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:

```java
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**

- CPU 0
- Iteration 0
- Iteration 1
- Iteration 2
Problem: High-Level Program Optimizations are Difficult

Modern system: many CPUs

Legacy code is sequential: unused cores
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:

```java
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:

```java
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:

```java
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

CPU 0  CPU 1  CPU 2  

Iteration 2  Iteration 11
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 $k_1 \neq k_2$.

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: \( \text{map.put}(k, v); \)

  Compiler sees:

  ```java
  void put(K key, V value) {
    int h = \text{key.x} \times 8931 + \text{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*!

```java
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

```java
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

For-Each

new Item()

List<Item>

Elements: unique

For-Each

analyze()

Side-effects: none

Map<Item, Result>

Values: unique per-key

Parallelizable
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

```java
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 value) {
    int h = key.hash();
    Node n = new Node(key, value);
    n.next = slots[h];
    slots[h] = n;
}
Points-to Analysis of a Hash Map

```
HashMap₁
  [Item₁]
  Item₁
  Result₁

VS.

HashMap₁
  buckets
  Array₁
    array elem
  Node₁
    key value

```

m

it

r
Points-to Analysis of a Hash Map: Problems

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

```java
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

```java
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

```plaintext
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*

```java
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*

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

```java
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

```java
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

```java
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

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

it := mapkeyiter map
flag := iterhasnext it
value := internext it

key := equivclass userkey
```
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

```plaintext
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**

\[
\text{pts}(q) = \{ T_1 \} \\
\text{pts}(T_1.f) = \{ \ldots \}
\]

**Map heap abstraction**

\[
\text{pts}(m) = \{ M_1 \} \\
\text{pts}(M_1.T_1) = \{ \ldots \} \\
\text{pts}(M_1.U_1) = \{ \ldots \}
\]
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),
  MapPointsTo(map, mapobj),
  MapPointsTo(key, keyobj),
  MapPointsTo(value, pointee).

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

```java
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!

```java
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”

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

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

```
for (i = ...; i < n) {
    array[2*i] = i;
    array[2*i+1] = i;
}
```

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

---

**Loop Parallelizability**

- Problem: no closed-form expression for *which n*
- Use alias analysis?
for (Item it : items) {
    it.field = new T();
}

Alias Analysis for Loop Parallelization?
Alias Analysis for Loop Parallelization?

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

→ Every iteration writes Item₁.field
Alias Analysis for Loop Parallelization?

• What if we know that it points to a different object each iteration?

```java
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?

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

```plaintext
for (...) {
    it = ...;
}
```
Distinctness on the Heap?

• Many programs preserve distinctness through the heap

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

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

Store instances:

\[
\begin{align*}
x_1.f &= y_1 \\
x_2.f &= y_2 \\
x_3.f &= y_3 \\
x_3.f &= y_4
\end{align*}
\]
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:*

\[
\begin{align*}
x_1.f &= y_1 \\
x_1.f &= y_2 \\
x_1.f &= y_1 \\
x_1.f &= y_4
\end{align*}
\]
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.

\[ q := p.f \]

\[ p @ \text{iter 1} \]
\[ p @ \text{iter 2} \]

\[ px_1 \]
\[ px_2 \]
\[ px_3 \]

\[ qy_1 \]
\[ qy_2 \]
\[ qy_3 \]
\[ qy_4 \]

Distinct
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.

\[ q := p.f \]

\[ q \text{ @ iter 1} \]
\[ q \text{ @ iter 2} \]

\[ y_1 \]
\[ y_2 \]
\[ y_3 \]
\[ y_4 \]

Distinct
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 \textbf{NotDistinct} as an analysis result
  - Meet-function at phi-nodes is \textit{intersection} – thus, natural implementation

\begin{verbatim}
NotDistinct(var, loop) :-
    Assign(instruction, var, from),
    NotDistinct(from, loop),
    LoopInContext(instruction, loop).
\end{verbatim}

\begin{verbatim}
NotConstant(var, loop) :-
    Assign(instruction, var, from),
    NotConstant(from, loop),
    LoopInContext(instruction, loop).
\end{verbatim}
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

\[
\text{FieldNotDistinct}(\text{obj}, \text{field}) \leftarrow \\
\text{Store}(\text{instruction}_1, \text{ptr}_1, \text{value}), \\
\text{VarPointsTo}(\text{ptr}_1, \text{obj}), \\
\text{Store}(\text{instruction}_2, \text{ptr}_2, \text{value}), \\
\text{VarPointsTo}(\text{ptr}_2, \text{obj}), \\
\text{instruction}_1 \neq \text{instruction}_2.
\]

\[
\text{FieldNotDistinct}(\text{obj}, \text{field}) \leftarrow \\
\text{Store}(\text{instruction}, \text{ptr}, \text{value}), \\
\text{LoopInContext}(\text{instruction}, \text{loop}), \\
\text{VarNotDistinct}(\text{value}, \text{loop}), \\
\text{VarNotConstant}(\text{ptr}, \text{loop}).
\]
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.

