data-Structure-Aware Static and Dynamic Analysis

Chris Fallin

15-745 Lecture

(derived from thesis defense)

February 27, 2019

Outline

- Introduction
- First-Class Data Structures
- DAEDALUS: Distinctness Analysis
- ICARUS: Incorporating Dynamic Checks

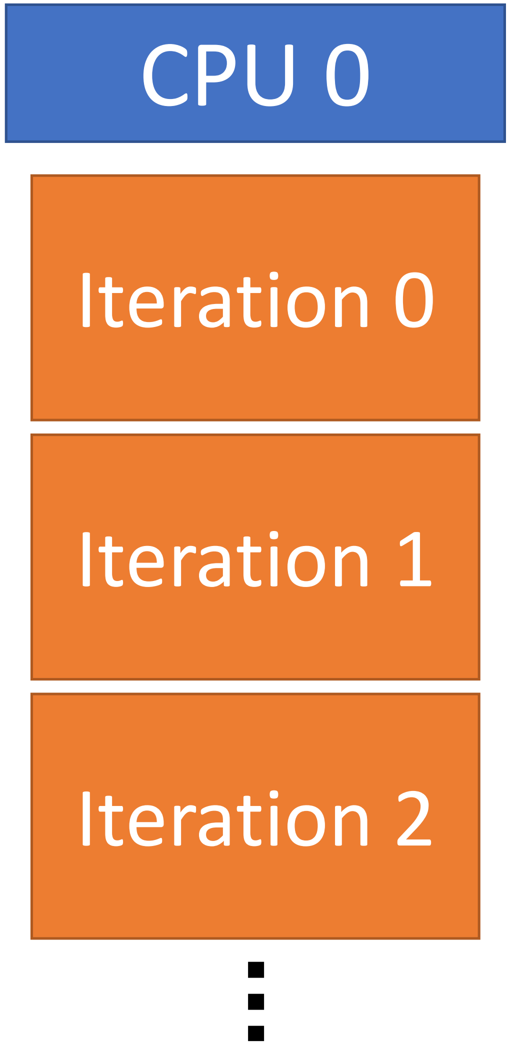
Problem: High-Level Program Optimizations are Difficult

We have the following legacy serial code that we wish to optimize:

```
HashMap<Item, Result> results = new HashMap<>();
List<Item> items = …;

for (Item it : items) {
  Result r = it.analyze();
  results.put(it, r);
}
```

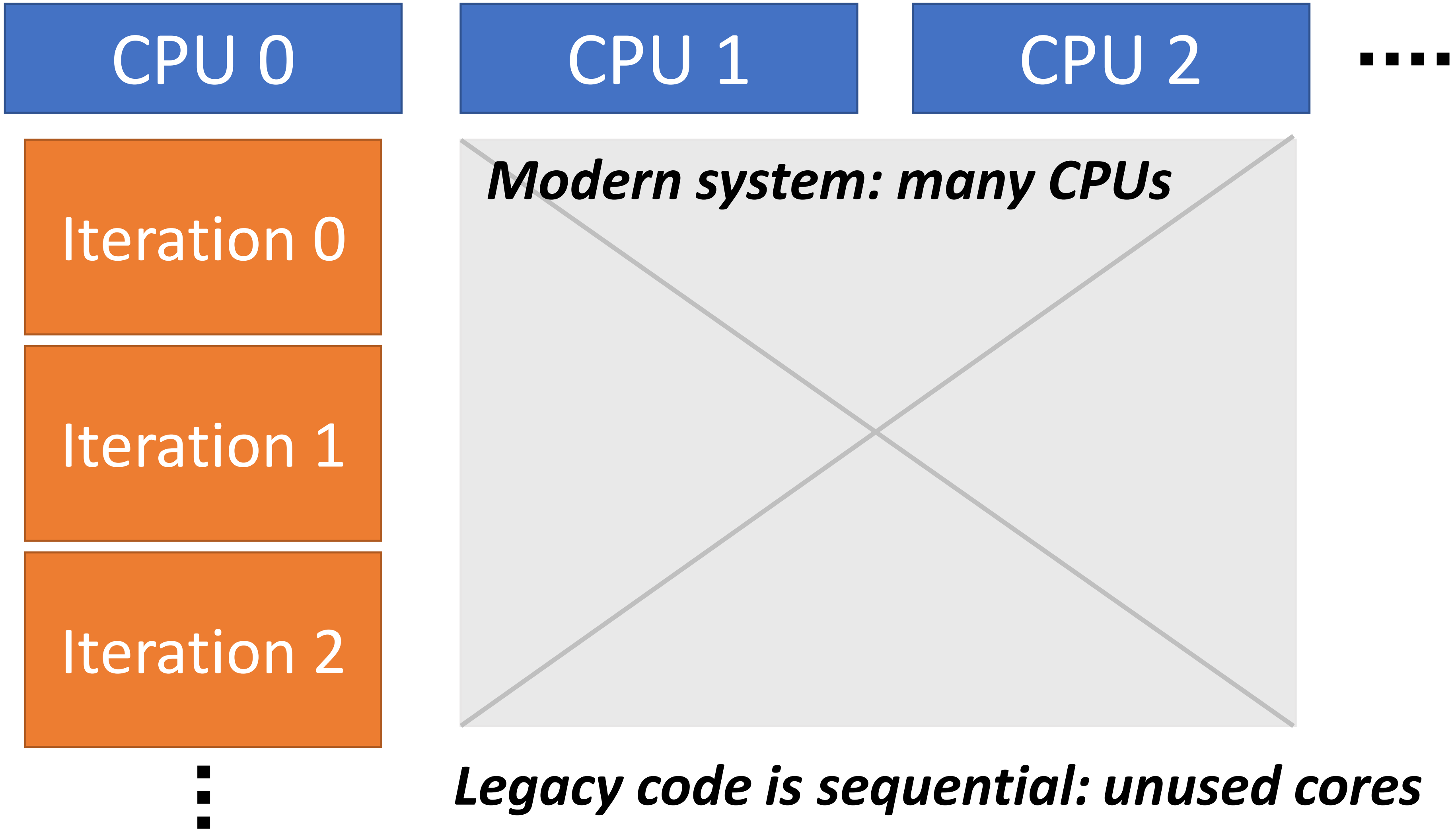
Hot loop

Problem: High-Level Program Optimizations are Difficult

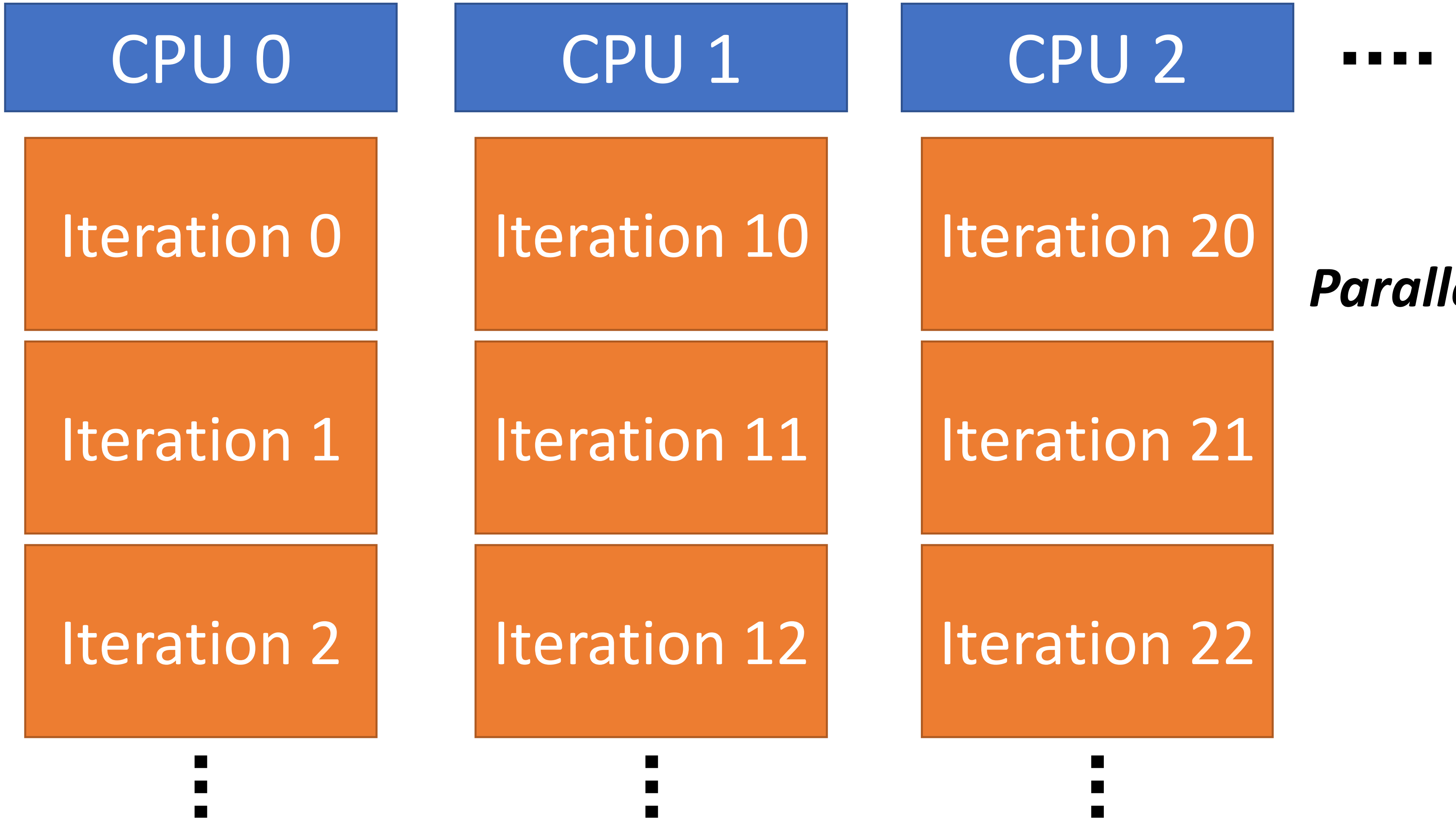


Legacy system: single CPU

Problem: High-Level Program Optimizations are Difficult



Problem: High-Level Program Optimizations are Difficult



Parallelize the Loop?

Problem: High-Level Program Optimizations are Difficult

We have the following legacy serial code that we wish to optimize:

```
HashMap<Item, Result> results = new HashMap<>();
List<Item> items = …;

for (Item it : items) {
  Result r = it.analyze();
  results.put(it, r);
}
```

Hot loop

Problem: High-Level Program Optimizations are Difficult

We have the following legacy serial code that we wish to optimize:

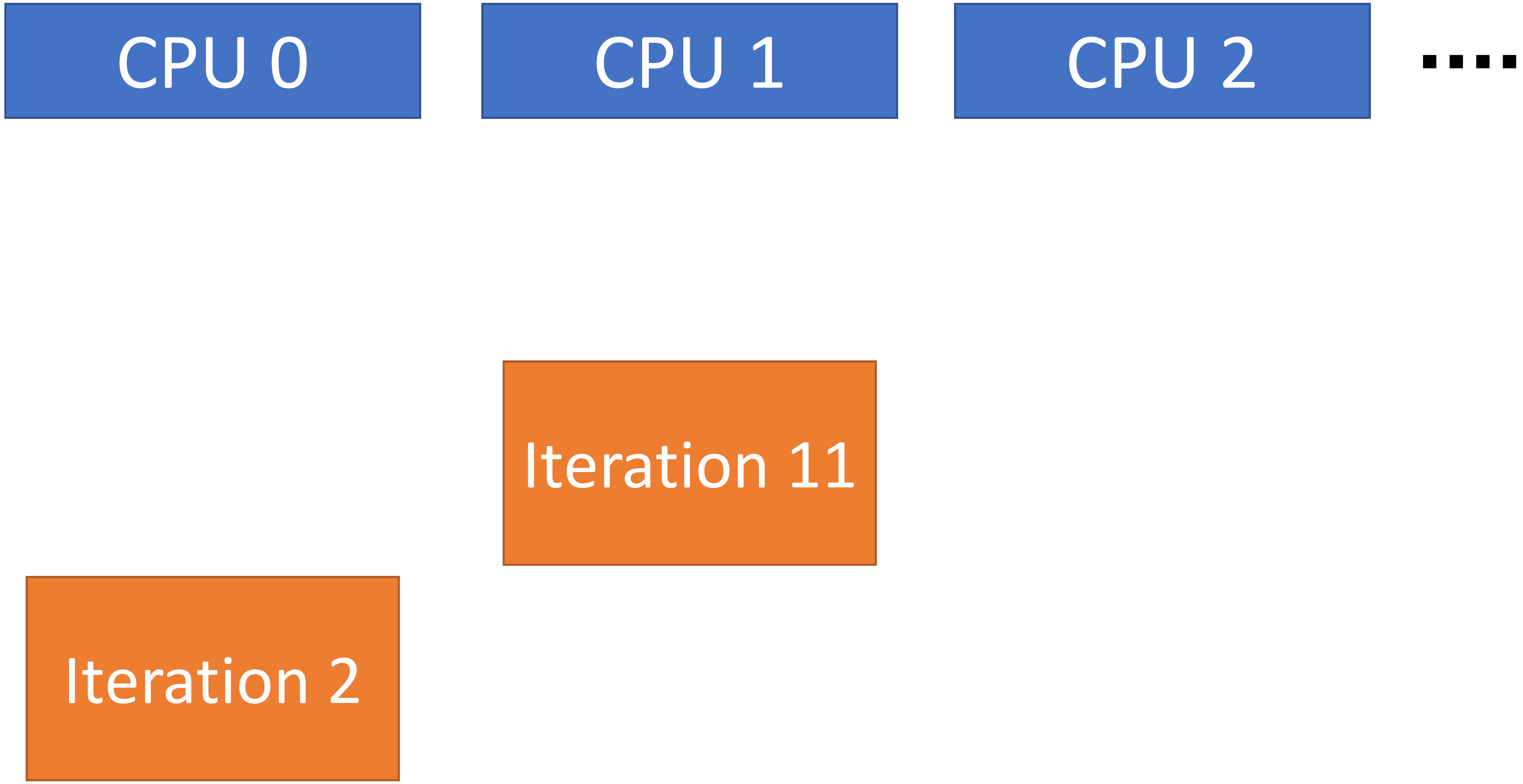
```
HashMap<Item, Result> results = new HashMap<>();
List<Item> items = …;

items.parallelStream().forEach(it -> {  // parallel-for
  Result r = it.analyze();
  results.put(it, r);
});
```

Problem: High-Level Program Optimizations are Difficult

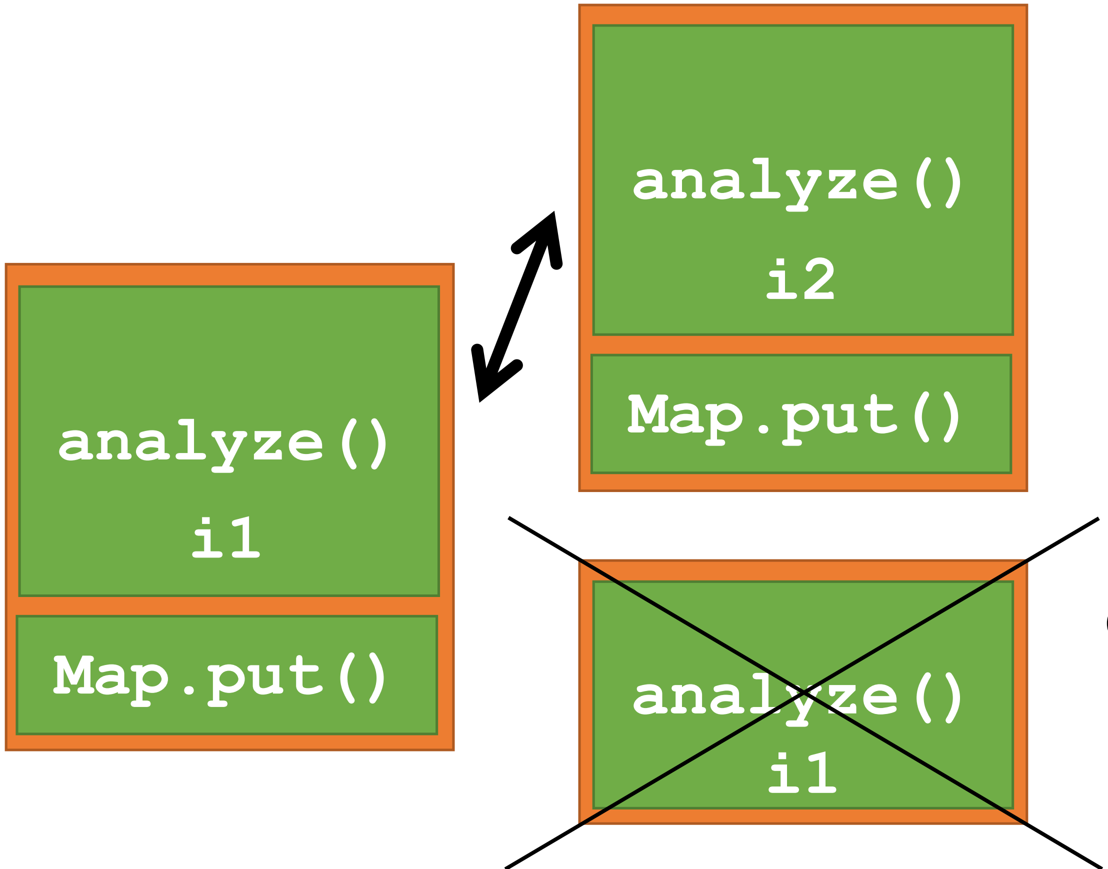
We have the following legacy serial code that we wish to optimize:

```
HashMap<Item, Result> results = new HashMap<>();
List<Item> items = …;

items.parallelStream().forEach(it -> {
  Result r = it.analyze();
  synchronized (results) {    // locking on shared Map
    results.put(it, r);
  }
});
```

