Bounded-Latency Regional Garbage Collection

Felix S Klock II (Adobe Systems)
William D Clinger (Northeastern University)

The problem

- Naïve GC
  -

The problem

- Naïve GC $\Longrightarrow$ long pauses
-

Scheme with stop+copy collector
pueue200:1000000:50:50

```
Scheme with stop+copy collector
pueue200:1000000:50:50

.........................
```

Scheme with stop+copy collector
pueue200:1000000:50:50

.........................#............

Scheme with stop+copy collector
pueue200:1000000:50:50

```
.........................#............#...........
........
```

Scheme with stop+copy collector
pueue200:1000000:50:50

```
.........................#............#...........
........#........................#................
.........#........................#...............
..........#........................#..............
```

The problem

- Naïve GC $\Longrightarrow$ long pauses
- Generational GC

The problem

- Naïve GC $\Longrightarrow$ long pauses
- Generational GC $\Longrightarrow$ long pauses less often
-

Java with generational collector

```
java -d32 -Xmx1900M PueueT 200 1000000 50 50
```

Java with generational collector

```
java -d32 -Xmx1900M PueueT 200 1000000 50 50
```

....

```
Java with generational collector
java -d32 -Xmx1900M PueueT 200 1000000 50 50

.....................#.............#.............#
.............................#....................
..........#................................#......
...........................#......................
```

The problem

- Naïve GC $\Longrightarrow$ long pauses
- Generational GC $\Longrightarrow$ long pauses less often
- Real-time / incremental / concurrent GC
  - may add overhead to all programs
  - may require mutator-specific fiddling
  -

The problem

- Naïve GC $\Longrightarrow$ long pauses
- Generational GC $\Longrightarrow$ long pauses less often
- Real-time / incremental / concurrent GC
  - may add overhead to all programs
  - may require mutator-specific fiddling
  - ***may still have long pauses***

```
Java with garbage-first collector
java -XX:+UnlockExperimentalVMOptions \
  -XX:+UseG1GC -Xmx1900M PueueT 200 1000000 50 50
..................................................
..................................................
........................#.........................
................................................#.
```

Dirty Little Secret...

Most Real-Time Garbage Collectors Aren't.

Most Incremental Garbage Collectors Aren't All That Great Either.

Longest GC Pause

		gcbench	perm	queue	pueue
Scheme	stop&copy	2.94	3.44	4.62	4.74
Scheme	generational	3.13	3.23	4.28	4.45
Java	default	2.78	2.93	3.24	3.32
Java	concurrent m/s	15.45	0.50	0.45	5.94
Java	garbage-first	2.13	4.68	4.29	5.84
Scheme	regional	0.12	0.13	0.09	0.21

Scheme with regional collector
pueue200:1000000:50:50

....
....
....
....

Scalability
in space and time

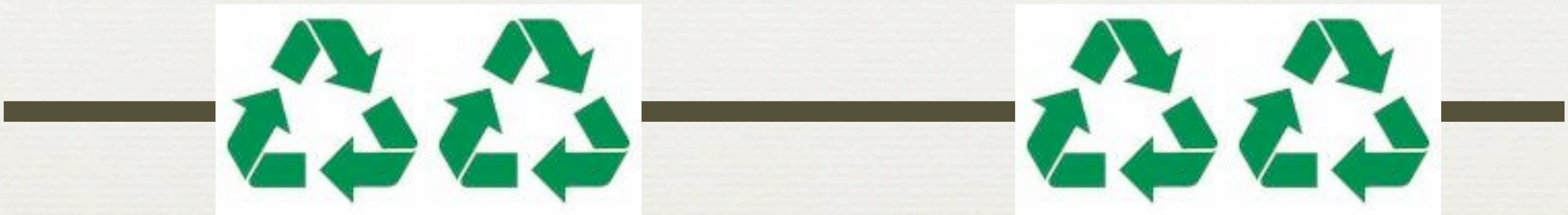
Control Space:

Metadata & Floating Garbage

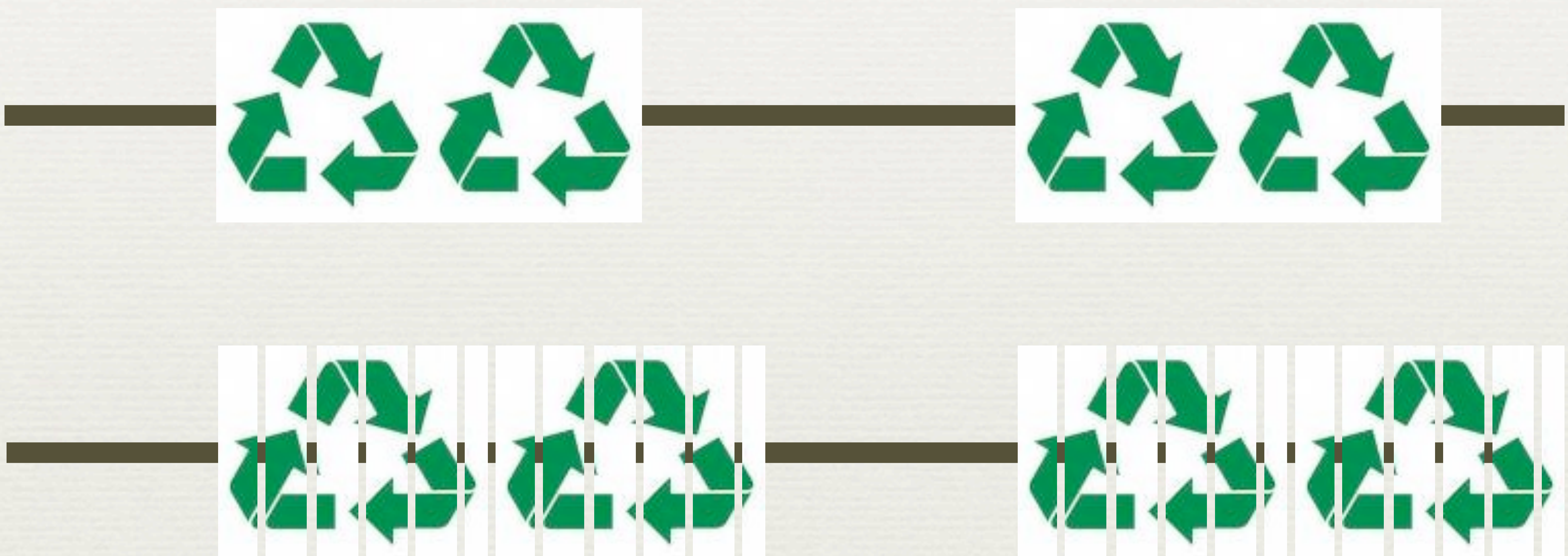
Control Time:

Pause times & Mutator Utilization

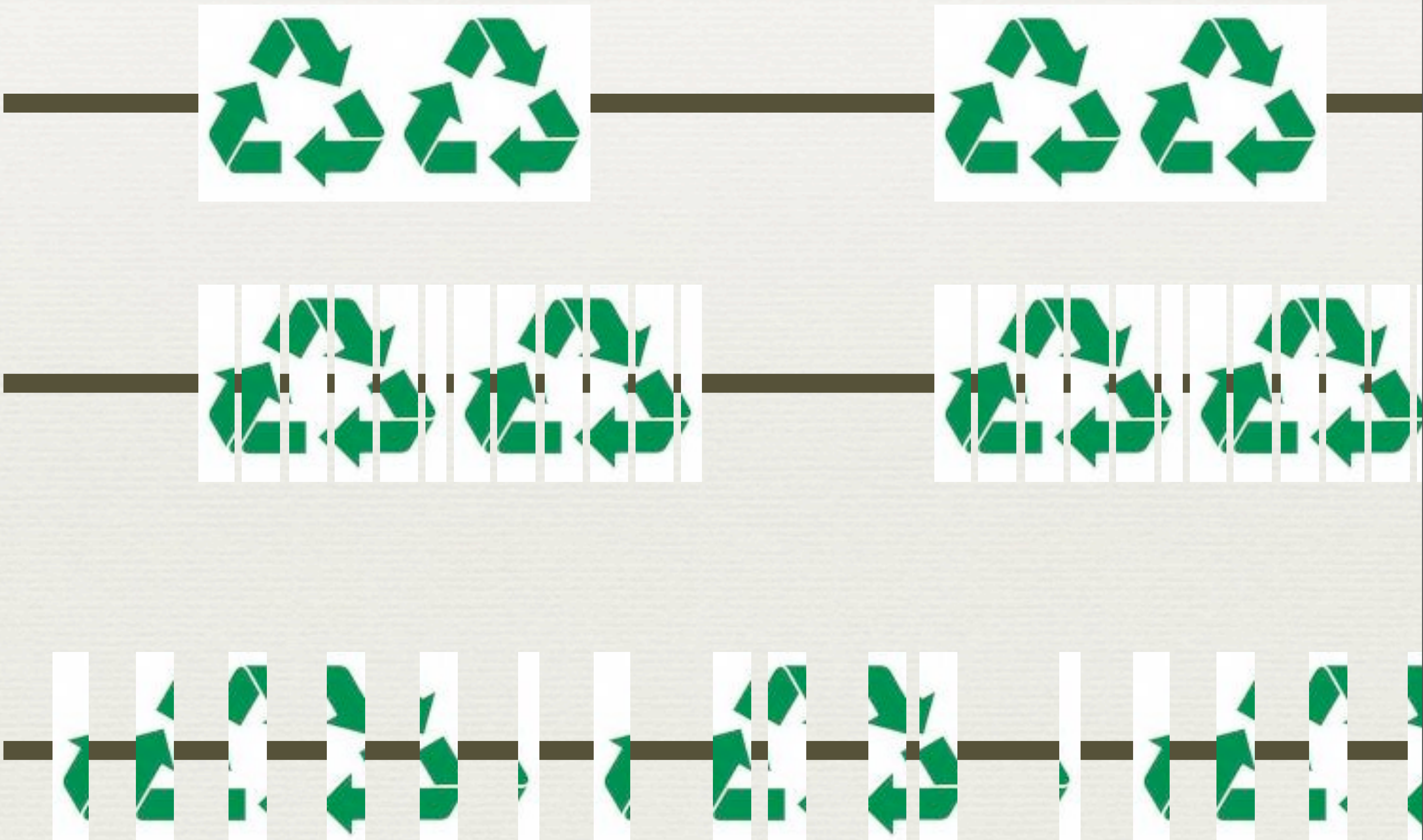
Pauses are disruptive



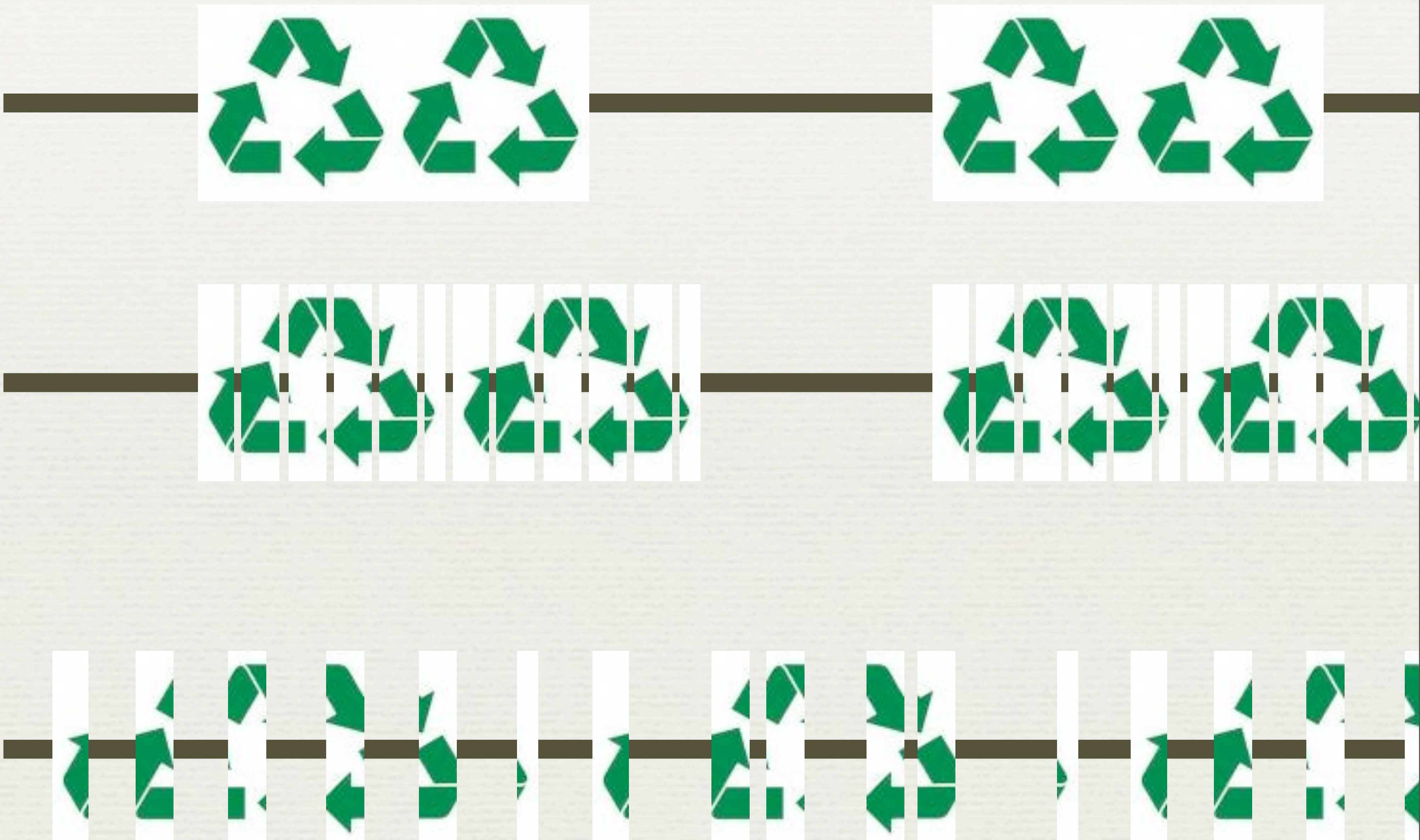
Pauses are disruptive

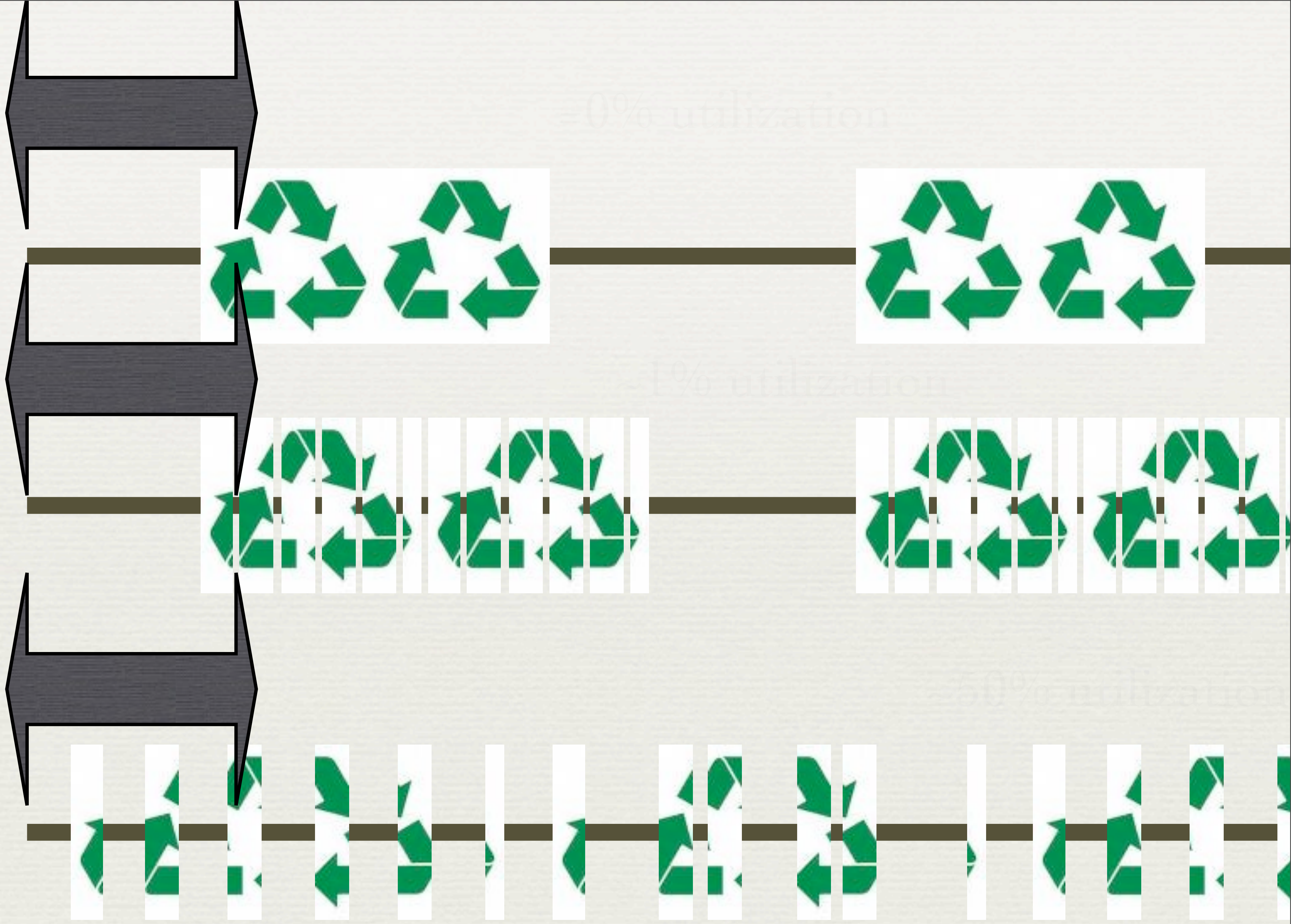


Bounded pauses can still be disruptive

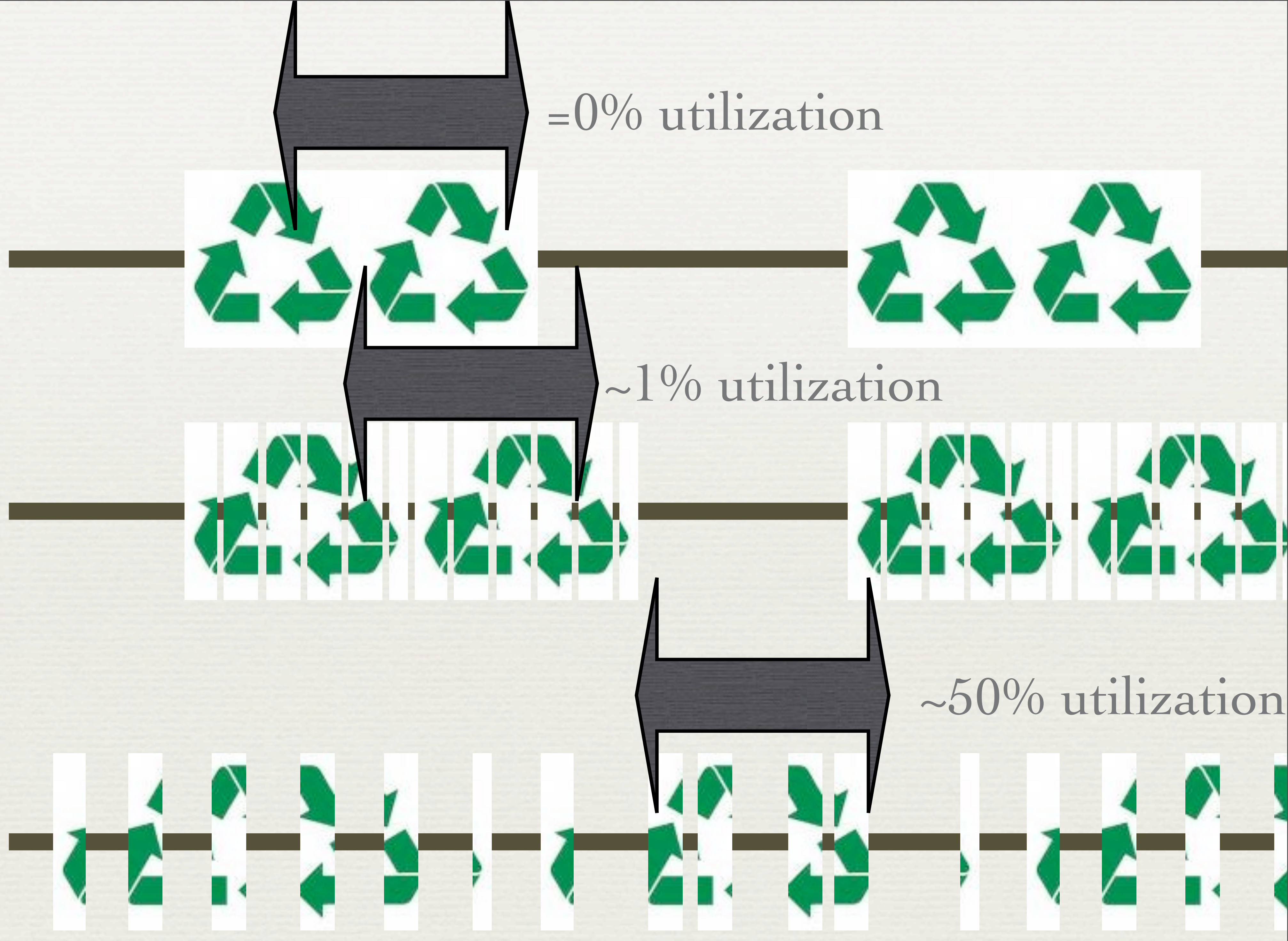


Minimum Mutator Utilization (MMU)





~0% utilization
~1% utilization
50% utilization


=0% utilization
~1% utilization
~50% utilization

Scalability (Definition)

There exist fixed worst-case bounds

Scalability (Definition)

There exist fixed worst-case bounds

Independent of mutator and heap size!