\[
\text{VarNotDistinct}(\text{dest, loop}) : - \\
\text{Load}(\text{inst, ptr, field, dest}), \\
\text{VarPointsTo}(\text{ptr, obj}), \\
\text{FieldNotDistinct}(\text{obj, field}), \\
\text{LoopInContext}(\text{inst, loop}).
\]

\[
\text{VarNotDistinct}(\text{dest, loop}) : - \\
\text{Load}(\text{inst, ptr, field, dest}), \\
\text{VarNotDistinct}(\text{ptr, loop}).
\]
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

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

- Integer induction variable distinct
- List elements are distinct
- Parent instance is distinct
- Map values are globally distinct
- `i` is distinct (map key iter value)
- `map.get(i)` is distinct
- Child instance is distinct
- `childPtr` is field-distinct
- `i` is distinct (from list)
- `map.get(i)` is distinct
- `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();
}

• 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.)

% of Dynamic Insns

- dacapo.batik
- dacapo.pmd
- dacapo.xalan
- cpu.djbdd
- cpu.jacc
- cpu.jlatexmath
- cpu.jscheme
- cpu.sabledd
- cpu.sat4j
- olden.bh
- olden.em3d
- olden.mst
- olden.power
- pbbs.intsort
- pbbs.nn
- pbbs.rmdup
- AVG

Affine  Daedalus

60.15
21.84
Evaluation: Parallel Speed-ups

Parallelization Speedup, 4 Cores

Perf. Vs. 1 Core

Affine-4 Daedalus-4

dacapo.batik dacapo.pmd dacapo.xalan cpu.djbb cpu.jacc cpu.jlatexmath cpu.jscheme cpu.sabledd cpu.sat4j olden.bh olden.em3d olden.mst olden.power pbbs.intsort pbbs.nn pbbs.rmdup GEOMEAN

1.274 1.076
Outline

• Introduction
• First-Class Data Structures
• DAEDALUS: Distinctness Analysis
• ICARUS: Incorporating Dynamic Checks
Is Static Analysis Enough?

• Consider the following snippet:

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

• 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?

```java
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.
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

- add(): it
- List (element)
- process(): it
- compute(): m
- metadata (within-map)

Possible check

Distinctness fact
Rule application

Distinct
Not Distinct
Systematically Leveraging Dynamic Checks

Possible check

Distinctness fact

Rule application

add(): it

List (element)

process(): it

compute(): it

metadata (within-map)

compute(): m

Distinct

Possibly Distinct

Not Distinct

 Needed
Systematically Leveraging Dynamic Checks

```
Distinctness fact
Rule application
add(): it
R1
List (element)
R2
process(): it
R3
metadata (within-map)
R4
compute(): it
compute(): m
Possible check
?
!
Needed for parallelization

Distinct

Possibly Distinct

Not Distinct

! Needed

Distinctness fact
Rule application
```
Executing with Dynamic Checks

• If checks always succeed, we’re done!
• What if a check fails?

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

```
CPU 0
i = 0
p = 0x1000
p.f = 42

CPU 1
i = 1
p = 0x2000
p.f = 42

CPU 2
i = 2
p = 0x1000
p.f = 42

...
Executing with Dynamic Checks

• If checks always succeed, we’re done!
• What if a check fails?

```plaintext
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*  
  
<table>
<thead>
<tr>
<th>Pointer (63 bits)</th>
<th>ND</th>
</tr>
</thead>
</table>

• **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

```c
for (...) {
    for (...) {
        p = ...;
    }
}
```

![CPU allocation diagram]

- CPU 0
  - $i = 0$
  - $p = 0x1000$
  - $P = 0x3000$

- CPU 1
  - $i = 0$
  - $p = 0x2000$
  - $P = 0x3000$

- CPU 2
  - $i = 1$
  - $p = 0x2000$

...
Sequencing the Checks

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

```c
for (...) {
    for (...) {
        p = ...;
    }
    i = 0
    p = 0x1000
    p = 0x2000
    p = 0x3000
    ... 
}
```

• **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

% of Dynamic Insns

- dacapo.pmd
- cpu.cloudsim
- cpu.djbdd
- cpu.jacc
- cpu.sabledd
- olden.bh
- olden.em3d
- olden.mst
- olden.power
- olden.power.comparison
- pbbs.comparison
- pbbs.insort
- pbbs.nn
- pbbs.nncast
- AVG

Affine | Daedalus | Icarus-Success | Icarus-Serialize

81
Evaluation: Speed-up (Upper Bound)

16 Cores, 1-Cycle WQ, Perf Caches

Parallel Speedup

- **dacapo.pmd**
- **cpu.cloudsim**
- **cpu.djbd**
- **cpu.jacc**
- **cpu.sabledd**
- **olden.bh**
- **olden.em3d**
- **olden.mst**
- **olden.power**
- **pbbs.comparisonsort**
- **pbbs.intsort**
- **pbbs.nn**
- **pbbs.raycast**
- **GEOMEAN**

**Affine**

**Daedalus**

**Icarus**
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?