Problem: High-Level Program Optimizations are Difficult



Problem: High-Level Program Optimizations are Difficult



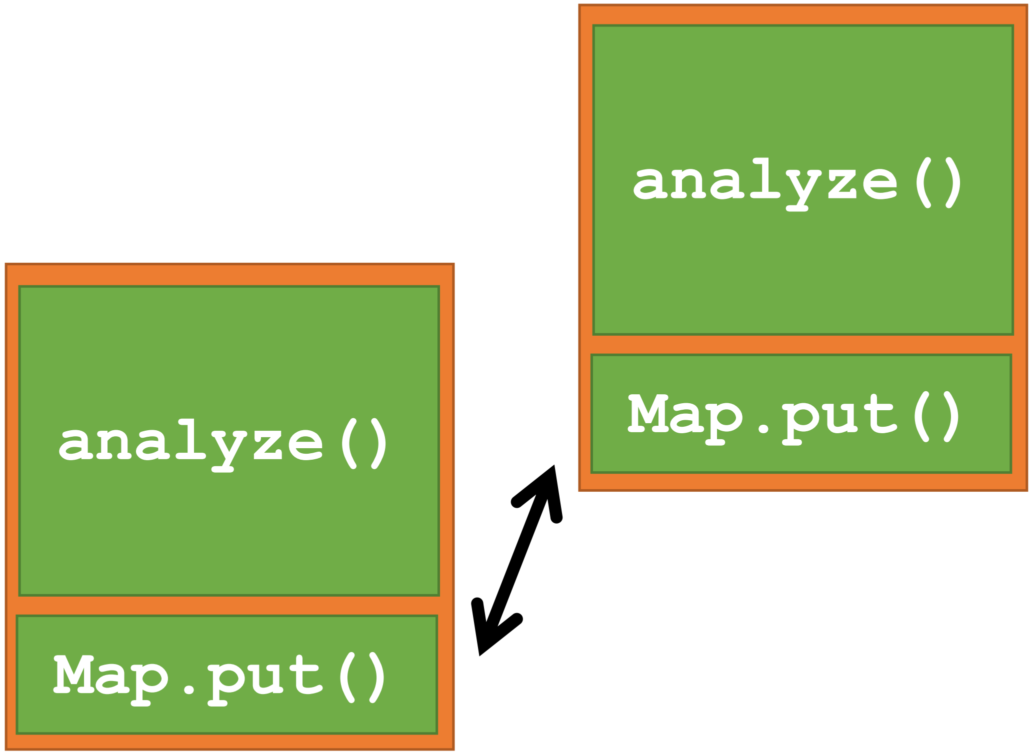
Is this sound?

- **`analyze()`** calls overlap and occur out-of-order

Invariant 1. Each **`analyze()`** instance reads/writes only its own **`Item`**.

Invariant 2. Each **`Item`** occurs in **`list`** exactly once.

Problem: High-Level Program Optimizations are Difficult



Is this sound?

- **`Map.put()`** calls occur out-of-order

Fact (API semantics). Map insertions are commutative for two keys **`k1`** ≠ **`k2`**.

Invariant 2 (again). Each **`Item`** occurs in **`list`** exactly once.

Problem: High-Level Program Optimizations are Difficult

- Human refactors by understanding *high-level invariants & semantics:*
  - **(Data Structure API)** Key-value map insertions are commutative when accessing two *different* keys.
  - **(Program invariant)** **`Item.analyze()`** accesses only **`this`**.
  - **(Program invariant)** No element appears in **`list`** more than once.

- Could the compiler do this too?

Could a Compiler Analysis Derive This?

- **(Data Structure API)** Key-value map insertions are commutative when accessing two *different* keys.

Human sees:

```
map.put(k, v);
```

Compiler sees:

```
void put(K key, V value) {
  int h = key.x * 8931 + key.y;
  Node n = new Node(key, value);
  n.next = slots[h];
  slots[h] = n;
}
```

- Unlikely to derive commutativity from first principles without help
- Similarly, "no duplicate elements in `list`" is very difficult

Solution: Domain-Specific Languages?

- DSLs separate *algorithm* and *implementation*!

Example: SQL

```
SELECT a, b FROM table WHERE a > 1
```

↓ *Query planner*

```
PROJECT t0.a, t0.b
  FILTER t0.a > 1
    SCAN table AS t0
```

Solution: Domain-Specific Languages?

- DSLs separate *algorithm* and *implementation*!

```
UniqueList items = …;

HashMap results = items.buildMap(it -> analyze(it));

pure Result analyze(Item it) {
  // can only access it and newly-allocated objects
  // …
}
```

Solution: Domain-Specific Languages?

- DSLs separate *algorithm* and *implementation*!
- But, not always applicable:
  - ***Legacy code***: already exists (rewrite costs effort + risk)
  - ***Mixed applications:*** multiple kernels (DSL integration?)
  - ***DSLs with limitations:*** a program may not map cleanly onto DSL

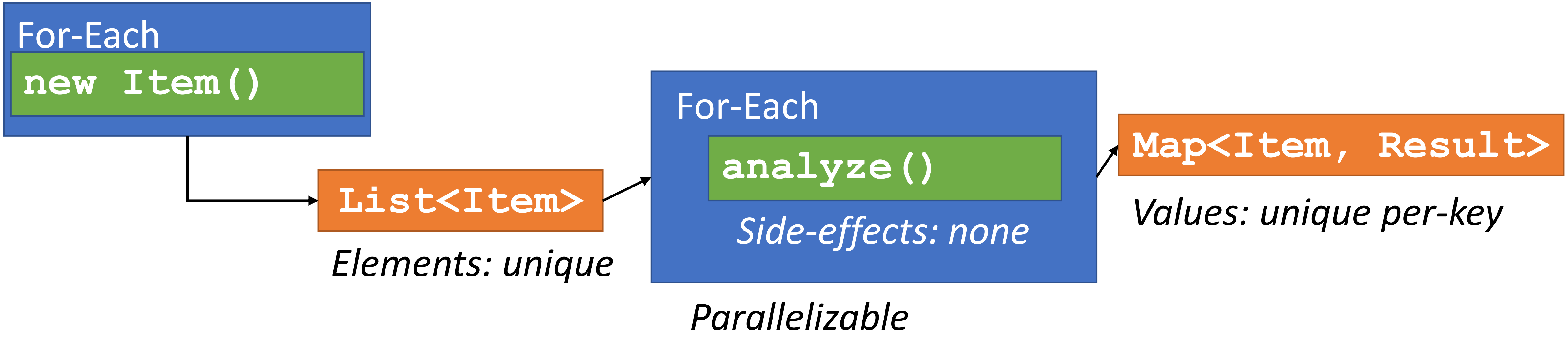
Our Approach: General Language + Analysis

- We want the full expressive power of a general-purpose language
- We want to *derive* the programmer-level understanding with analyses

```
HashMap<Item, Result> results = new HashMap<>();
List<Item> items = …;

for (Item it : items) {
  Result r = it.analyze();
  results.put(it, r);
}
```

Our Approach: General Language + Analysis

- We want the full expressive power of a general-purpose language
- We want to *derive* the programmer-level understanding with analyses



Outline

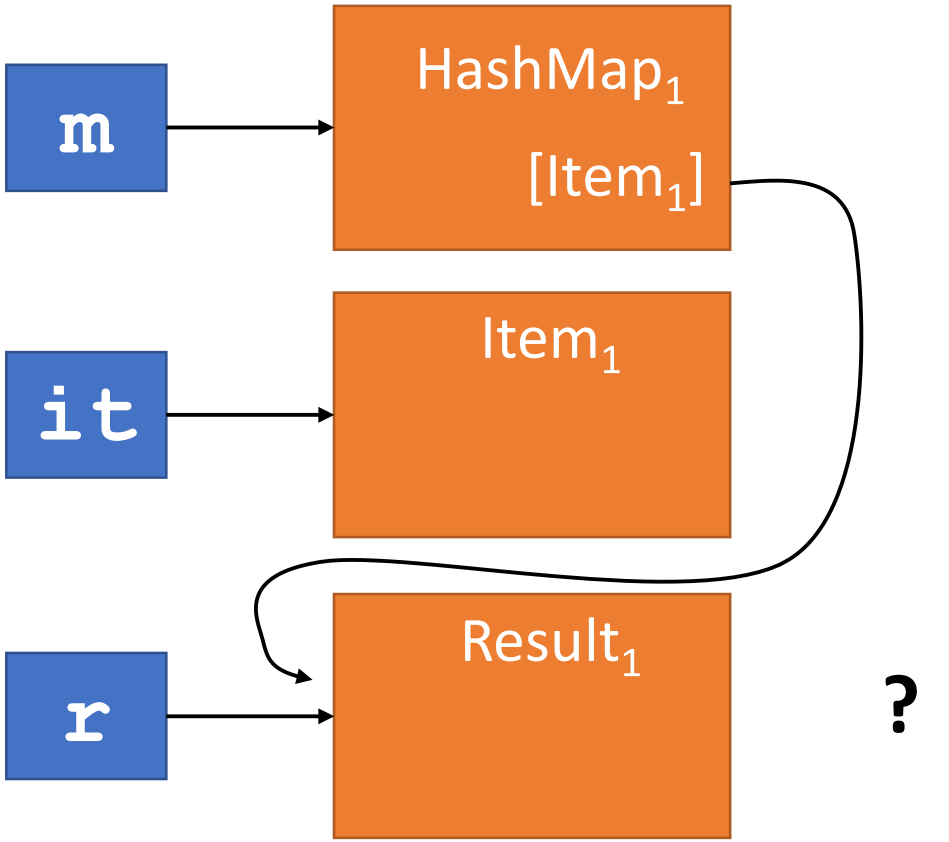
- Introduction
- First-Class Data Structures
- DAEDALUS: Distinctness Analysis
- ICARUS: Incorporating Dynamic Checks

Outline

- Introduction
- First-Class Data Structures
- DAEDALUS: Distinctness Analysis
- ICARUS: Incorporating Dynamic Checks

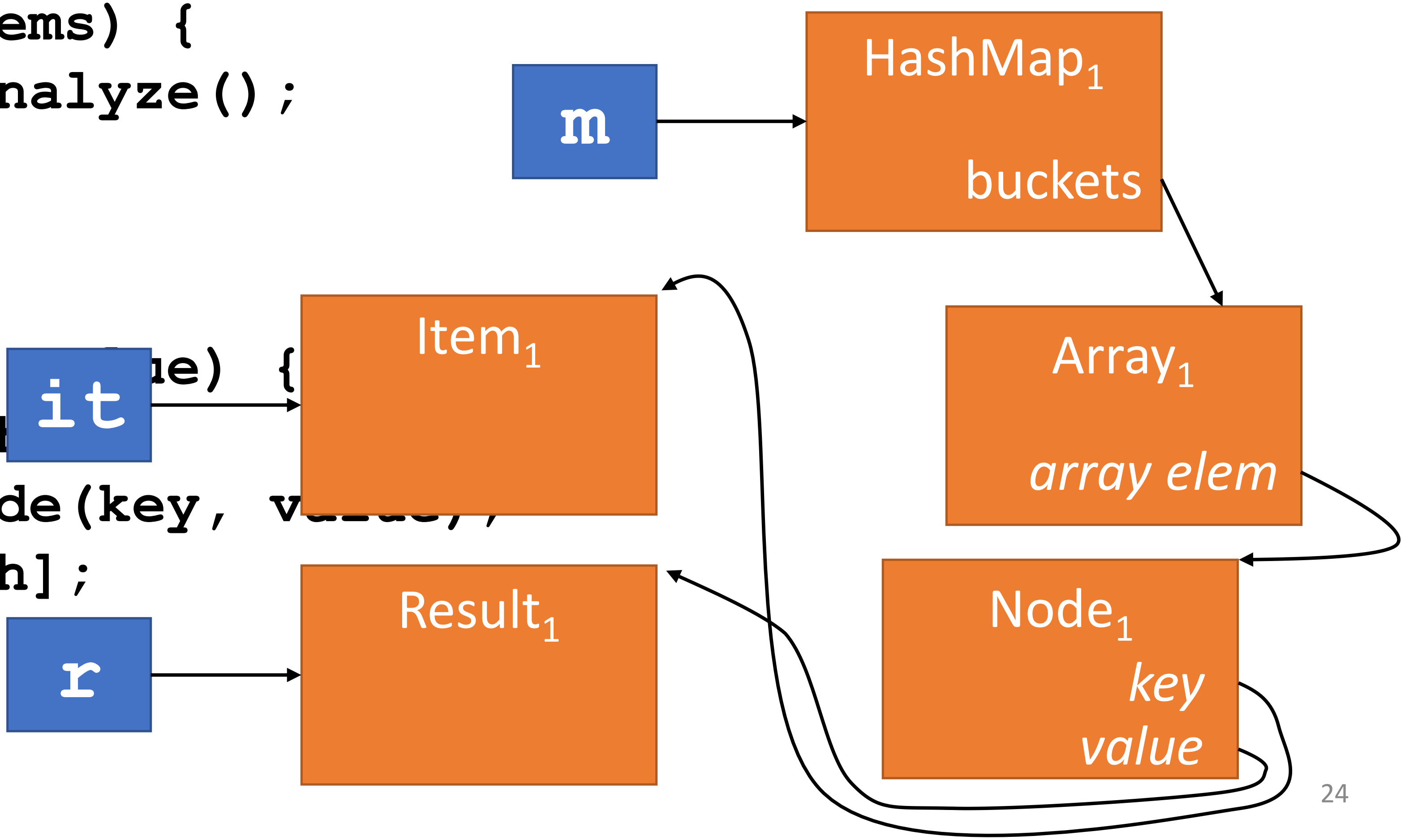
Points-to Analysis of a Hash Map

```
Map<Item, Result> m = new HashMap<>();

for (Item it : items) {
  Result r = it.analyze();
  m.put(it, r);
}
```