Scalability (Definition)

There exist fixed worst-case bounds such that

Scalability (Definition)

There exist fixed worst-case bounds such that

For all mutators, no matter what they do:

Scalability (Definition)

There exist fixed worst-case bounds such that

For all mutators, no matter what they do:

1. All GC pauses are shorter than the fixed bound (which is independent of heap size).

2. Minimum Mutator Utilization is bounded from below (independent of heap size).

3. Memory usage is O(P), where P = peak volume of reachable objects.

Scalability (Theorem)

There exist fixed worst-case bounds such that

For all mutators, no matter what they do:

1. All GC pauses are shorter than the fixed bound (which is independent of heap size).

2. Minimum Mutator Utilization is bounded from below (independent of heap size).

3. Memory usage is O(P), where P = peak volume of reachable objects.

How It Works

“Simple” Idea

- Divide heap into “regions” of fixed size.
- Collect each region independently.
- Since regions are bounded in size, we should be able to do this in bounded time, right?

“Simple” Idea

- Divide heap into “regions” of fixed size.
- Collect each region independently.
- Since regions are bounded in size, we should be able to do this in bounded time, right?

(yes, but just barely)

Collect one region using Cheney's algorithm

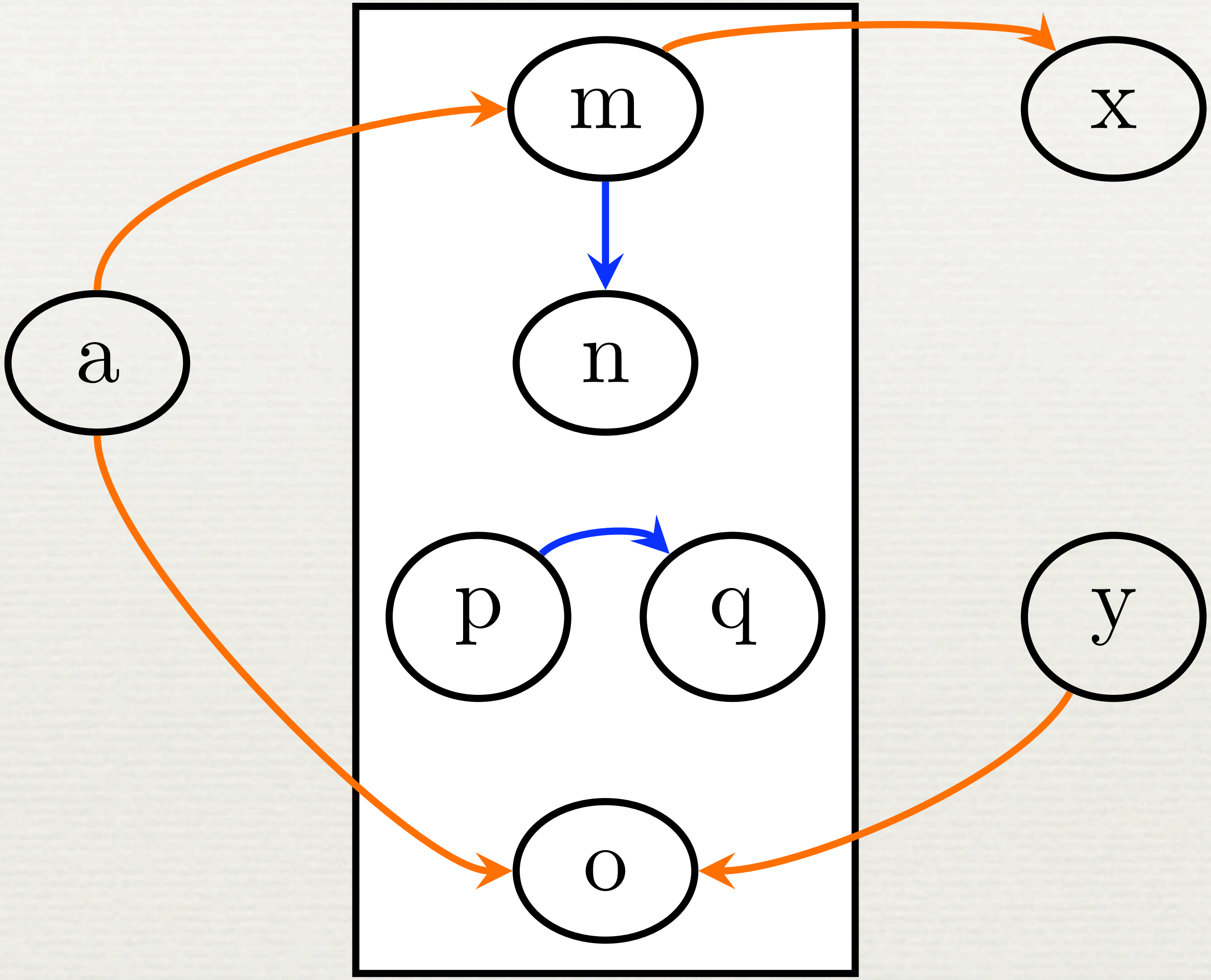
(stop&copy)

How to do this scalably?

Don't inspect extraneous state

Remembered Set?

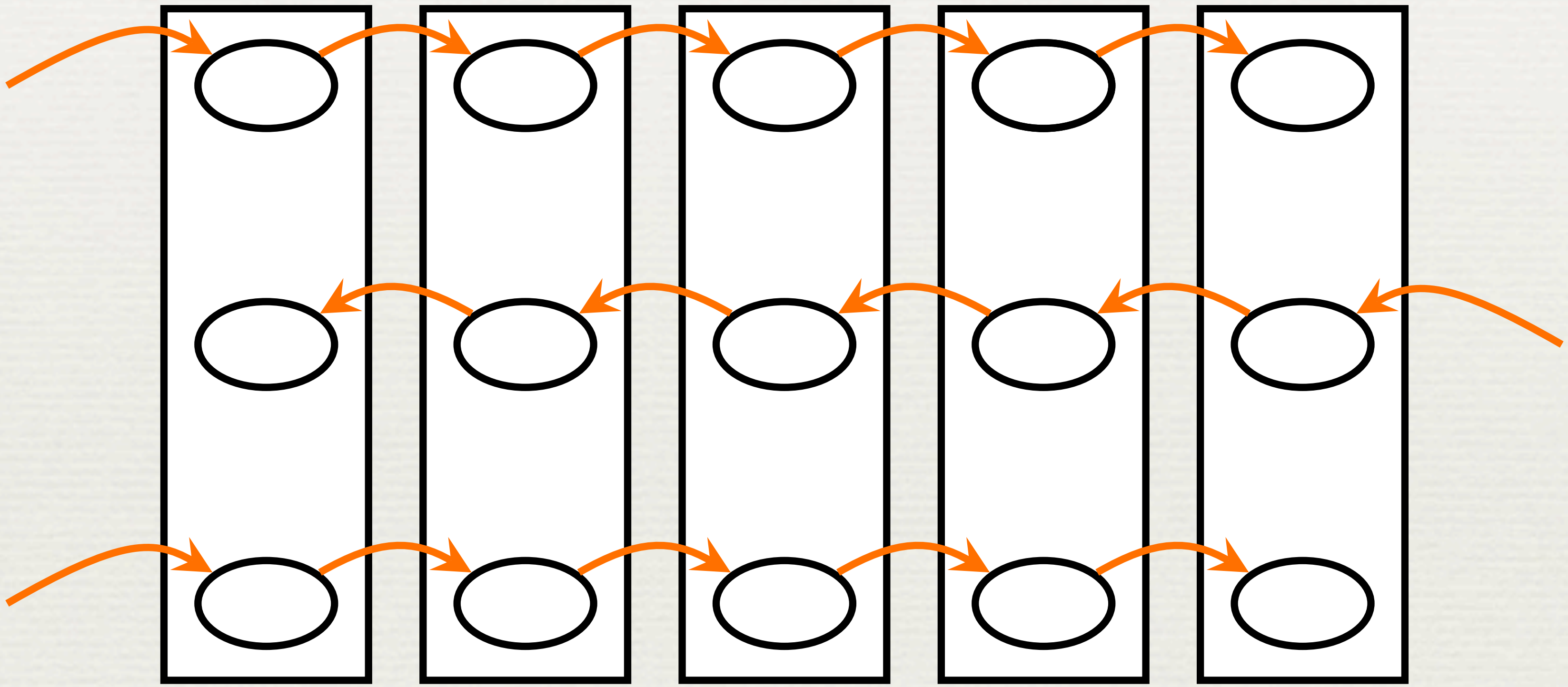
(Generational Collection
[Lieberman and Hewitt ’83, Ungar ’84])



Rem. Set $\supseteq$ { a, m, y }

Problem with Remembered Set

$|\text{Rem. Set}| \propto |\text{Heap}|$



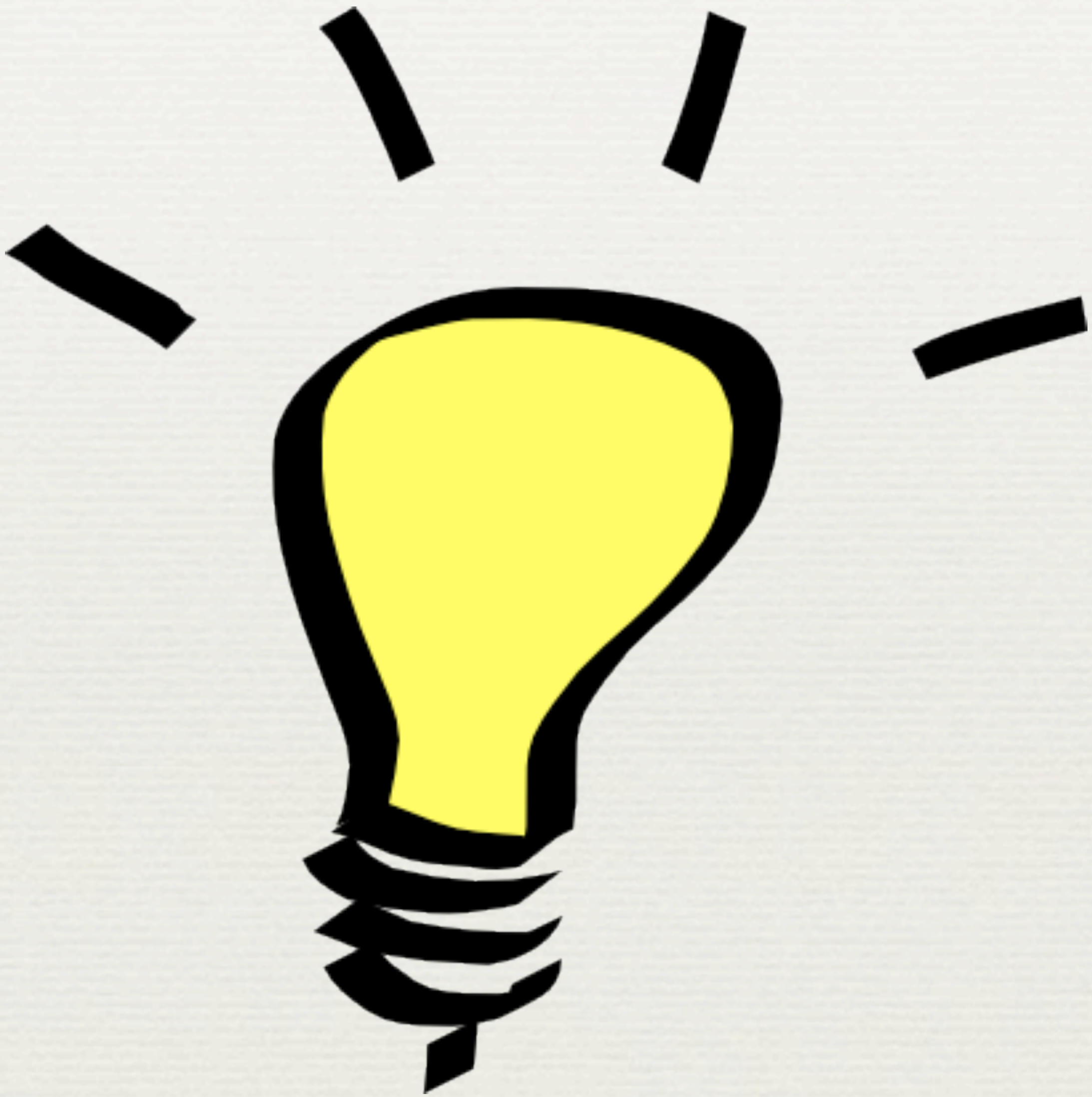
scan time could be worse than proportional to region size

Per-region remembered sets?

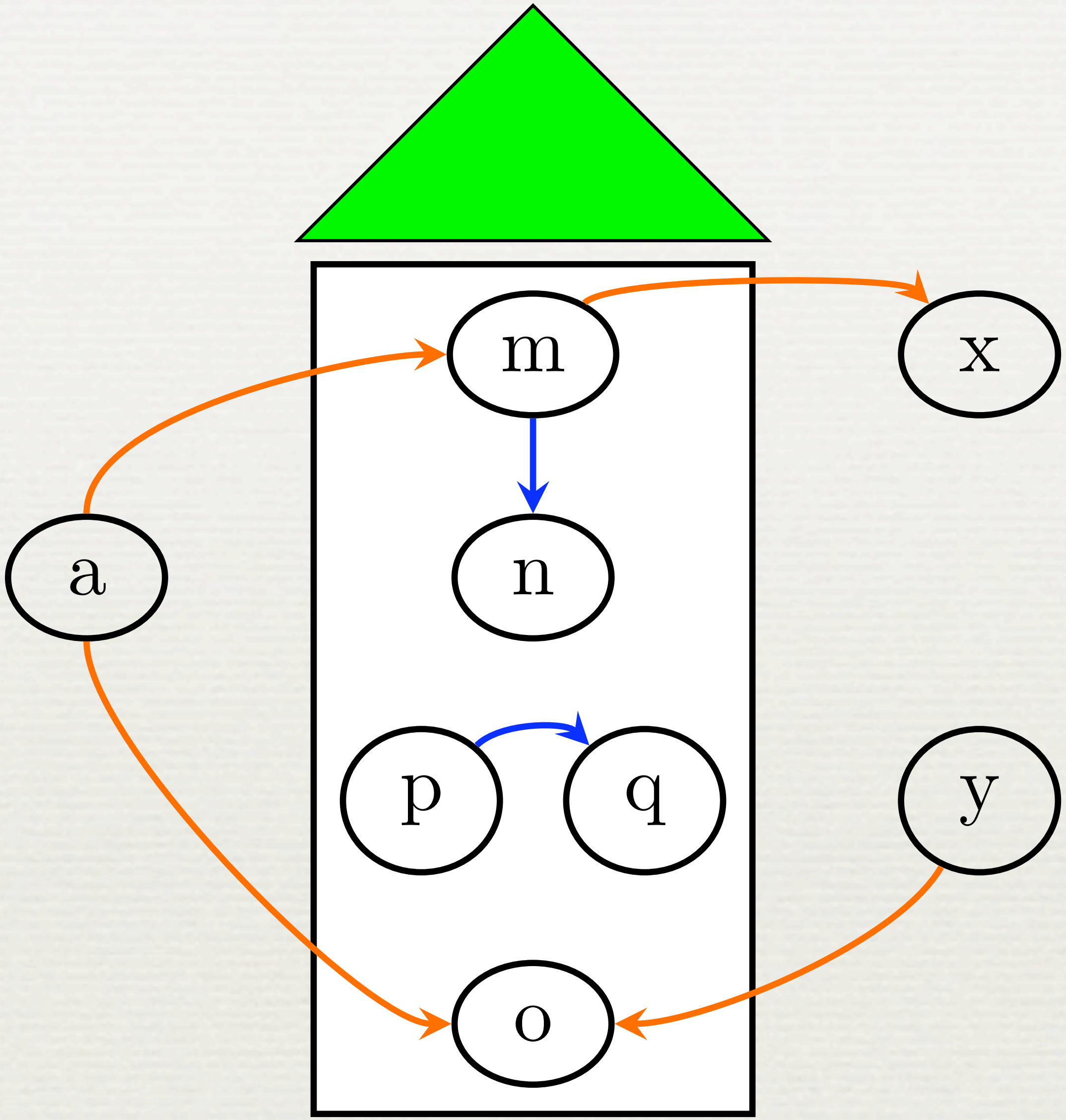
(Garbage-First Collection [Detlefs '04])

Need Space Bounds!

- Garbage-First "Points-into remembered sets"
- Unacceptable $O(N^2)$ worst-case space cost



Compute Summary Sets Just in Time

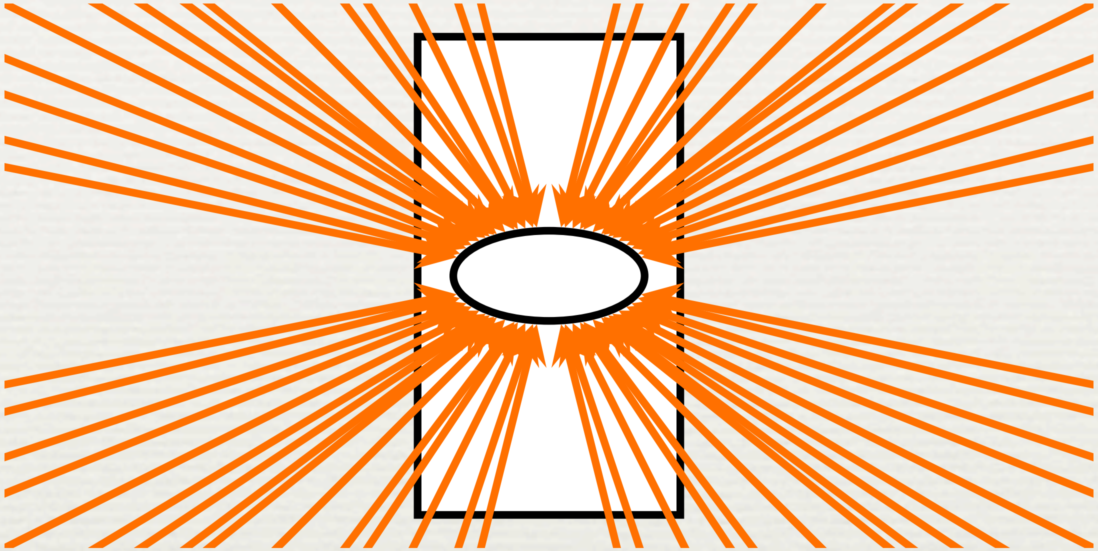


This summary set ⊇ { &a[1], &a[3], &y[0] }

Summary Sets

- Does it work?
  - Popular objects / regions
  - Space cost

Problem #1: Popularity



- Many locations may point to one object
  - (or group of objects co-located in same region)
- Summary set will be LARGE!

Problem #2: Space

- Maintaining precise summary sets for every region at all times is unrealistic
  - (takes too much time)
- Maintain imprecise summary sets throughout execution?
  - (no, that takes us back to the unacceptable $O(N^2)$ bound of Garbage-First)

Key Insight:
Not all regions are above average.

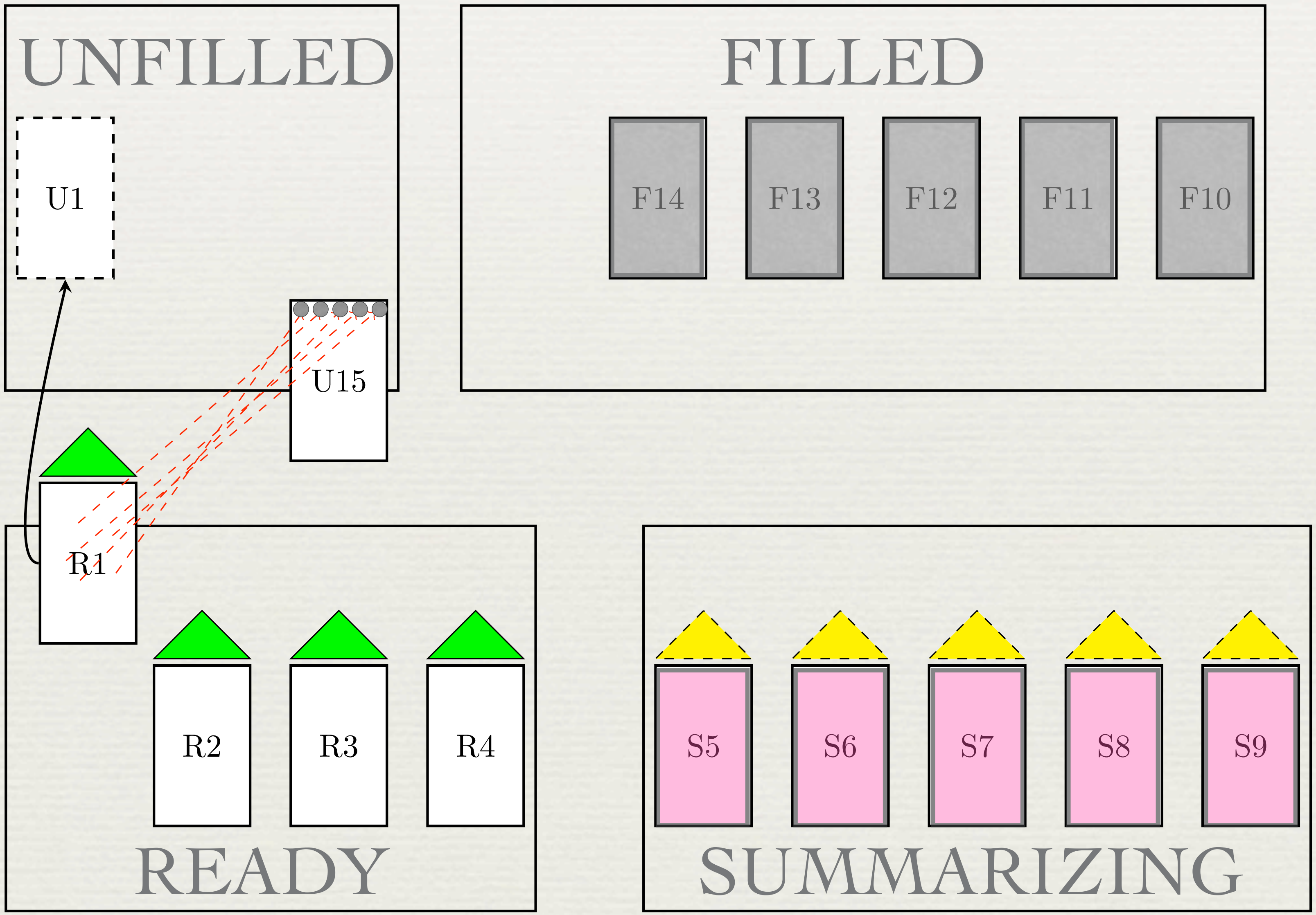
Popular Regions

- *Unusually* popular regions *must be* unusual.
- *Don't collect* unusually popular regions!
- *Wave off* their summaries before completion!
- Solves *both* problems