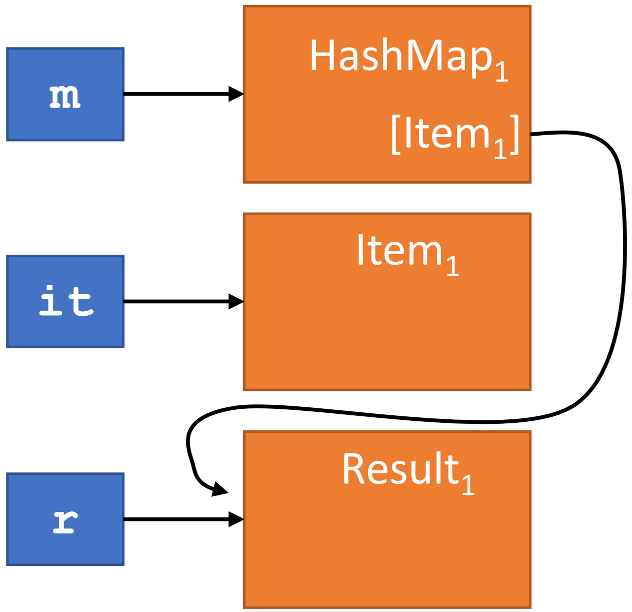


Points-to Analysis of a Hash Map

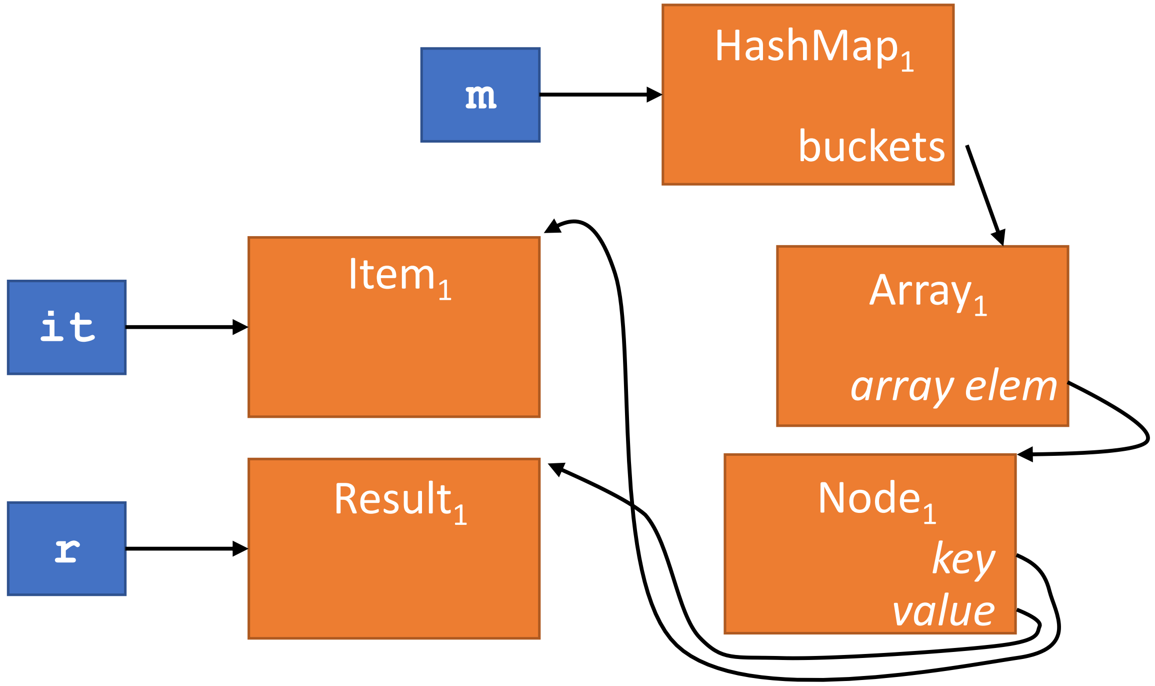
```
Map<Item, Result> m = new HashMap<>();

for (Item it : items) {
  Result r = it.analyze();
  m.put(it, r);
}

void put(K key, V    ue) {
  int h = key.has
  Node n = new Node(key, value);
  n.next = slots[h];
  slots[h] = n;
}
```



Points-to Analysis of a Hash Map



vs.



Points-to Analysis of a Hash Map: Problems

- **Problem 1:** All value slots in key-value map artificially merged into one points-to set

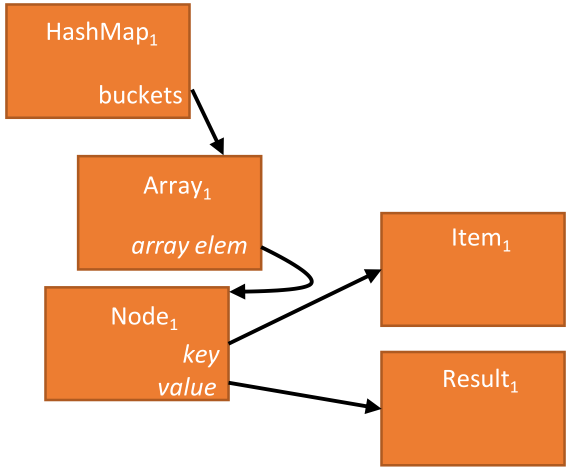
```
Key k1 = new Key(…);
Key k2 = new Key(…);
map.put(k1, v1);
map.put(k2, v2);
Value v = map.get(k); // pts-to set {v1, v2}
```

- **Problem 2:** Analysis will not reveal commutativity
  - Reordering operations produces a different heap (but Map.get() doesn't care)
- **Problem 3:** Analysis of implementation is not scalable

Solution: First-Class Data Structures

- **Key Idea**: provide *compiler intrinsics* for key-value maps and lists so that analyses can reason directly about these data structures

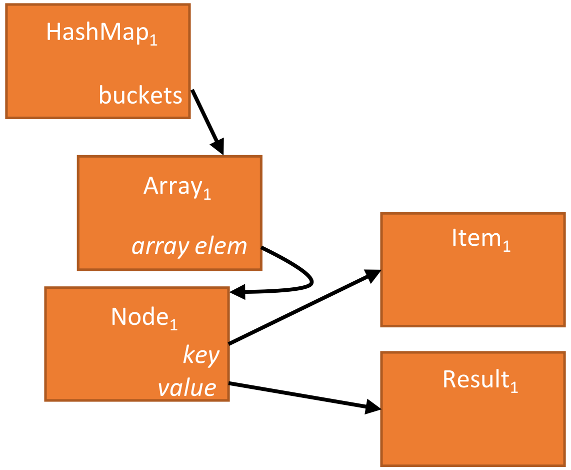
```
void put(K key, V value) {
  int h = key.hash();
  Node n = new Node(key, value);
  n.next = slots[h];
  slots[h] = n;
}
```



Solution: First-Class Data Structures

- **Key Idea**: provide *compiler intrinsics* for key-value maps and lists so that analyses can reason directly about these data structures
- *Part 1*: replace implementation in library with an equivalent model

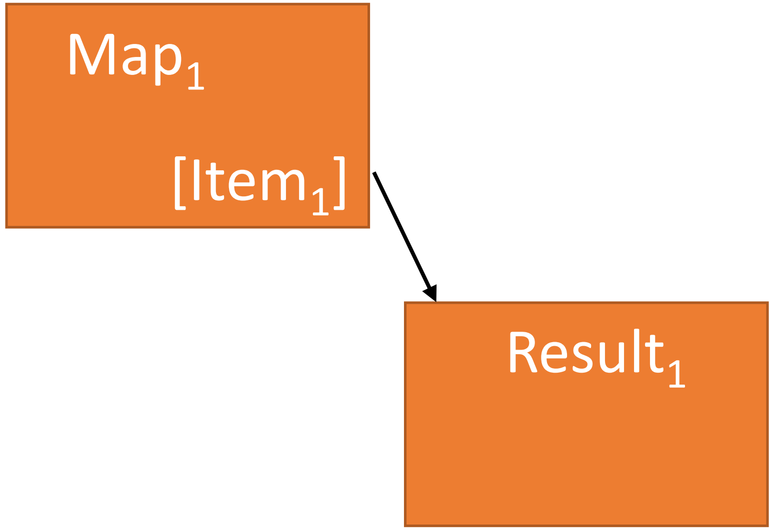
```
model void put(K key, V value) {
  mapput this.m, key, value;
}
```



Solution: First-Class Data Structures

- **Key Idea**: provide *compiler intrinsics* for key-value maps and lists so that analyses can reason directly about these data structures
- *Part 1*: replace implementation in library with an equivalent model
- *Part 2*: define intrinsics and extend points-to analysis

```
model void put(K key, V value) {
  mapput this.m, key, value;
}
```



Semantic Models: Explicit Library Semantics

- **Key Idea**: replace portions of program *as analyzed* with simpler logic
  - Modify callgraph during analysis: resolve to "model override" methods

Callgraph

void MyClass.f()

```
for (Item it : items) {
  Result r = it.analyze();
  m.put(it, r);
}
```

void HashMap.put(…)

```
int h = key.hash();
Node n = new Node(key, value);
n.next = slots[h];
slots[h] = n;
```

Semantic Models: Explicit Library Semantics

- **Key Idea**: replace portions of program *as analyzed* with simpler logic
  - Modify callgraph during analysis: resolve to “model override” methods

Callgraph

void MyClass.f()

```
for (Item it : items) {
  Result r = it.analyze();
  m.put(it, r);
}
```

model void Map.put(…)

```
mapput this.m, key, value;
```

Semantic Models: Conservative Behavior

- Models are **conservative**
  - May have additional side-effects: overapproximate accessed-memory footprint
  - May return additional or “unknown” values

```
void HashMap.equals(Object o) {
  for (Entry e : this) {
    if (!e.value().equals(other.get(e.key())) {
      return false;
    }
  }
  return true;
}
```

Semantic Models: Conservative Behavior

- Models are **conservative**
  - May have additional side-effects: overapproximate accessed-memory footprint
  - May return additional or “unknown” values

```
model void HashMap.equals(Object other) {
  // conservatively call `.equals()` on all items
  for (Key k : mapkeyiter this.m) {
    e.equals(e);
  }
  // likewise for `other`

  return unknown;  // could be `true` or `false`
}
```

First-Class Key-Value Maps

- **Key Idea:** provide key-value maps as new language-level object type
  - On the same level as arrays, or heap objects with fields

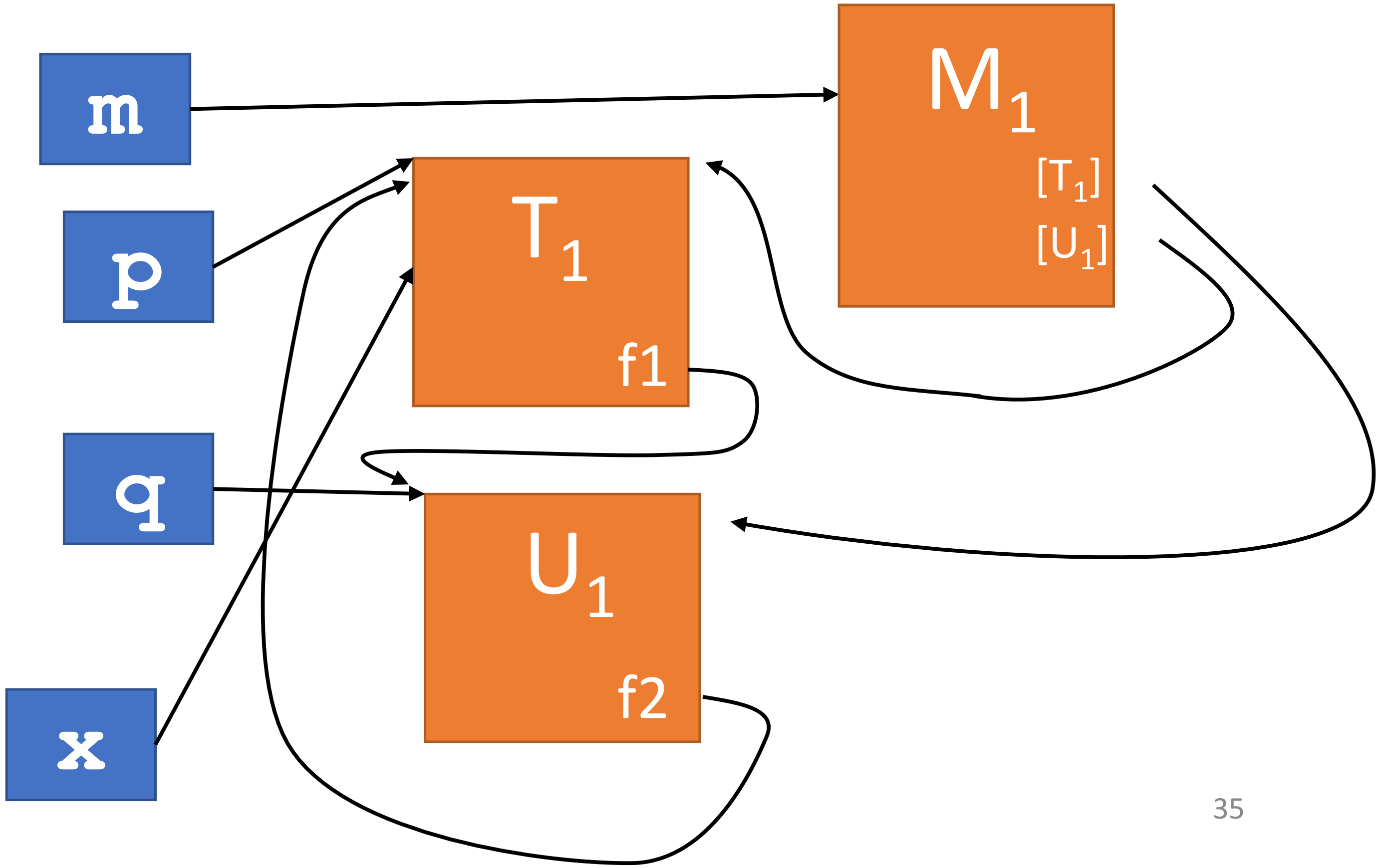
```
map   := mapnew
value := mapget map, key
         mapput map, key, value
value := mapremove map, key
flag  := mapprobe map, key
len   := maplength map

key   := equivclass userkey
```

```
it    := mapkeyiter map
flag  := iterhasnext it
value := iternext it
```

Points-to Analysis of Maps

- **Key Idea:** provide key-value maps as new language-level object type
  - On the same level as arrays, or heap objects with fields

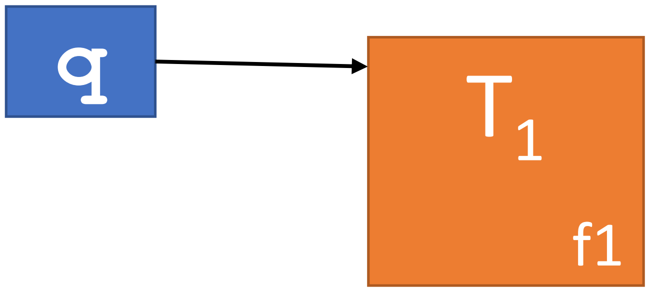
```
p = new T();
q = new U();
p.f1 = q;
q.f2 = p;
x = p.f1.f2;
→ m = mapnew;
→ mapput m, p, q;
→ mapput m, q, p;
```



Points-to Analysis of Maps

- **Key Idea:** provide key-value maps as new language-level object type
  - On the same level as arrays, or heap objects with fields

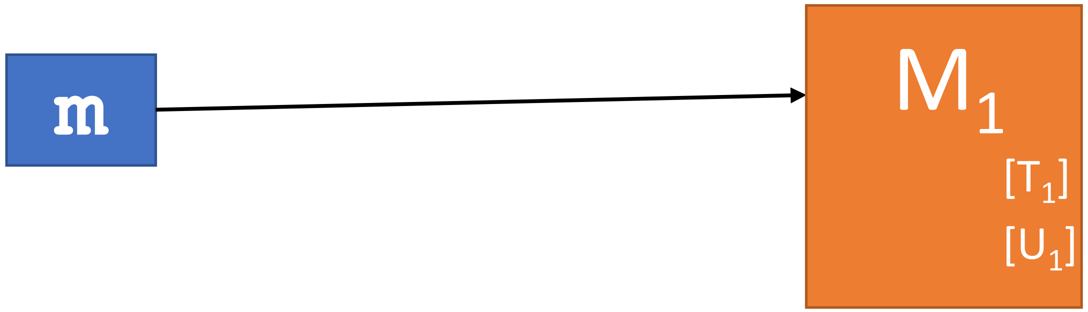
Ordinary heap abstraction



$pts(q) = \{ T_1 \}$

$pts(T_1.f) = \{ \dots \}$

Map heap abstraction



$pts(m) = \{ M_1 \}$

$pts(M_1.T_1) = \{ \dots \}$

$pts(M_1.U_1) = \{ \dots \}$

Points-to Analysis of Maps: Inference Rules

- **Key Idea:** provide key-value maps as new language-level object type
  - On the same level as arrays, or heap objects with fields

```
// Store
FieldPointsTo(obj, field, pointee) :-
  Store(ptr, field, value),
  VarPointsTo(ptr, obj),
  VarPointsTo(value, pointee).

// Load
VarPointsTo(dest, pointee) :-
  Load(dest, ptr, field),
  VarPointsTo(ptr, obj),
  FieldPointsTo(obj, field, pointee).
```