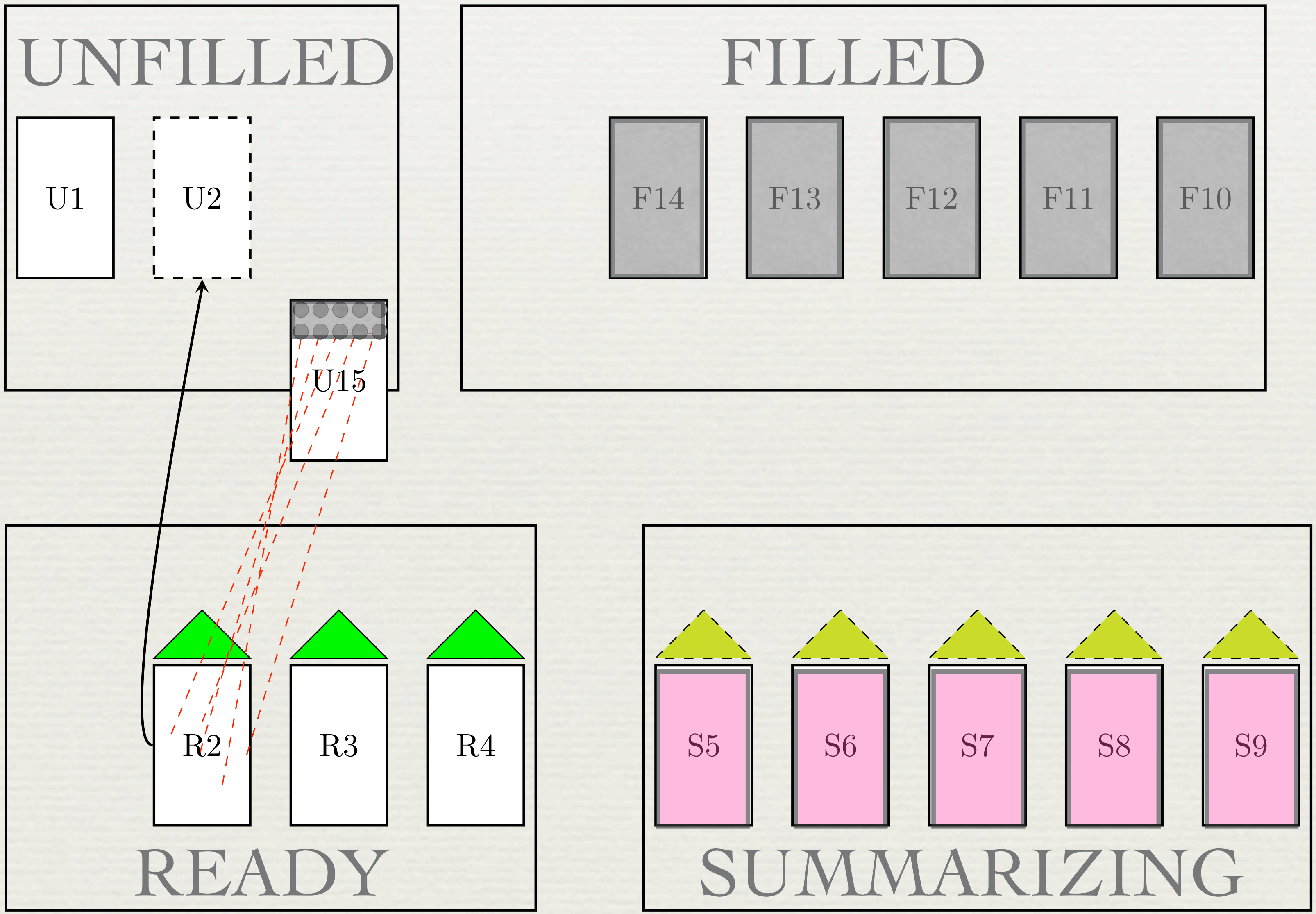
Summarization: Amortized

- Constructing one summary set generally involves scanning the entire heap.
- Not enough time to construct the next summary set unless we start early, so
- Start early!
- Amortize the effort!
- Construct summary sets for many regions at once during one incremental scan.

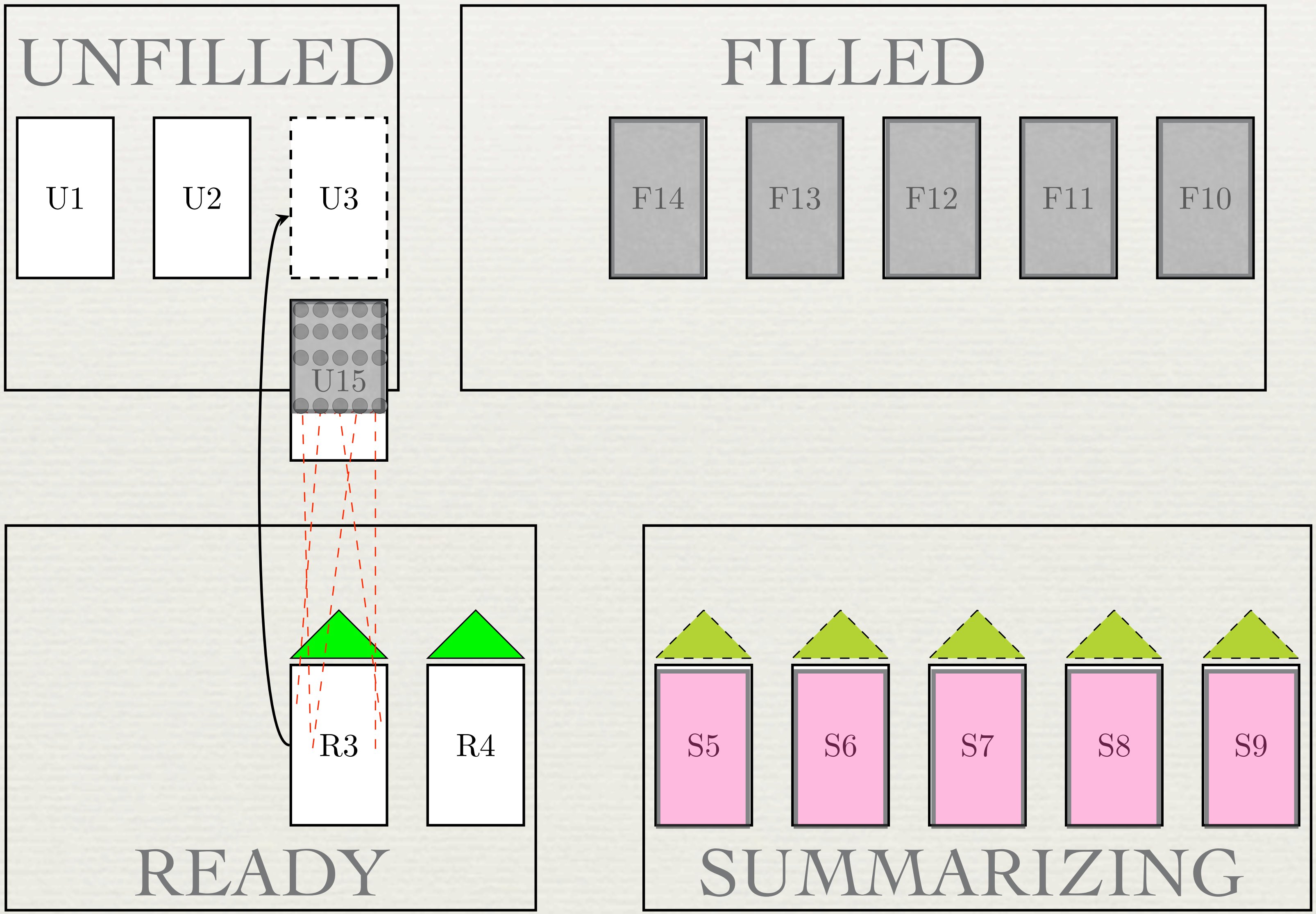
Summarization Cycle



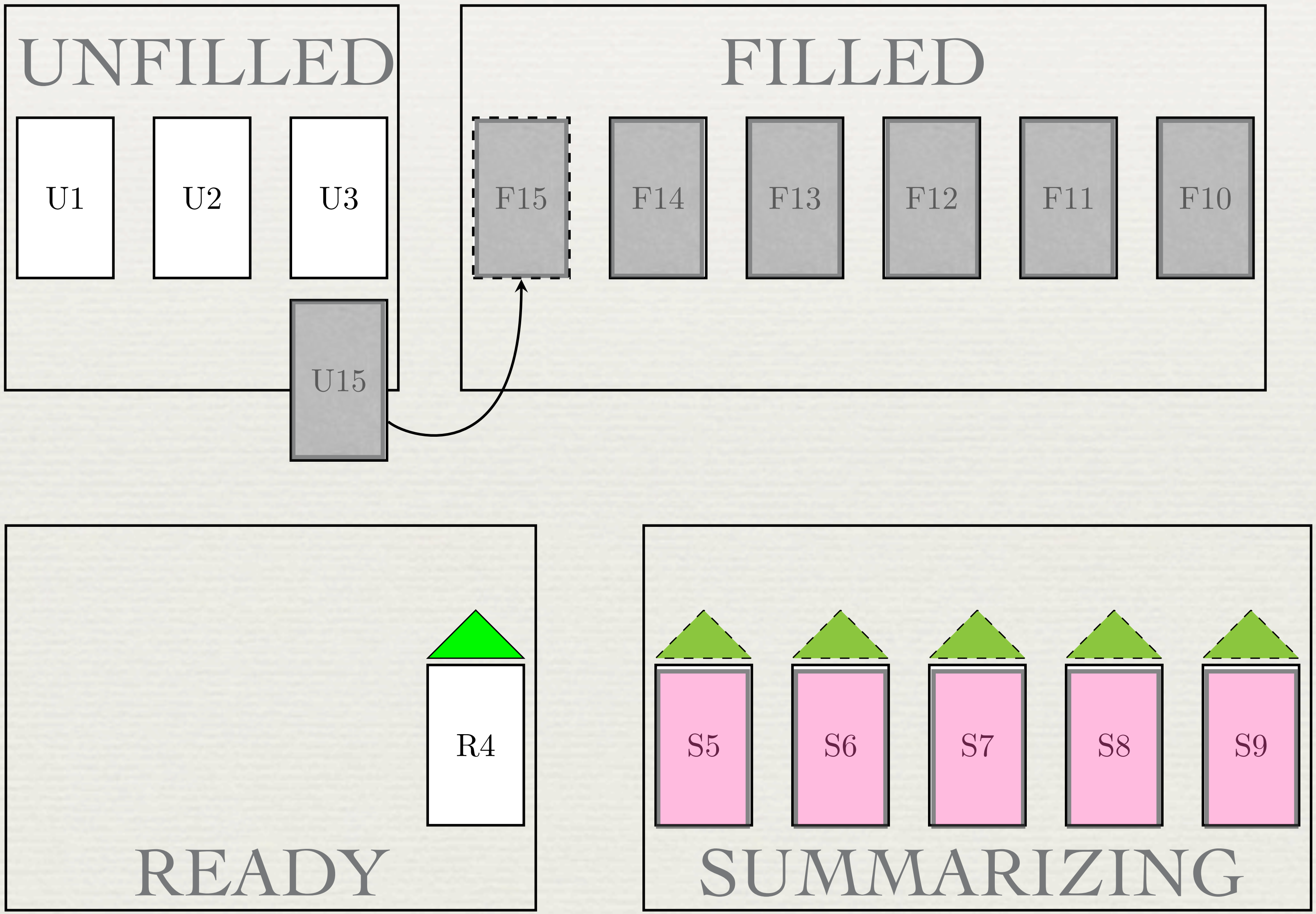
Summarization Cycle



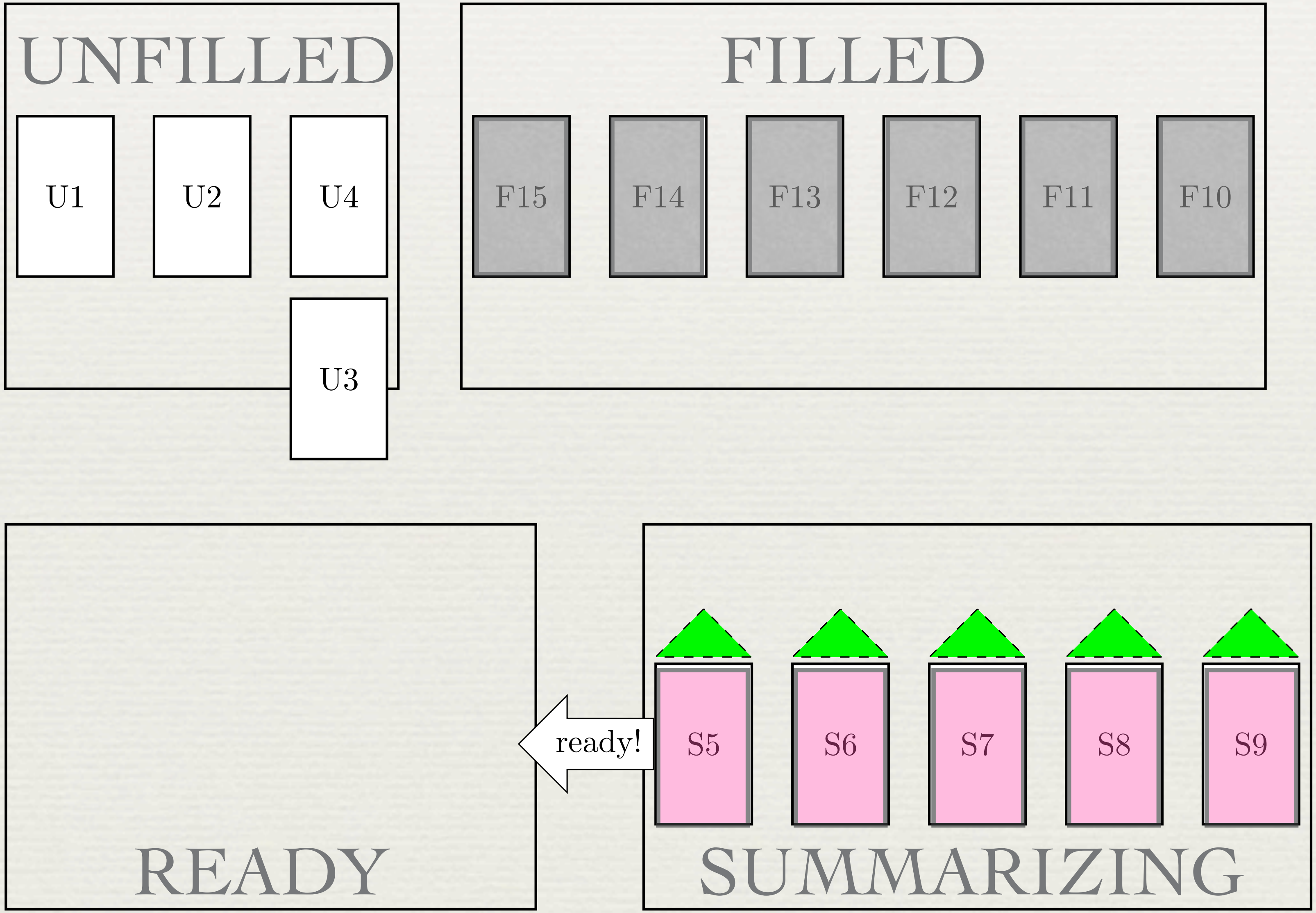
Summarization Cycle



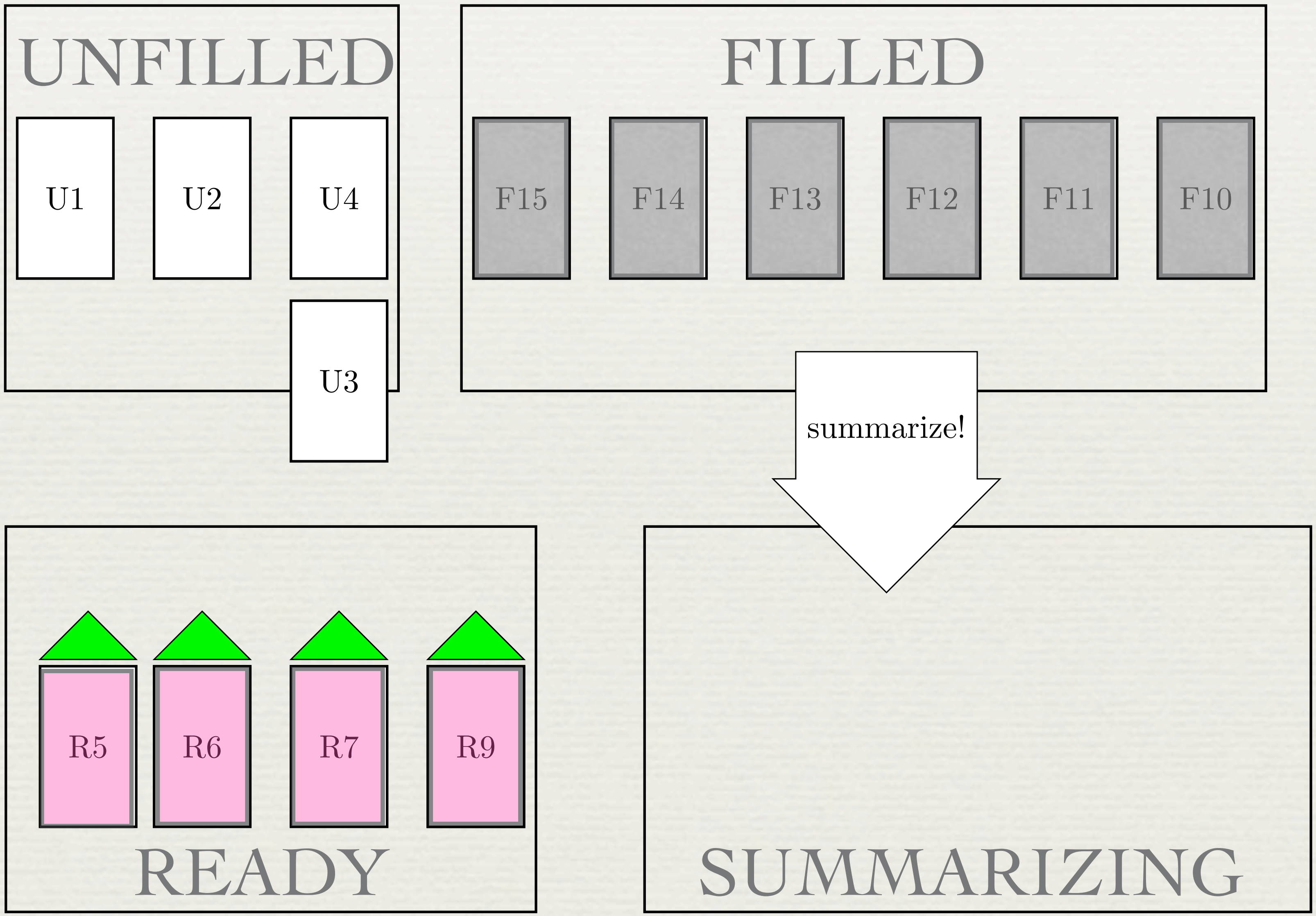
Summarization Cycle