Points-to Analysis of Maps: Inference Rules

- **Key Idea:** provide key-value maps as new language-level object type
  - On the same level as arrays, or heap objects with fields

```
// Map Store
MapPointsTo(mapobj, keyobj, pointee) :-
  MapStore(map, key, value),
  VarPointsTo(map, mapobj),
  VarPointsTo(key, keyobj),
  VarPointsTo(value, pointee).

// Map Load
VarPointsTo(dest, pointee) :-
  MapLoad(dest, map, key),
  VarPointsTo(map, mapobj),
  VarPointsTo(key, keyobj),
  MapPointsTo(mapobj, keyobj, pointee).
```

Points-to Analysis of Maps: Lists

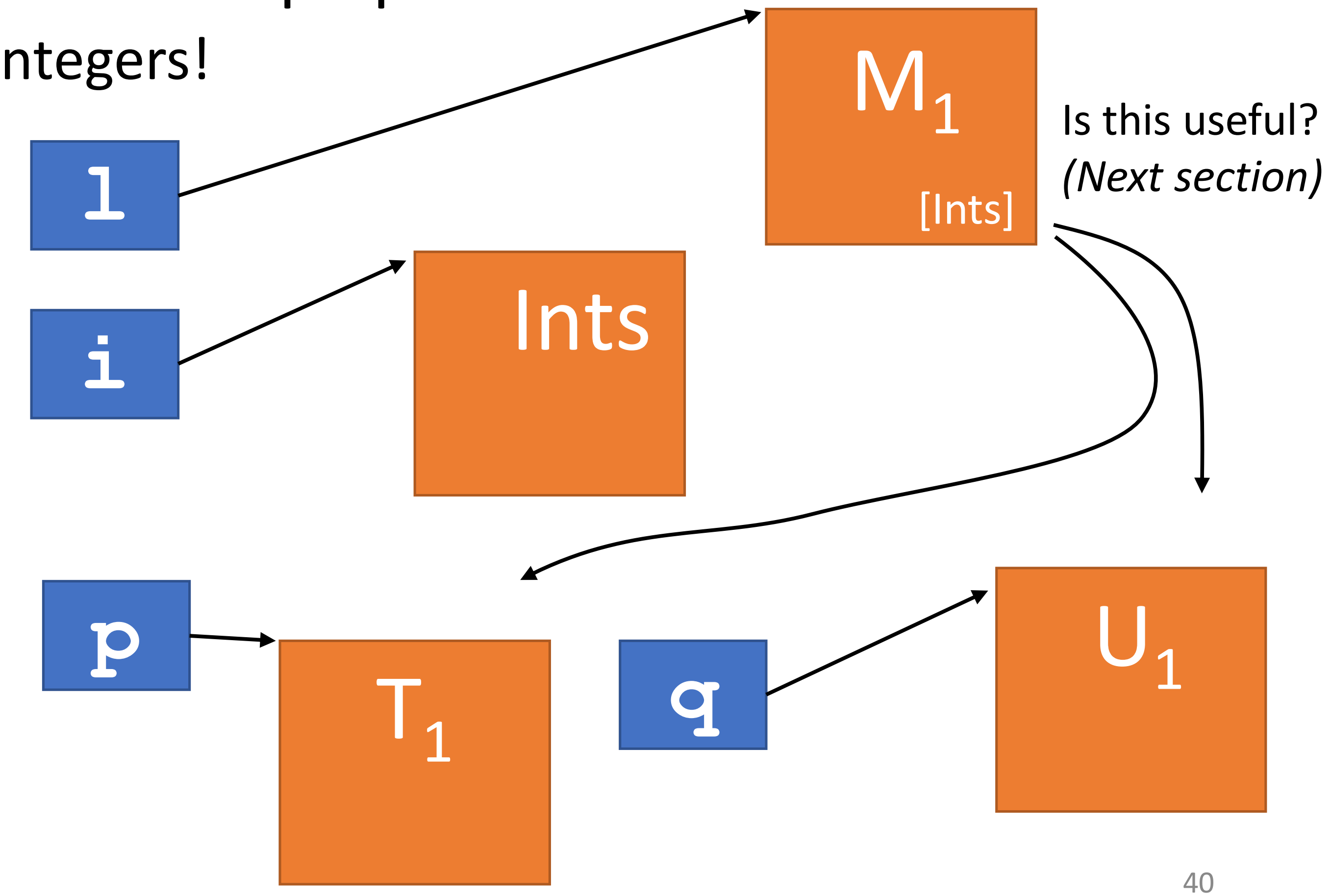
- **Key Idea:** provide lists (sequences) as new language-level object type
- **In analysis**: *lower* list operations to map operations
  - A list is just a map indexed by integers!

```
p = new T();
q = new U();
l = listnew;
i = 0;
listput l, i, p;
i = i + 1;
listput l, i, q;
```

Points-to Analysis of Maps: Lists

- **Key Idea:** provide lists (sequences) as new language-level object type
- **In analysis**: *lower* list operations to map operations
  - A list is just a map indexed by integers!

```
p = new T();
q = new U();
l = mapnew;
i = 0;
mapput l, i, p;
i = i + 1;
mapput l, i, q;
```



Points-to Analysis of Maps: Lists

- **Problem:** Not all list operations are explicitly indexed
- **Idea:** provide a *primitive* for *"some unique index"*

```
p = new T();
q = new U();
l = listnew;
listappend l, p;
listappend l, q;
```

→

```
p = new T();
q = new U();
l = mapnew;
idx1 = virtualindex l;
mapput l, idx1, p;
idx2 = virtualindex l;
mapput l, idx2, q;
```

Outline

- Introduction
- First-Class Data Structures
- DAEDALUS: Distinctness Analysis
- ICARUS: Incorporating Dynamic Checks

Can We Parallelize This Program?

- Standard parallelizability analyses understand arrays with *affine indexing functions*

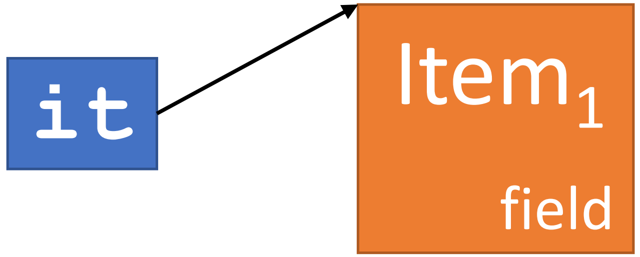
Loop Parallelizability

```
for (item…)t{: item) {
  array[2]i= 2*i;T();
} array[2*i+1] = i;
}
```

- Problem: no closed-form expression for *which* n
- Use alias analysis?

Alias Analysis for Loop Parallelization?

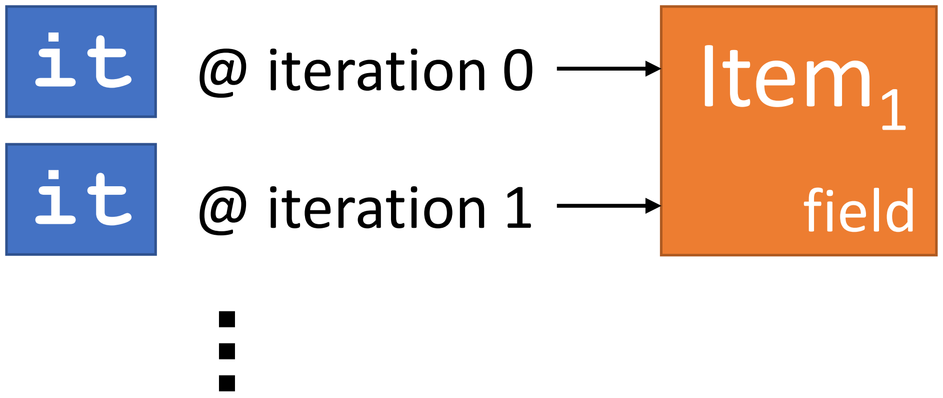
```
for (Item it : items) {
  it.field = new T();
}
```



Alias Analysis for Loop Parallelization?

```
for (Item it : items) {
  it.field = new T();
}
```

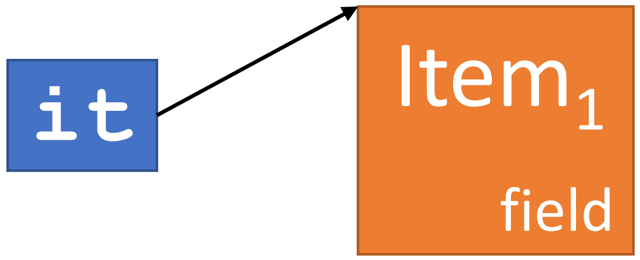
→ *Every iteration writes $Item_1$.field*



Alias Analysis for Loop Parallelization?

- What if we know that **`it`** points to a different object each iteration?

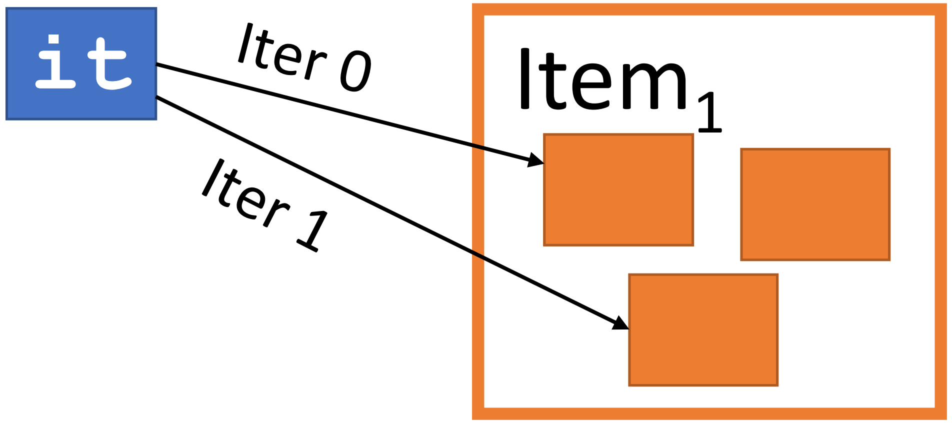
```
for (Item it : items) {
  it.field = new T();
}
```



Alias Analysis for Loop Parallelization?

- What if we know that **`it`** points to a different object each iteration?

```
for (Item it : items) {
  it.field = new T();
}
```



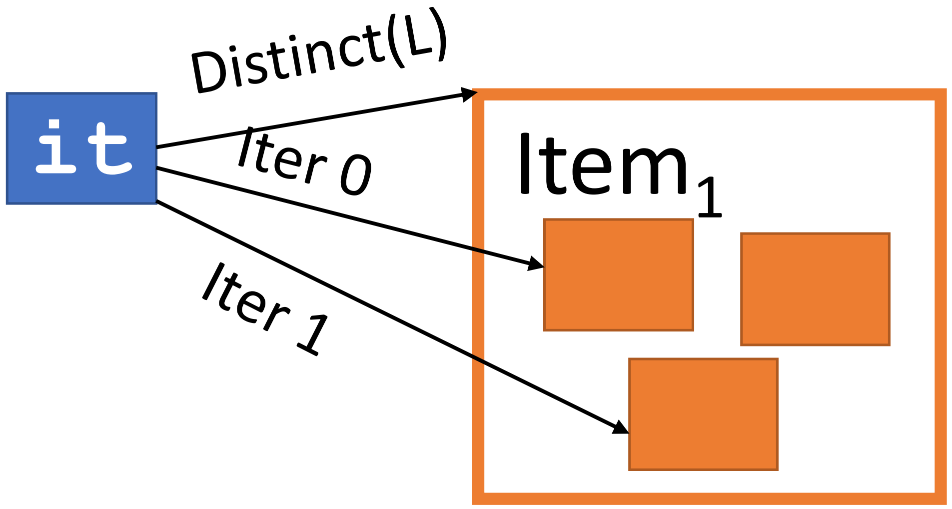
Distinctness Analysis: Variable Distinctness

- **Key Idea**: *annotate* points-to edges to indicate additional non-aliasing
  - A variable is *distinct with respect to a loop* if its value in iteration *i* does not alias its value in iteration *j*, within a single loop instance

```
L:  for (…) {
      it = …;
    }
```

instance
- it @ iter 0 → x_1
- it @ iter 1 → x_2
- …

instance
- it @ iter 0 → x_2
- it @ iter 1 → x_1
- …



Distinctness on the Heap?

- Many programs preserve distinctness *through the heap*

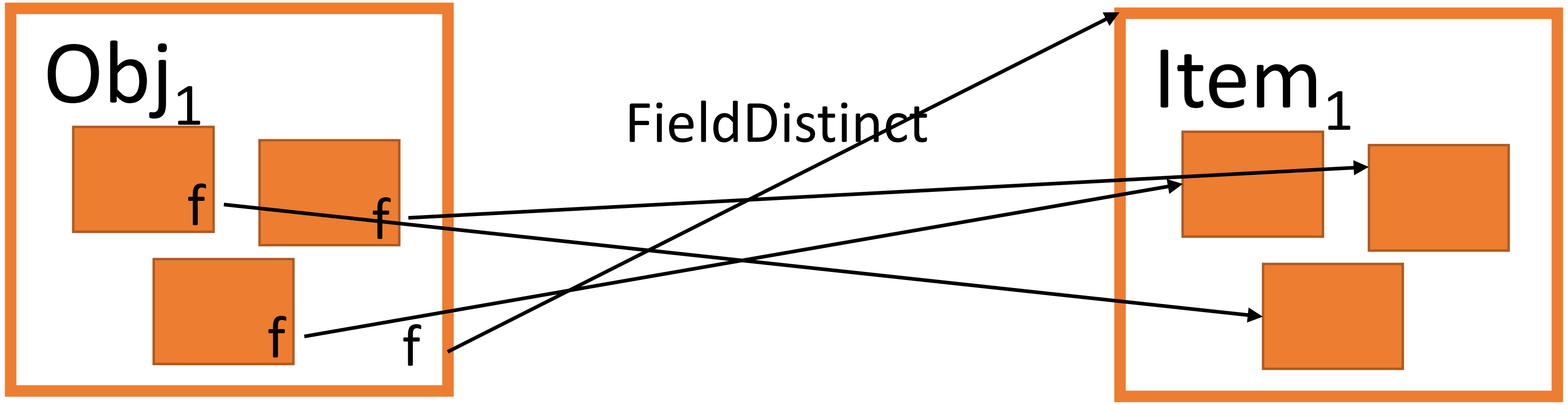
```
for (…) {
  parent = new Parent();
  parent.field = new Child();
  list.add(parent);
}

for (Parent p : list) {
  f(p.field);
}
```

Distinct

Distinct?

Distinctness Analysis: Heap Distinctness

- **Key Idea**: *annotate* points-to edges to indicate additional non-aliasing
  - A *field* on a heap abstraction is distinct if, for each object instance in this abstraction, the field has a different pointer value.



- Similarly for **map distinctness** (handles lists too)

Inferring Heap-Field Distinctness

- A field on a heap abstraction is *distinct* if:
  - For every loop around the one store statement to the field,
    - The stored value is distinct w.r.t. this loop, OR

time ↓

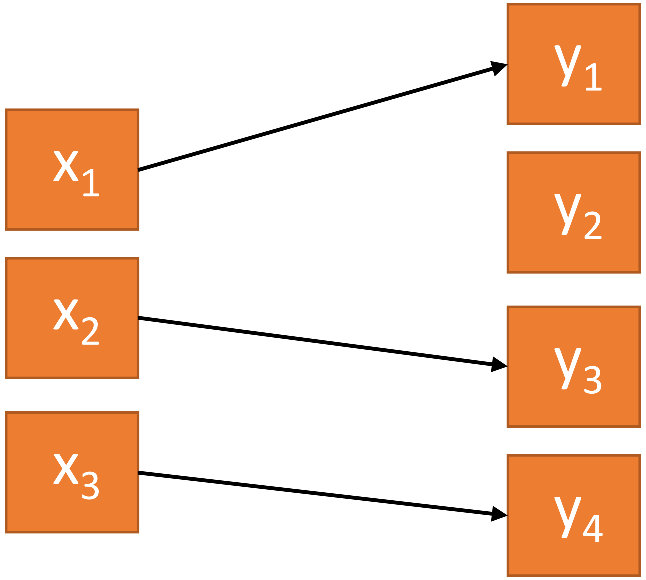
Store instances:

$x_1.f = y_1$

$x_2.f = y_2$

$x_2.f = y_3$

$x_3.f = y_4$



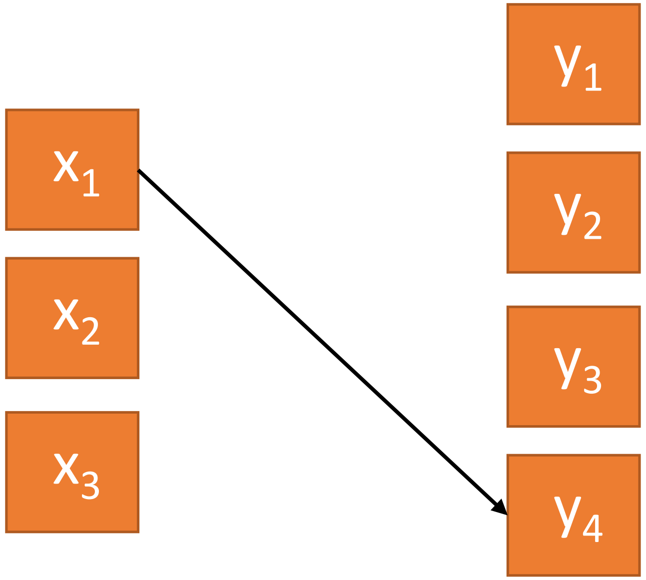
Inferring Heap-Field Distinctness

- A field on a heap abstraction is *distinct* if:
  - For every loop around the one store statement to the field,
    - The stored value is distinct w.r.t. this loop, OR
    - The stored-to pointer is constant w.r.t. this loop.

Store instances:

time

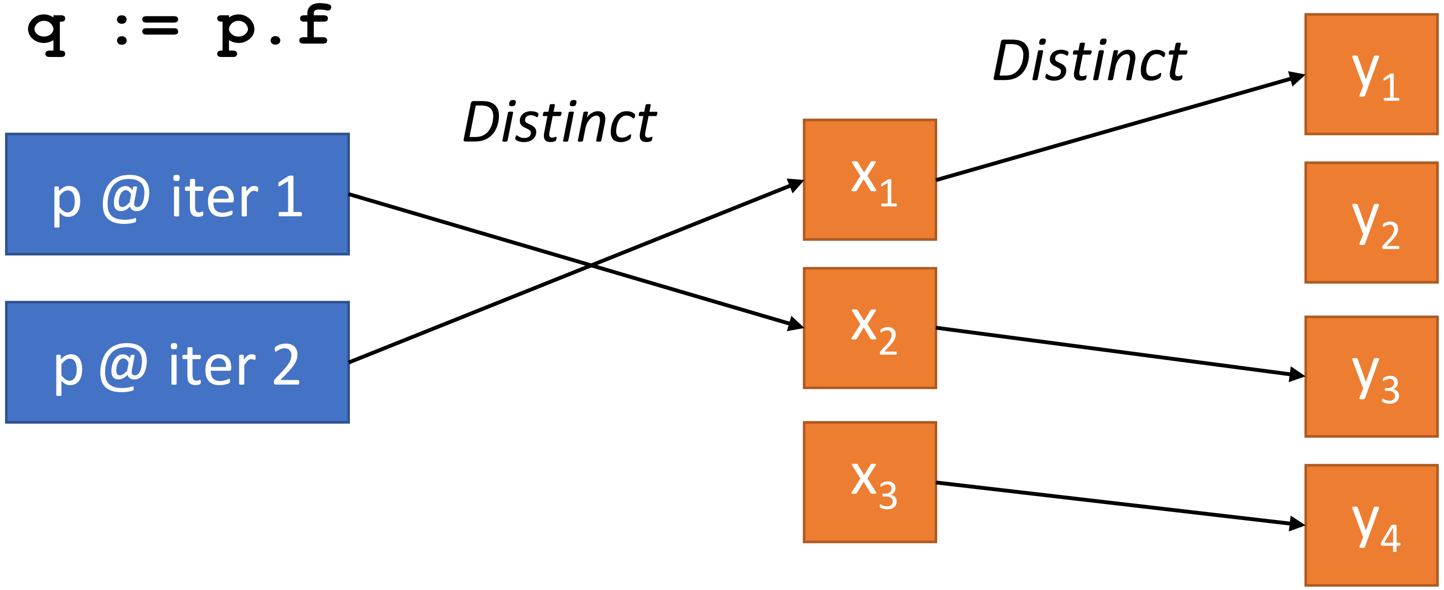
$x_1.f = y_1$
$x_1.f = y_2$
$x_1.f = y_1$
$x_1.f = y_4$



Using Heap-Field Distinctness

- A **load** result is distinct w.r.t. a loop if:
  - The loaded-from pointer is distinct w.r.t. this loop, AND
  - The heap field on all loaded-from abstractions are distinct, AND
  - No two loaded-from abstractions have intersecting points-to sets.

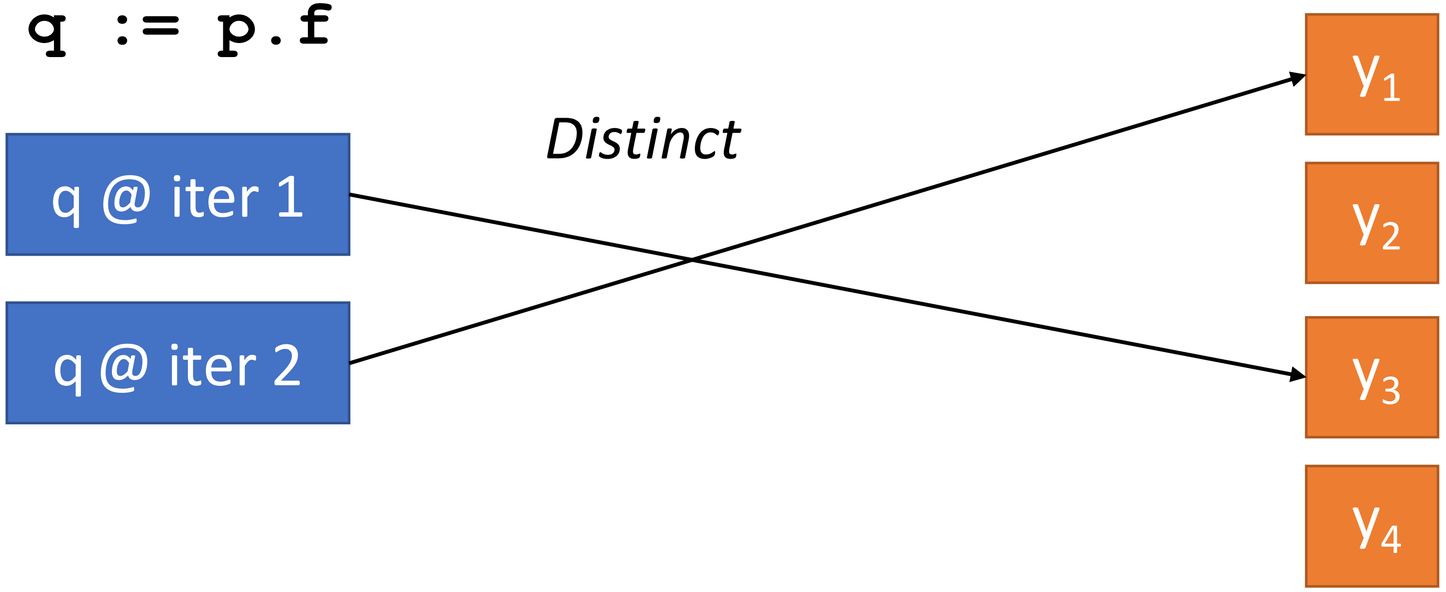
```
q := p.f
```



Using Heap-Field Distinctness

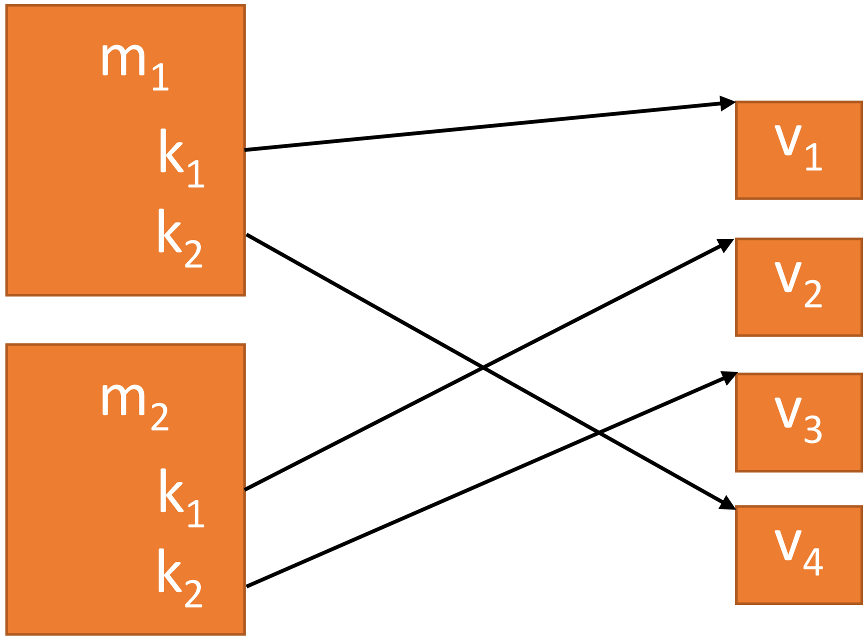
- A **load** result is distinct w.r.t. a loop if:
  - The loaded-from pointer is distinct w.r.t. this loop, AND
  - The heap field on all loaded-from abstractions are distinct, AND
  - No two loaded-from abstractions have intersecting points-to sets.

```
q := p.f
```



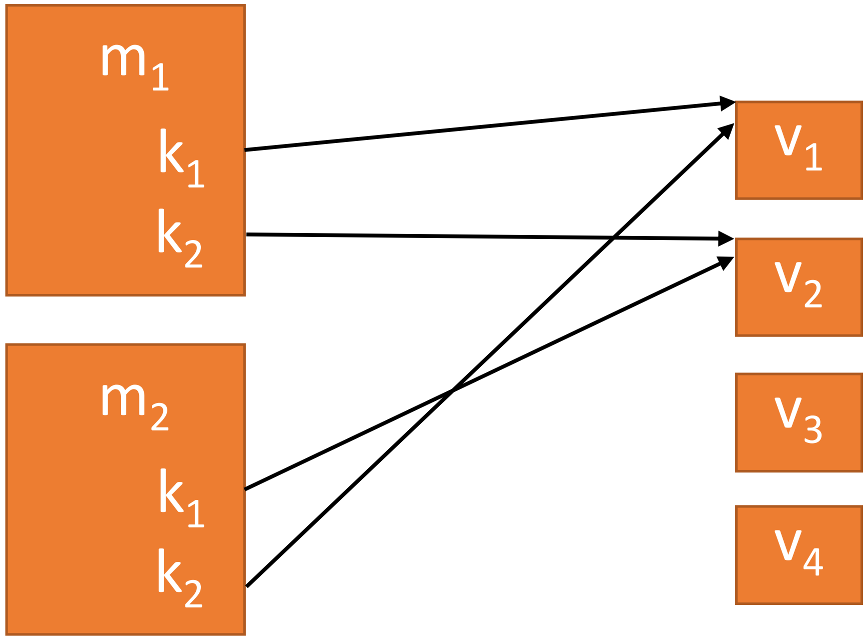
Distinctness Analysis: Map Distinctness

- *Key-Value Maps* have **two** possible types of distinctness for a given (Map, Key, Value) 3-tuple of abstractions:
  - *Global map distinctness*: no two keys in *any* two maps point to same value



Distinctness Analysis: Map Distinctness

- *Key-Value Maps* have **two** possible types of distinctness for a given (Map, Key, Value) 3-tuple of abstractions:
  - *Global map distinctness*: no two keys in *any* two maps point to same value
  - *Within-map distinctness*: no two keys in a *single* map point to same value



Distinctness Analysis in Detail: Assignment

- We actually compute **NotDistinct** as an analysis result
  - Meet-function at phi-nodes is *intersection* – thus, natural implementation

```
NotDistinct(var, loop) :-
  Assign(instruction, var, from),
  NotDistinct(from, loop),
  LoopInContext(instruction, loop).

NotConstant(var, loop) :-
  Assign(instruction, var, from),
  NotConstant(from, loop),
  LoopInContext(instruction, loop).
```

Distinctness Analysis in Detail: Load + Store

- We can derive the inverted (not-distinct) forms from the more intuitive positive-polarity versions with help of DeMorgan's Law:
  - A field is *not distinct* if (i) more than one store writes to it, or (ii) for any store, for any loop in context, stored value is not-distinct *and* pointer is not-constant

```
FieldNotDistinct(obj, field) :-
  Store(instruction1, ptr1, value),
  VarPointsTo(ptr1, obj),
  Store(instruction2, ptr2, value),
  VarPointsTo(ptr2, obj),
  instruction1 != instruction2.
FieldNotDistinct(obj, field) :-
  Store(instruction, ptr, value),
  LoopInContext(instruction, loop),
  VarNotDistinct(value, loop),
  VarNotConstant(ptr, loop).
```

Distinctness Analysis in Detail: Load + Store

- We can derive the inverted (not-distinct) forms from the more intuitive positive-polarity versions with help of DeMorgan's Law:
  - A load result is *not distinct* if (i) it reads from abstractions with overlapping field points-to sets, or (ii) the field is not-distinct on any pointed-to abstraction, or (iii) the pointer is not-distinct.

```
VarNotDistinct(dest, loop) :-
  Load(inst, ptr, field, dest),
  VarPointsTo(ptr, obj),
  FieldNotDistinct(obj, field),
  LoopInContext(inst, loop).
VarNotDistinct(dest, loop) :-
  Load(inst, ptr, field, dest),
  VarNotDistinct(ptr, loop).
```

Distinctness Analysis in Detail: Map Store

```
MapNotDistinct(mapobj, keyobj), MapNotDistinctWithinMap(mapobj, keyobj) :-
  MapStore(inst1, map1, key1, dest1),
  VarPointsTo(map1, mapobj),
  VarPointsTo(key1, keyobj),
  MapStore(inst2, map2, key2, dest2),
  VarPointsTo(map2, mapobj),
  VarPointsTo(key2, keyobj),
  inst1 != inst2.

MapNotDistinct(mapobj, keyobj) :-
  MapStore(inst, map, key, dest),
  VarPointsTo(map, mapobj),
  VarPointsTo(key, keyobj),
  VarNotDistinct(dest, loop),
  (VarNotConstant(map, loop); VarNotConstant(key, loop)).

MapNotDistinctWithinMap(mapobj, keyobj) :-
  MapStore(inst, map, key, dest),
  VarPointsTo(map, mapobj),
  VarPointsTo(key, keyobj),
  VarNotDistinct(dest, loop),
  VarNotConstant(key, loop).
```

Distinctness Analysis in Detail: Map Load

```
VarNotDistinct(dest, loop) :-
  MapLoad(inst, map, key, dest),
  VarPointsTo(map, mapobj),
  VarPointsTo(key, keyobj),
  MapNotDistinct(mapobj, keyobj),
  MapNotDistinctWithinMap(mapobj, keyobj),
  LoopInContext(inst, loop).
VarNotDistinct(dest, loop) :-
  MapLoad(inst, map, key, dest),
  VarPointsTo(map, mapobj),
  VarPointsTo(key, keyobj),
  // may still be distinct within map
  MapNotDistinct(mapobj, keyobj),
  (VarNotConstant(map, loop); VarNotDistinct(key, loop)).
VarNotDistinct(dest, loop) :-
  MapLoad(inst, map, key, dest),
  VarNotDistinct(map, loop),
  VarNotDistinct(key, loop).
```

Example: Distinctness in Action

```
for (int i = 0; i < 100; i++)
  list.add(i);
for (Integer i : list)
  map.put(i, new Parent());
for (Integer i : map.keyset())
  map.get(i).childPtr = new Child();
```

- `for (int i = 0; i < 100; i++)` → Integer induction variable distinct
- `list.add(i);` → List elements are distinct
- `for (Integer i : list)` / `new Parent()` → Parent instance is distinct
- `map.put(i, new Parent());` → Map values are globally distinct
- `for (Integer i : map.keyset())` → **i** is distinct (map key iter value)
- `map.get(i).childPtr = new Child();` → **map.get(i)** is distinct
- `new Child()` → **Child** instance is distinct
- `childPtr` → childPtr is field-distinct

```
for (Integer i : list)
  map.get(i).childPtr.field = i;
```

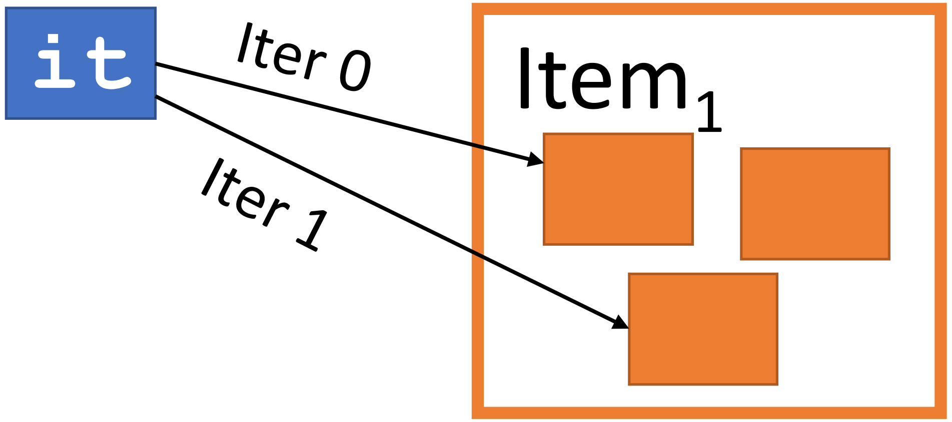
- `for (Integer i : list)` → **i** is distinct (from list)
- `map.get(i).childPtr.field = i;` → **map.get(i)** is distinct
- `childPtr` → **map.get(i).childPtr** is distinct
- → Store to **field** is parallelizable

Side-Effect Analysis for Parallelization

- When can we parallelize a loop L?