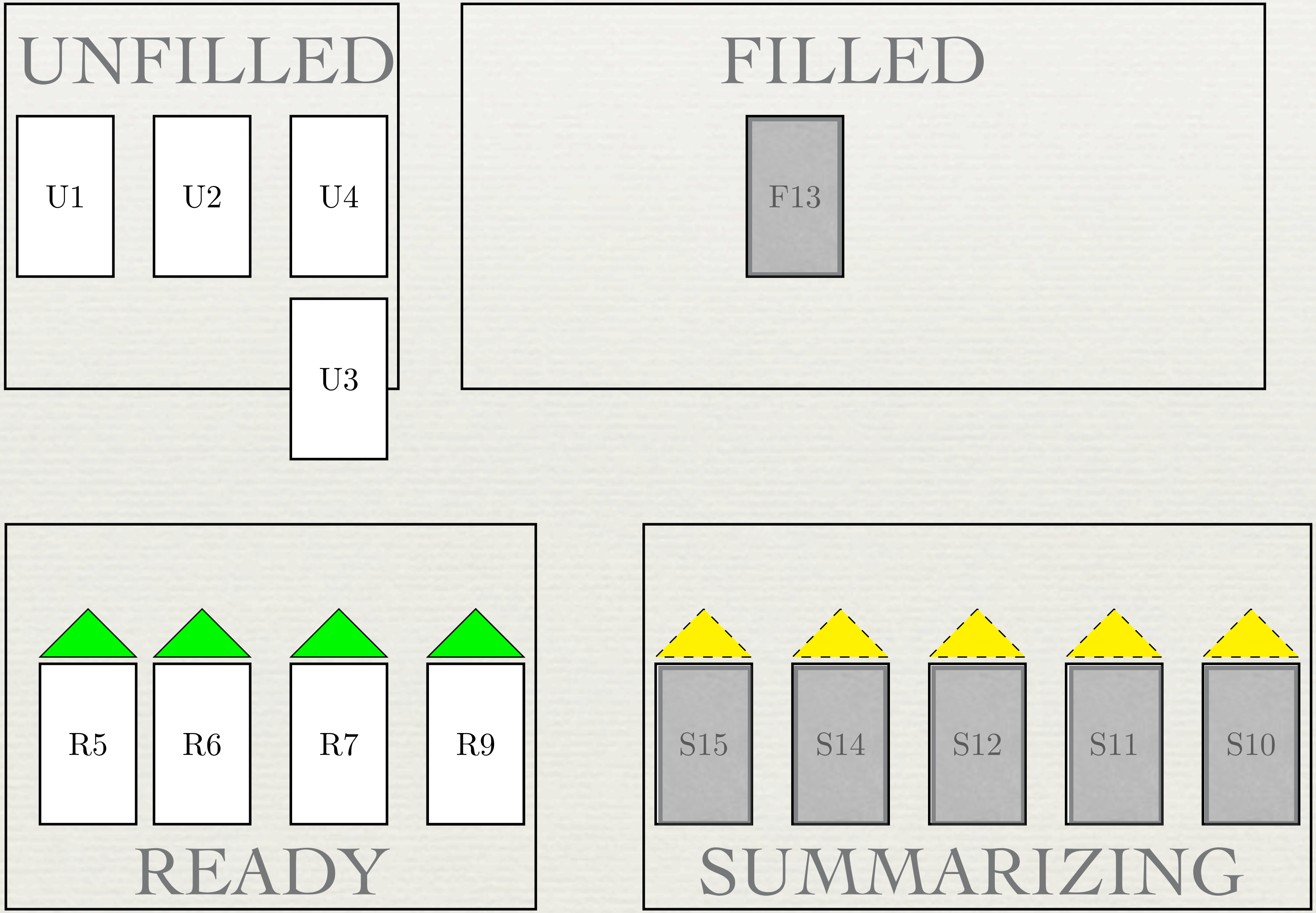
Summarization Cycle



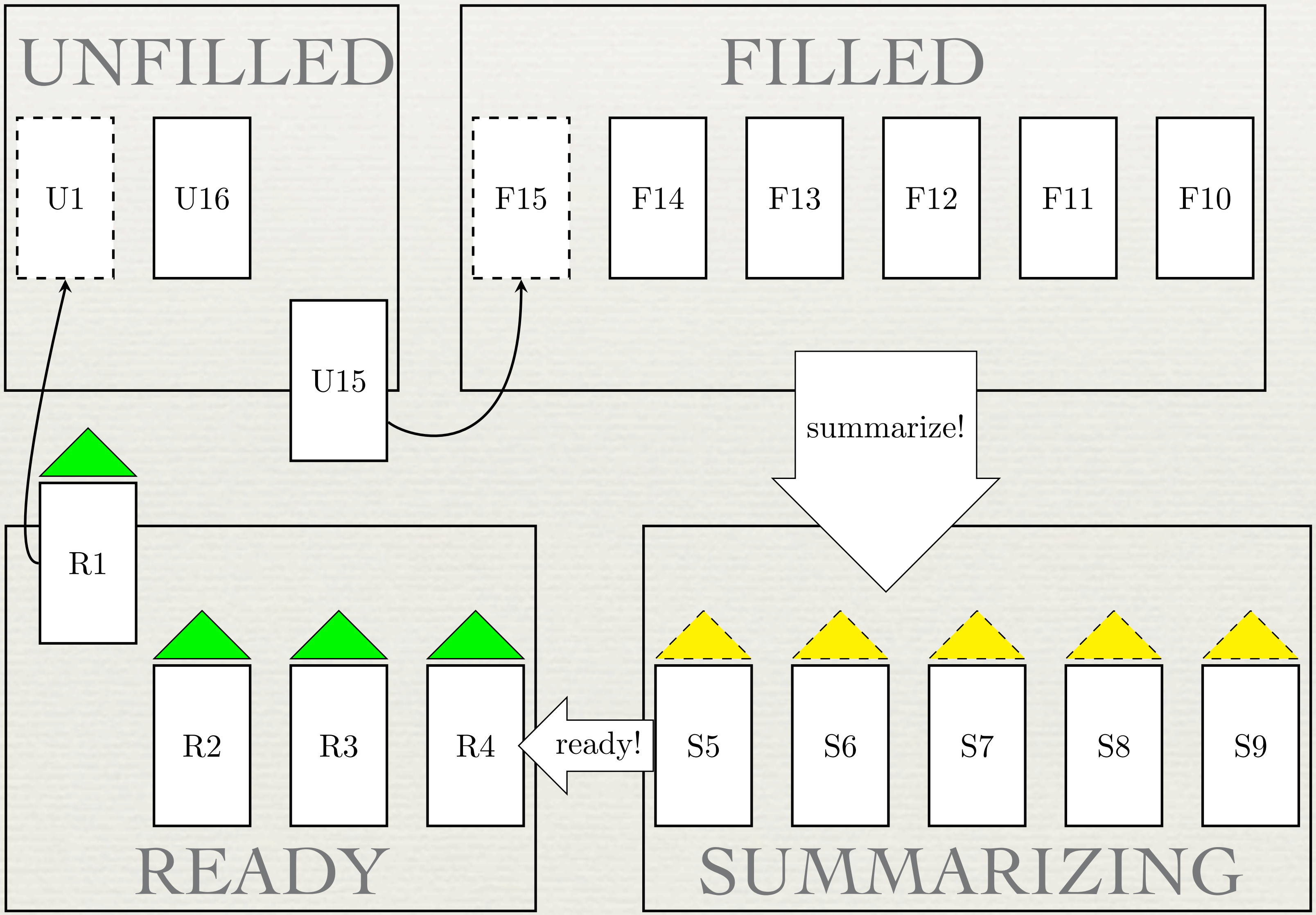
Summarization Cycle



Summarization Cycle

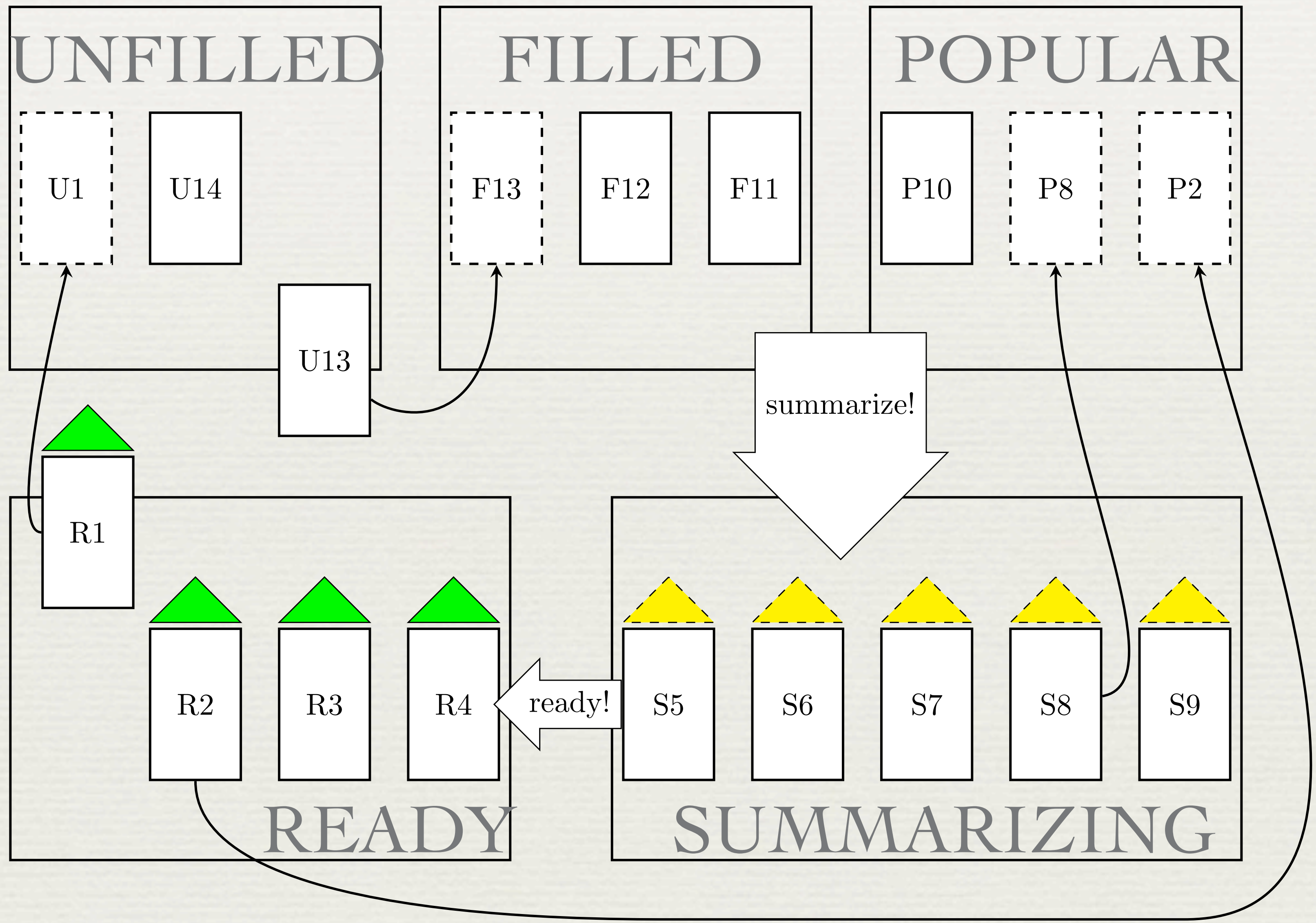


Summarization Cycle



What about the popular regions?