L:
```
for (Item it : items) {
  it.field = new T();
}
```

Distinct w.r.t. L

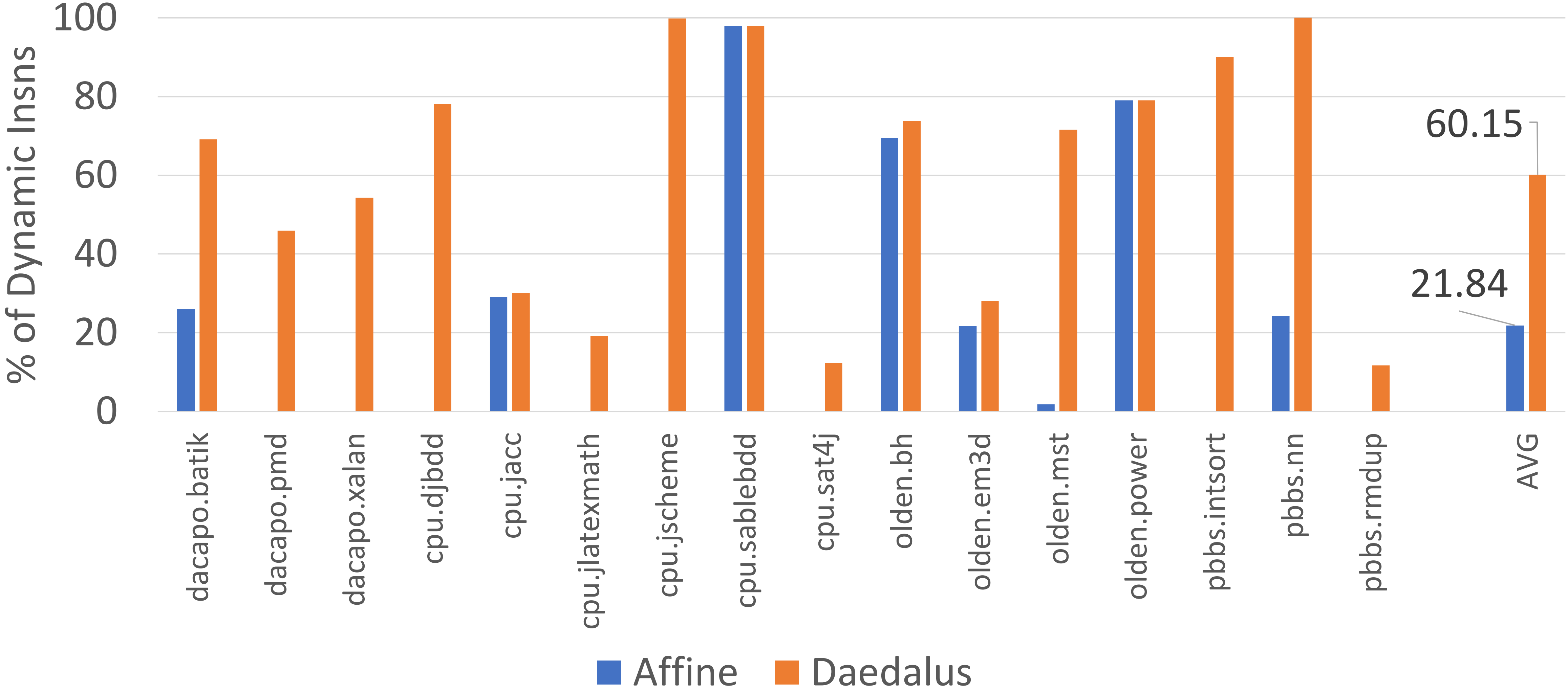


- For each written-to location (abstraction.field or map[key]):
  - Every written-to pointer to this location is *distinct* w.r.t. L
  - All of the written-to pointers (if > 1) alias each other (*same* distinct object)
- See thesis for: must-alias analysis; map/list side-effects + commutativity; locking

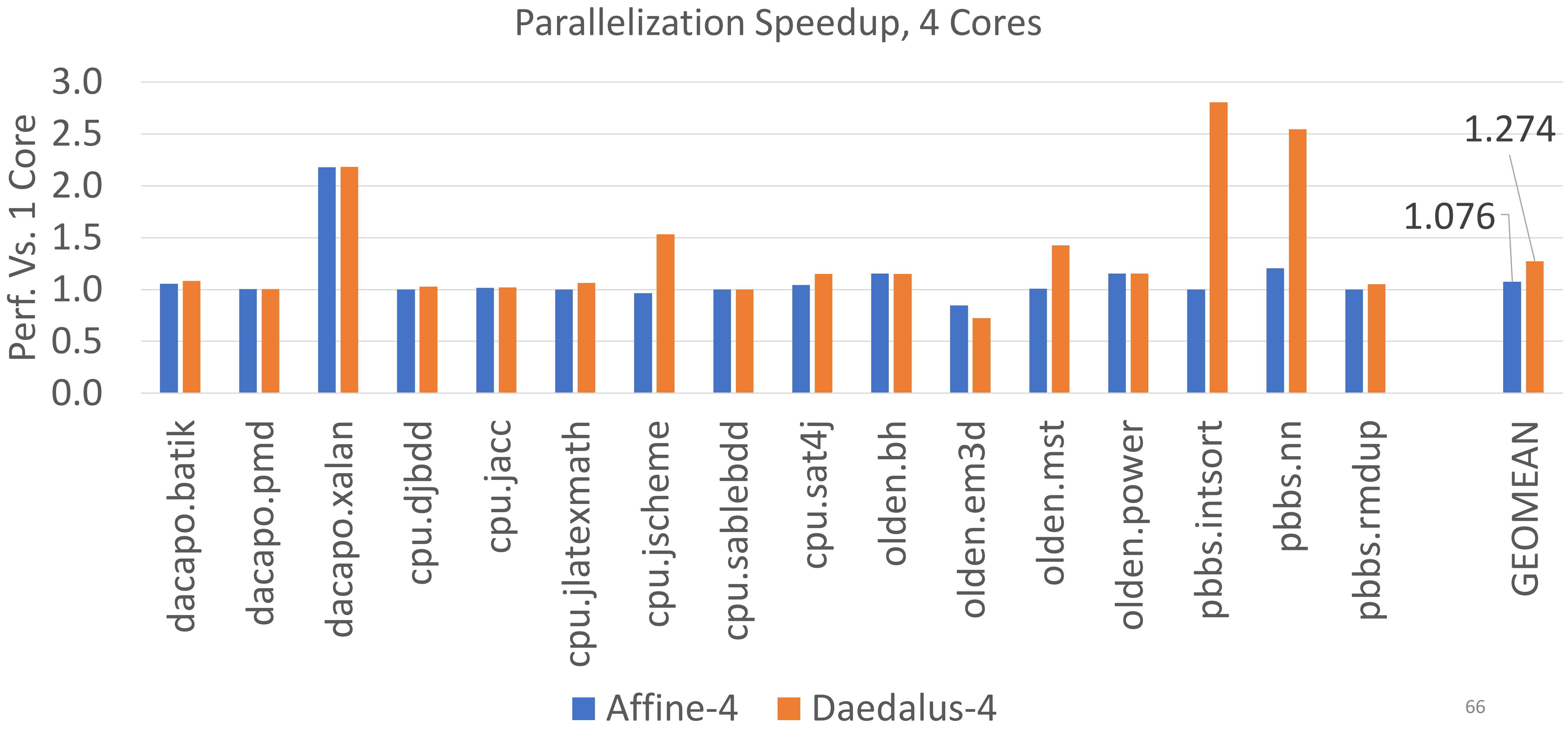
Evaluation: Methodology

- Analyses
  - Our system: DAEDALUS (Data-structure-aware Distinctness Analysis)
  - Baseline: standard array-based parallelization analysis
- Java Benchmark suites
  - dacapo: Well-known benchmark suite of full programs
  - olden: Small data-structure-intensive programs
  - pbbs: Problem-Based Benchmark Suite
  - cpu: “CPU-intensive” programs – compilers, simulators, …
- Simulation-based performance results

Evaluation: Parallelization Coverage (High Opp.)



Evaluation: Parallel Speed-ups



Outline

- Introduction
- First-Class Data Structures
- DAEDALUS: Distinctness Analysis
- ICARUS: Incorporating Dynamic Checks

Is Static Analysis Enough?

- Consider the following snippet:

```
List<Item> l = …;
for (Item it : input) {
  if (!it.seen) {
    l.add(it);
    it.seen = true;
  }
}
for (Item it : l) { f(it); }
```

Deduplication logic (bracketing the `if (!it.seen) { … }` block)

→ *Parallelizable* (the second loop)

- Are `l`’s elements distinct?
- Could we parallelize the second loop?

Simple Dynamic Checks

- What if we check, then we parallelize *only if safe*, at runtime?

```
List<Item> l = …;
// …
if (distinct(l)) {
  l.parallelStream().forEach(it -> f(it));
} else {
  for (Item it : l) { f(it); }
}
```

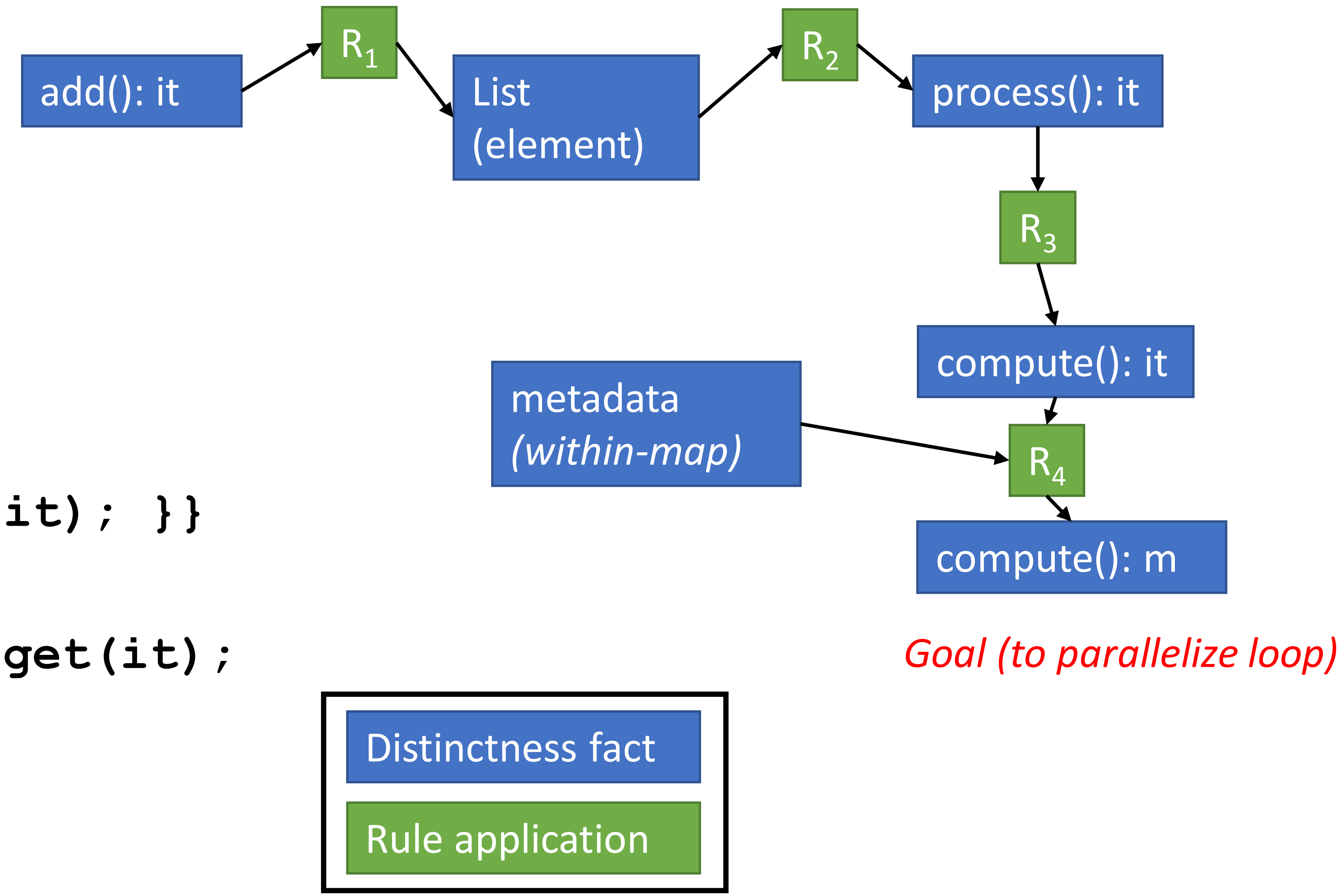
Systematically Leveraging Dynamic Checks

- **Goal**: insert minimal set of checks while maximizing parallelized loops

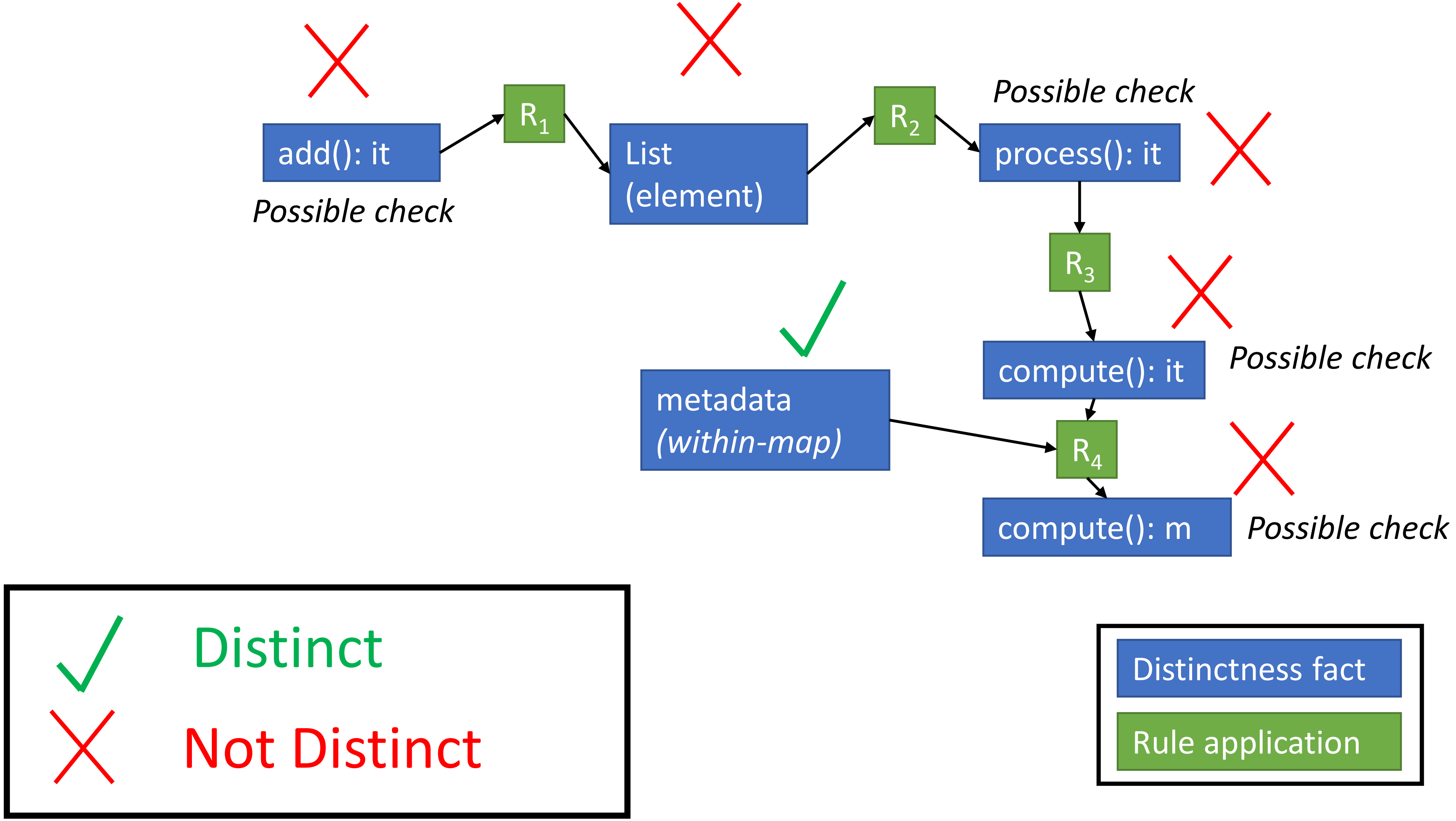
- **Key Idea**: extend static-analysis rules in a systematic way
  - *Step 1.* Compute *possible distinctness*
  - *Step 2*. Evaluate parallelization; choose actually-needed dynamic possibilities
  - *Step 3.* Propagate *needed distinctness* backward to choose check sites.

Systematically Leveraging Dynamic Checks

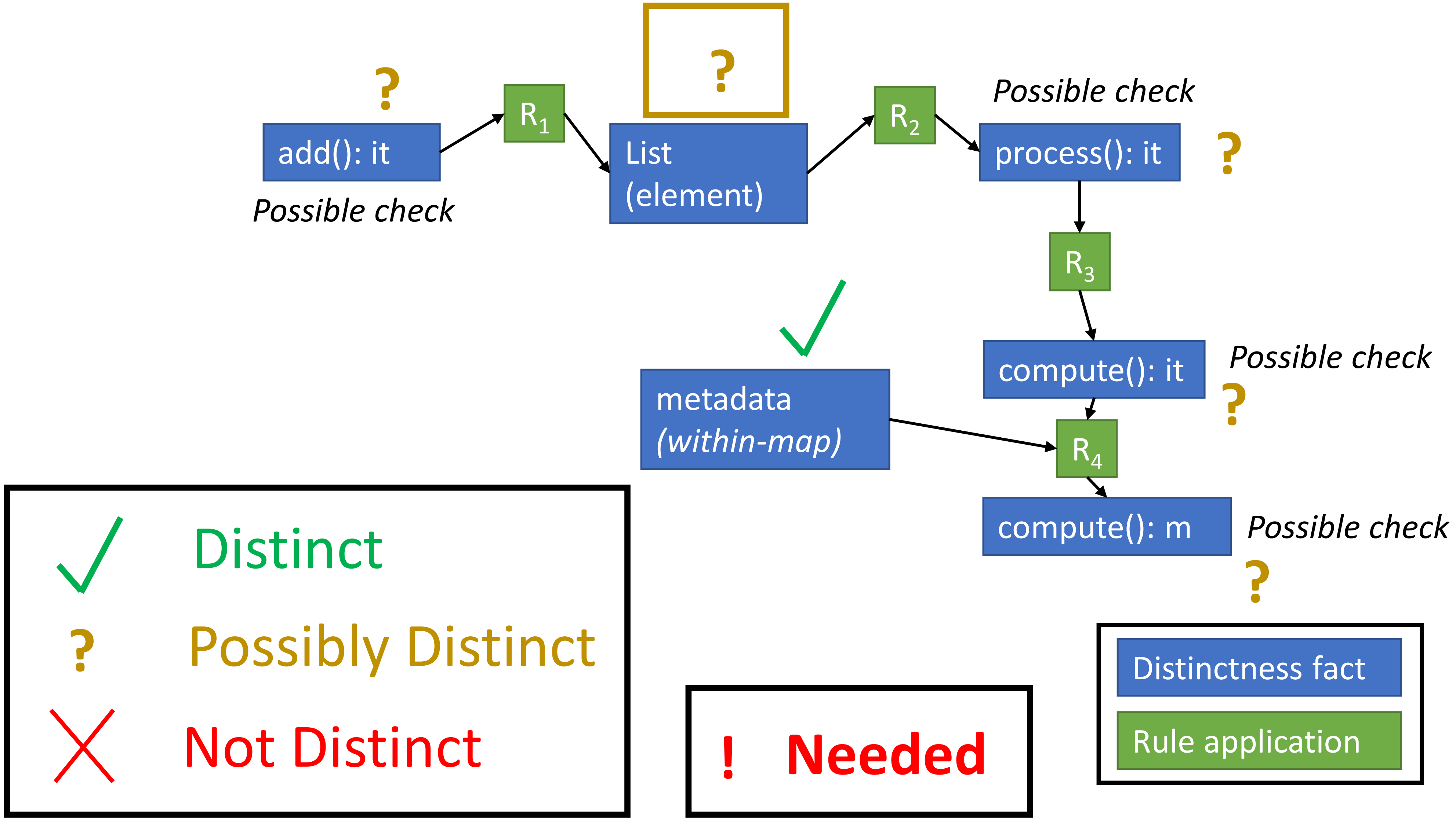
```
void add(Item it) {
  if (!it.seen) {
    list.add(it);
    it.seen = true; }}
void process() {
  for (Item it : list) {
    it.result = compute(it); }}
int compute(Item it) {
  Metadata m = metadata.get(it);
  m.update();
  return m.result(); }
```



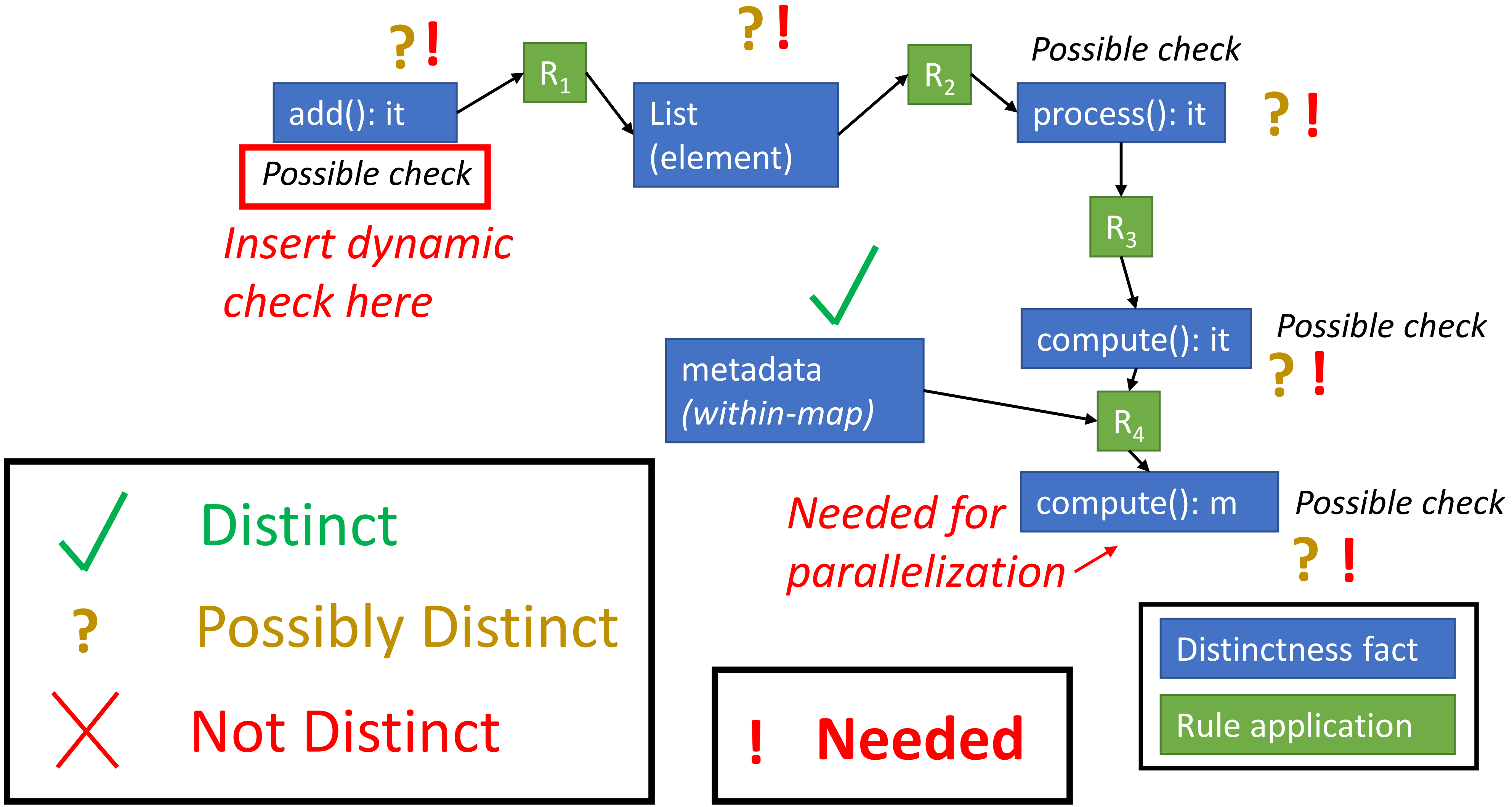
Systematically Leveraging Dynamic Checks



Systematically Leveraging Dynamic Checks



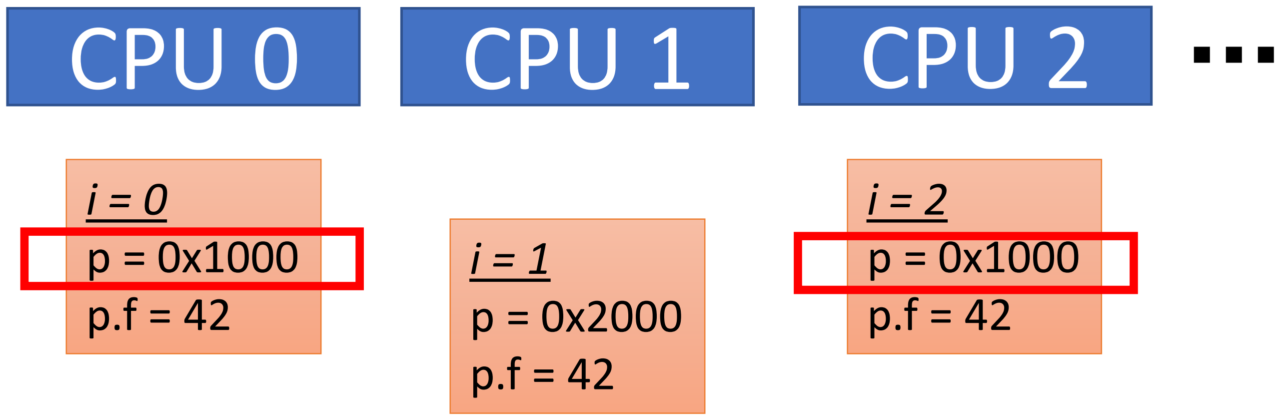
Systematically Leveraging Dynamic Checks



Executing with Dynamic Checks

- If checks always succeed, we're done!
- What if a check fails?

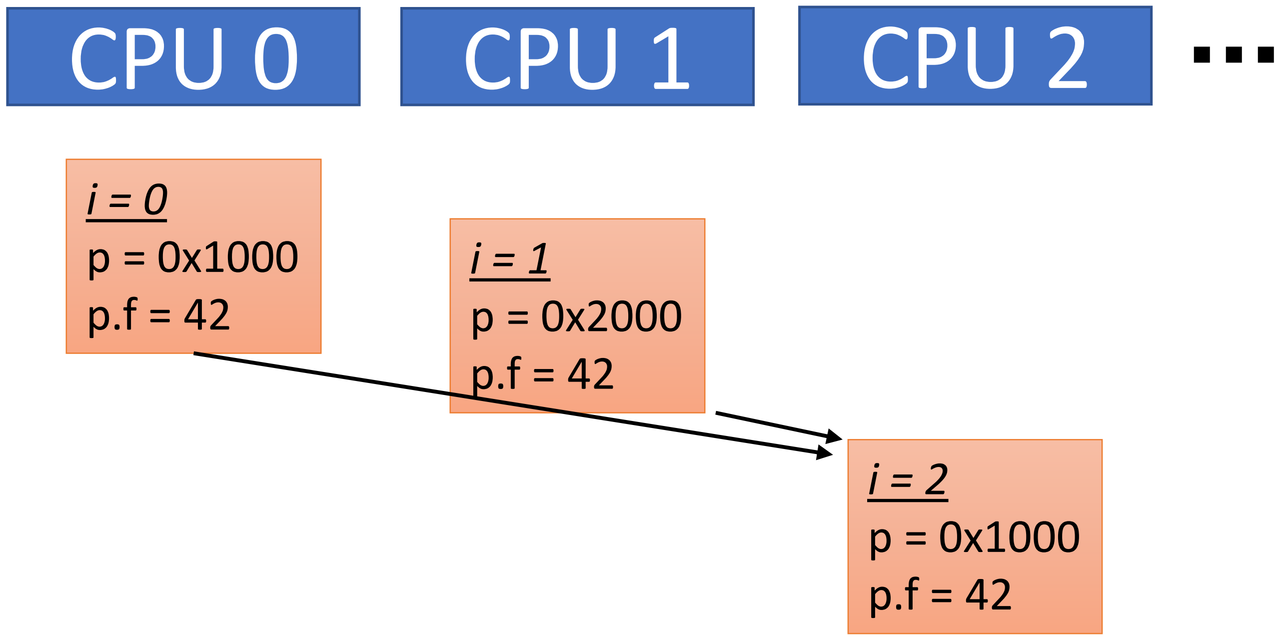
```
for (…) {
  p = …;
  p.f = …;
}
```



Executing with Dynamic Checks

- If checks always succeed, we're done!
- What if a check fails?

```
for (…) {
  p = …;
  p.f = …;
}
```



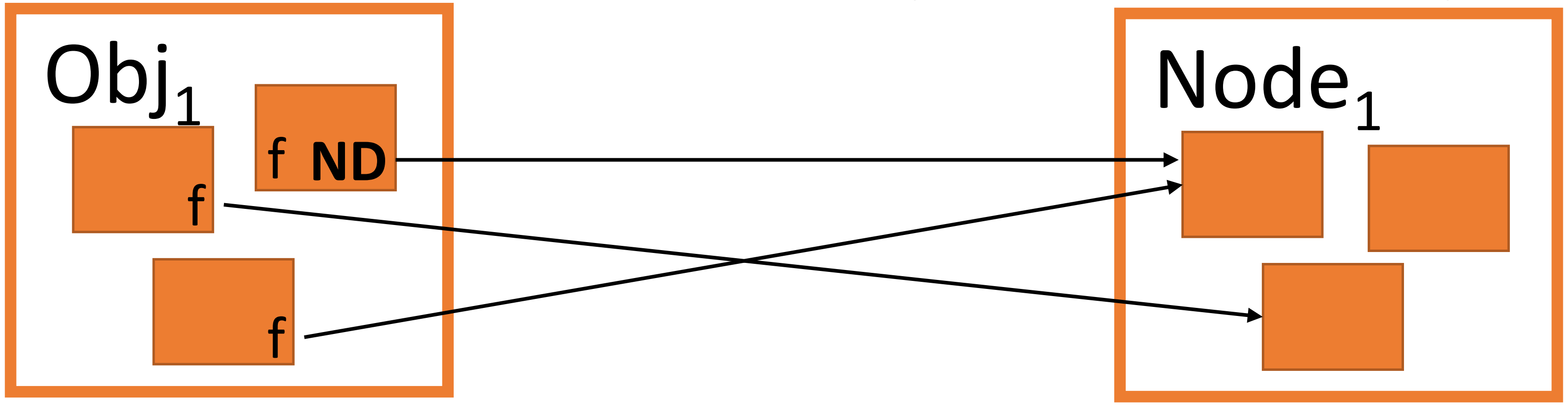
- **Key Idea**: pause at the check & wait for prior iters → no rollback!

Dynamic Heap-Distinctness Checks

- How do we dynamically check *field distinctness*?
  - Prohibitive to check directly: iterate over all objects on heap…?
- **Key Idea**: maintain a *non-distinct bit* on pointer fields with checks

Pointer word	Pointer (63 bits)	ND

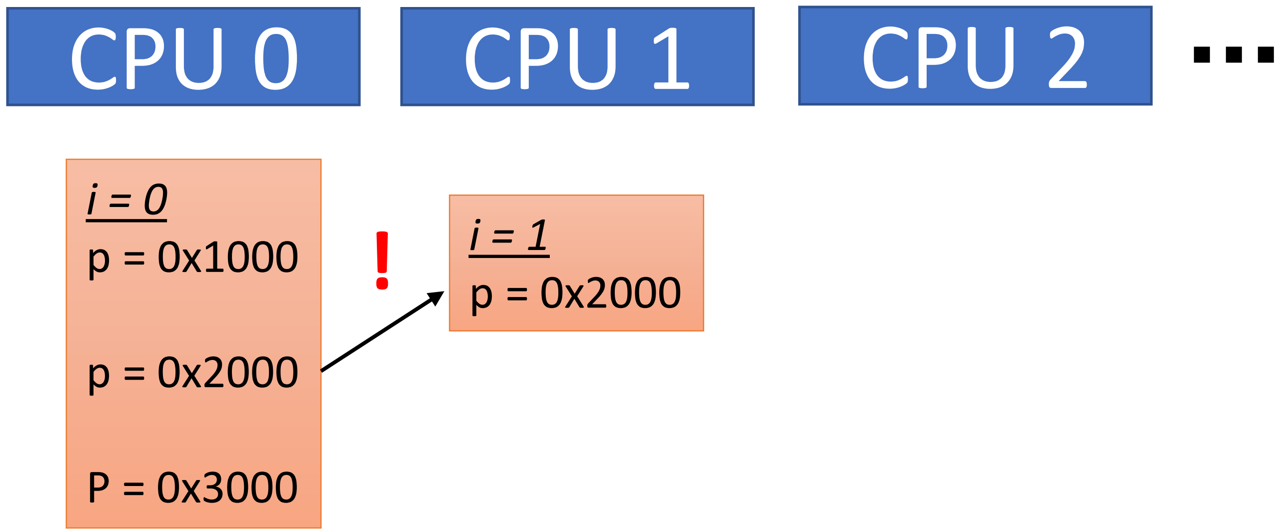
  - **Update** on store if containing loop has had a failed check
  - **Check** on load and serialize on failure (as for variable checks)



Sequencing the Checks

- How do we know a check has succeeded?
  - We must know *all addresses generated by this check in prior iterations*

```
for (…) {
  for (…) {
    p = …;
  }
}
```

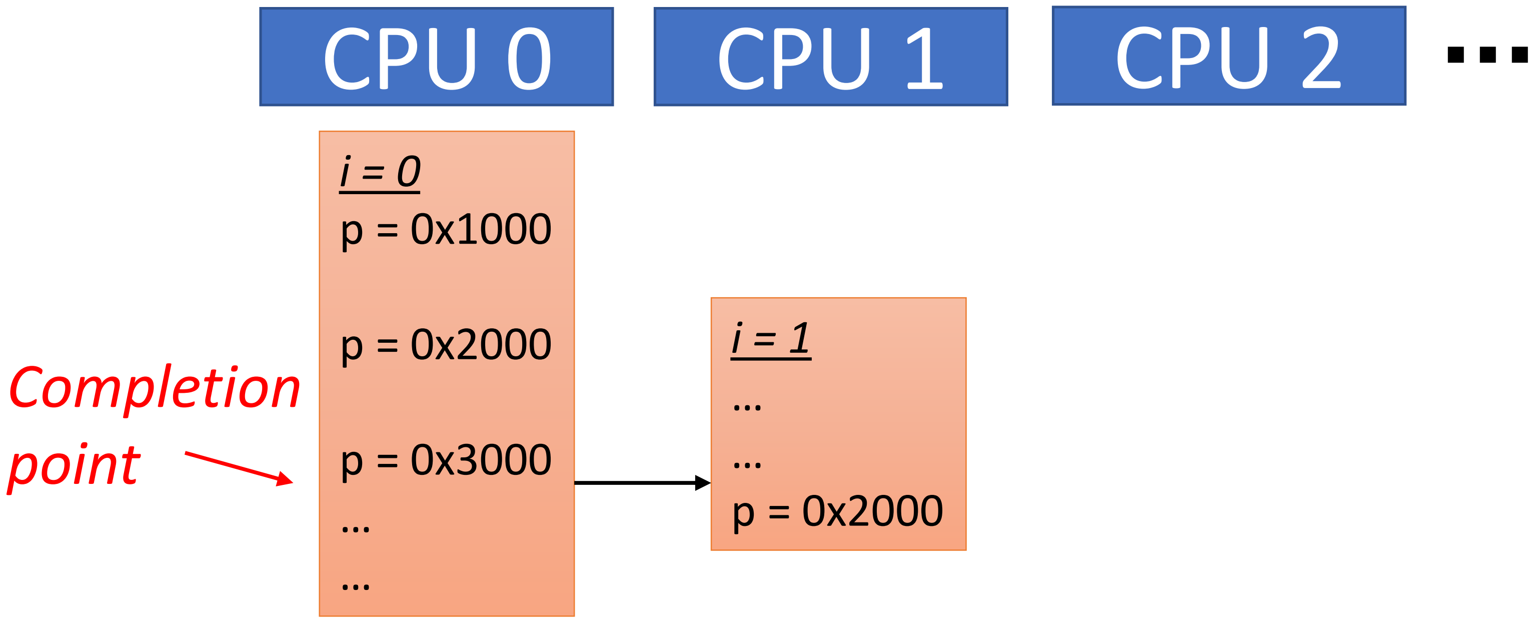


Sequencing the Checks

- How do we know a check has succeeded?
  - We must know *all addresses generated by this check in prior iterations*

```
for (…) {
  for (…) {
    p = …;
  }
}
```

Completion point → (closing brace of inner loop)



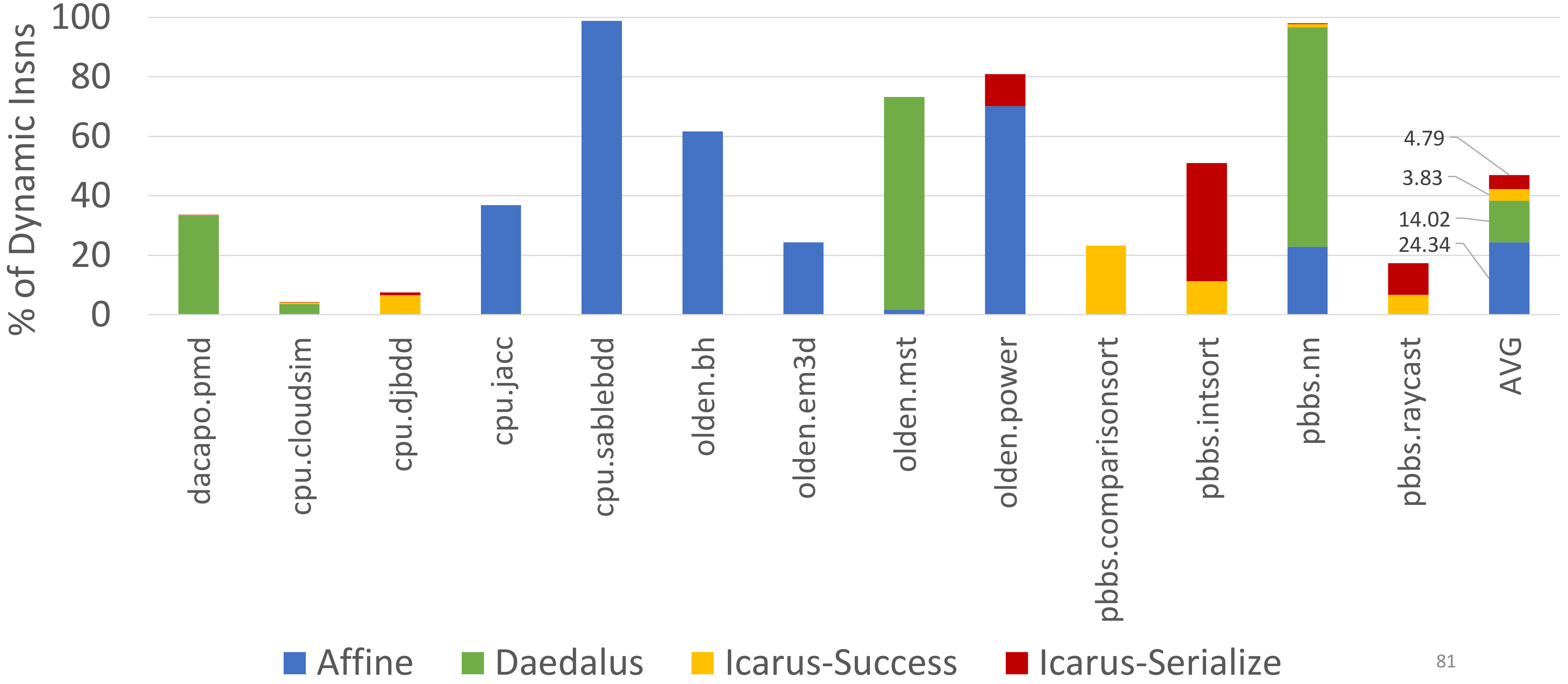
- **Key Idea**: Check waits for the "check completion point" of prior iteration

Evaluation: Methodology

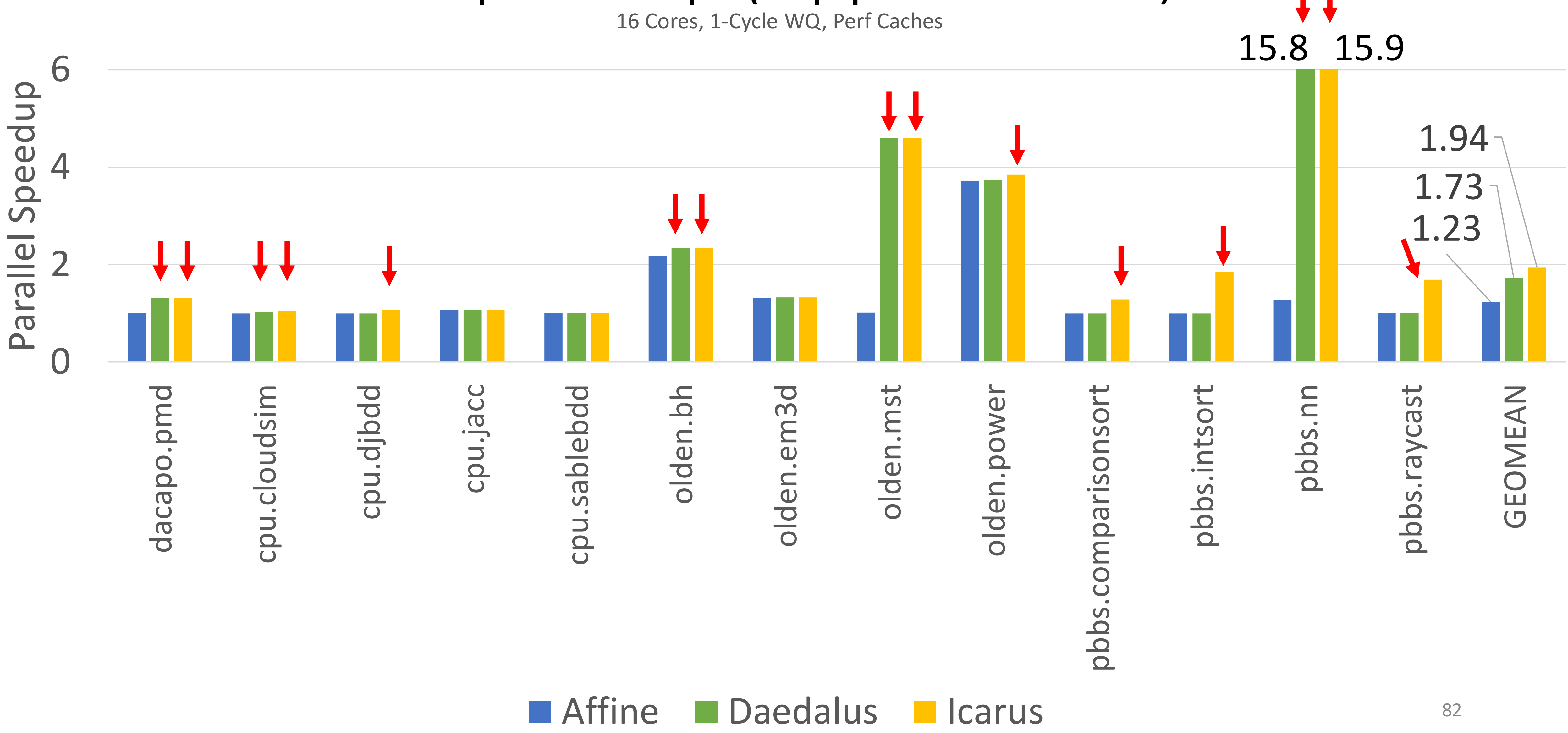
- Analyses
  - ICARUS (Integrated Compiler and Runtime with User-level Semantics)
  - DAEDALUS
  - Standard array-based baseline

- Simulation-based performance results
  - New traces w.r.t. DAEDALUS evaluation (to incorporate values for checks)

Evaluation: Parallelization Coverage



Evaluation: Speed-up (Upper Bound)



Evaluation: Discussion

- Significant *opportunity* with Daedalus, improved under Icarus
- Additional speedup will require:
  - Heuristics to choose the most appropriate loops to parallelize
  - Effective means of parallelizing small-iteration loops
- Our focus in this work was on *analysis*; backend engineering is very important, but separate work (with many interesting problems)

Summary

- *Data-structure-aware* analysis framework
  - First-class primitives for key-value maps and lists
- DAEDALUS: New loop-centric, simple alias analysis using *distinctness*
  - Analyzes cross-loop-iteration and on-heap pointer aliasing
- ICARUS: Hybrid dynamic-static analysis approach to improve precision
  - Systematic method of deriving hybrid analysis from static analysis rules
  - Execution techniques to enable loop parallelization with dynamic checks

Future Directions

- Additional IR primitives / built-ins
  - Can we build, e.g., a graph-aware analysis?
  - Primitives for queries/updates (dataflow) and *traversals* (control flow)
- Generalize the hybrid dynamic-static scheme
  - Where else can we make use of dynamic checks for better precision?
  - Need to think about execution strategy
- Apply to systems languages: C/C++
  - Can we apply the same ideas to a more complex heap model?
  - Pointers to inner data structures & pointer arithmetic; value types; …
- More scalable analysis
  - Can we build a distinctness-like analysis on top of a more scalable foundation?
  - Avoid e.g. blowup in contexts (with function summaries) or heap abstractions (with careful merging)
- Use parts of our infrastructure in your project!
  - Datalog is a *really productive* way to build static analyses

Thanks! Questions?