More Accurate Picture

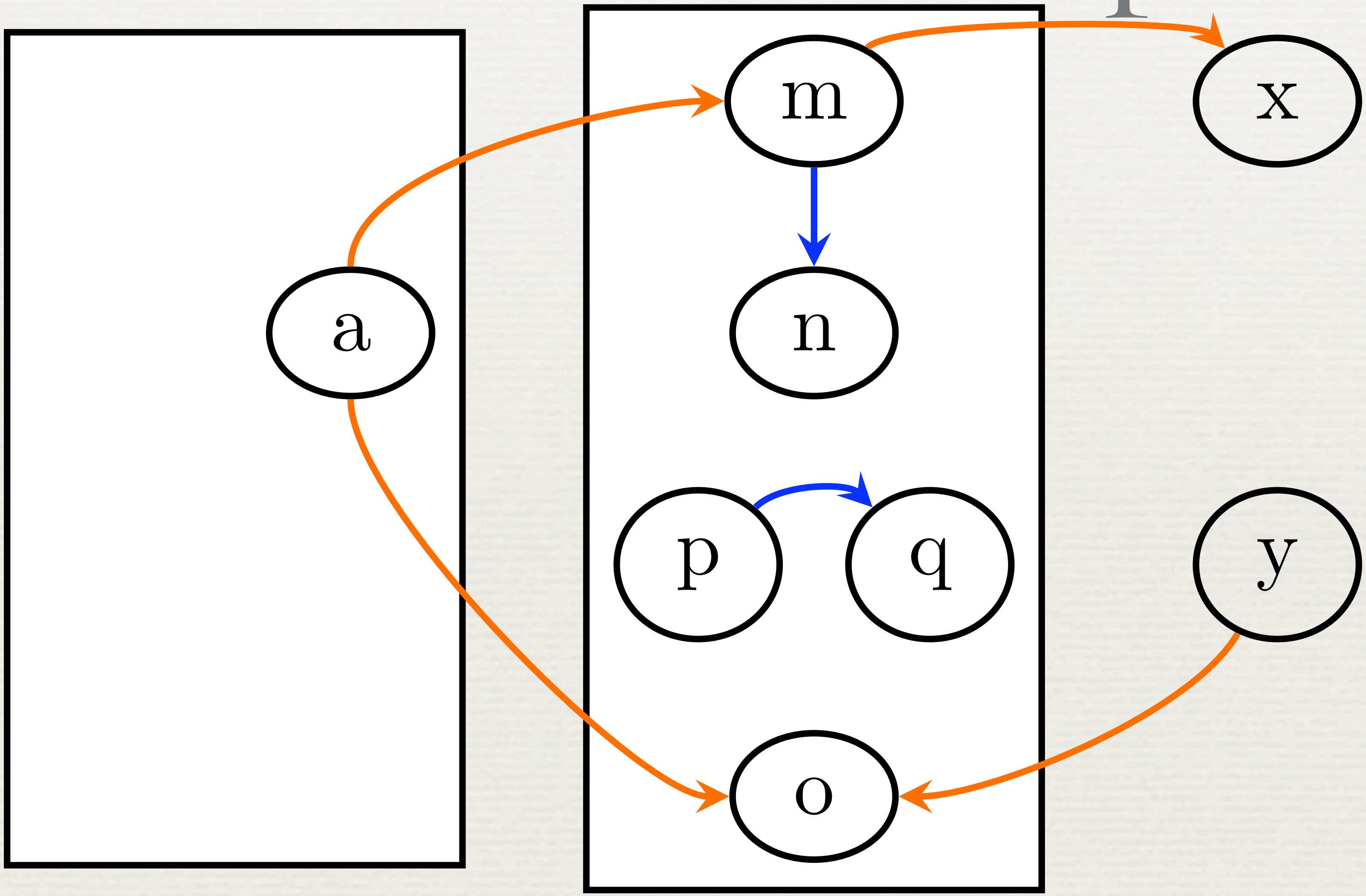


Cyclic Garbage May Cross Region Boundaries

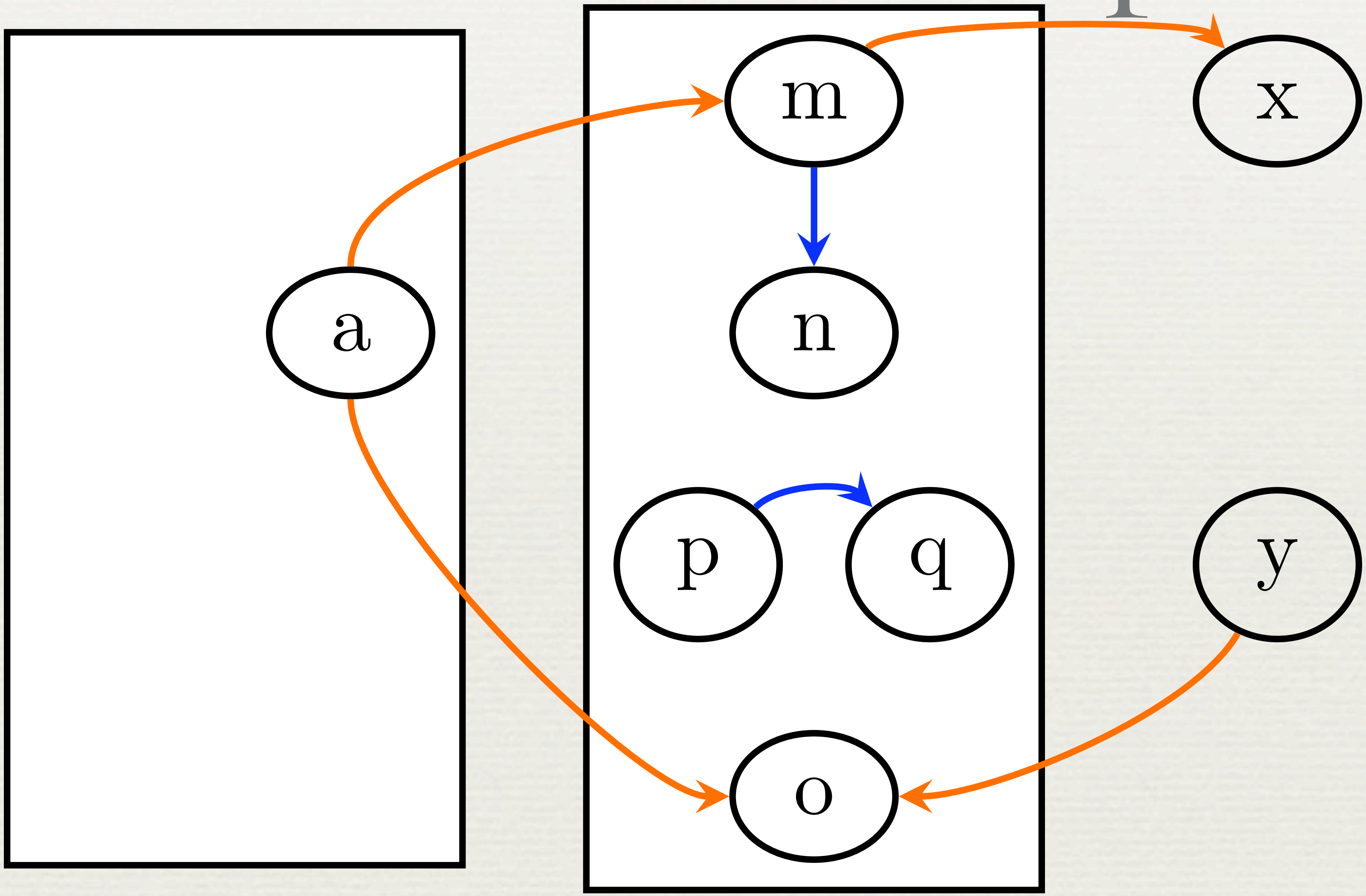
How to collect cycles?

- Use Snapshot-at-the-Beginning (SATB) [Yuasa'90] to refine remembered set and summary sets.
- Also ensures popular regions won't hold onto other regions' objects forever!

Refinement via Snapshot

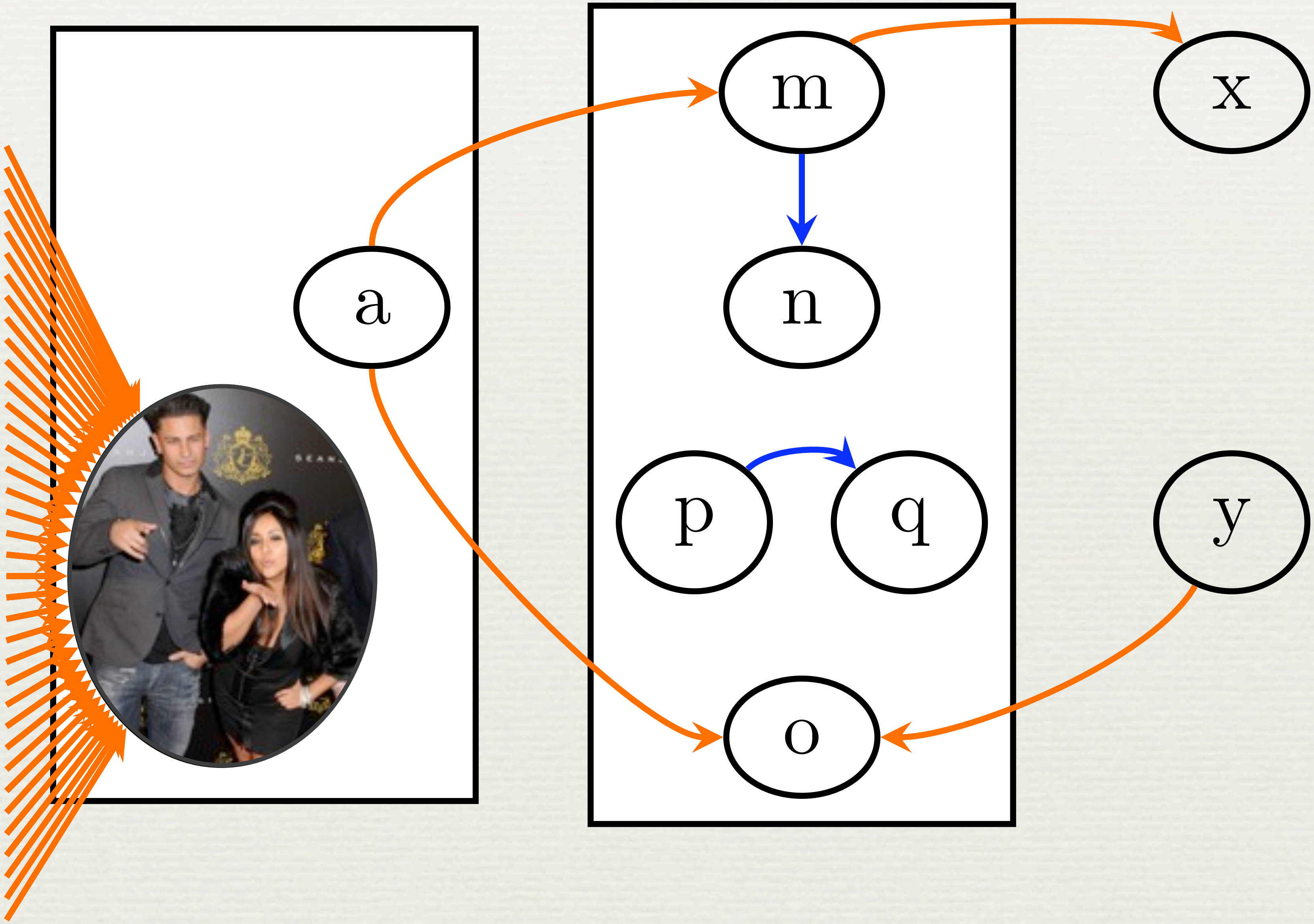


Refinement via Snapshot

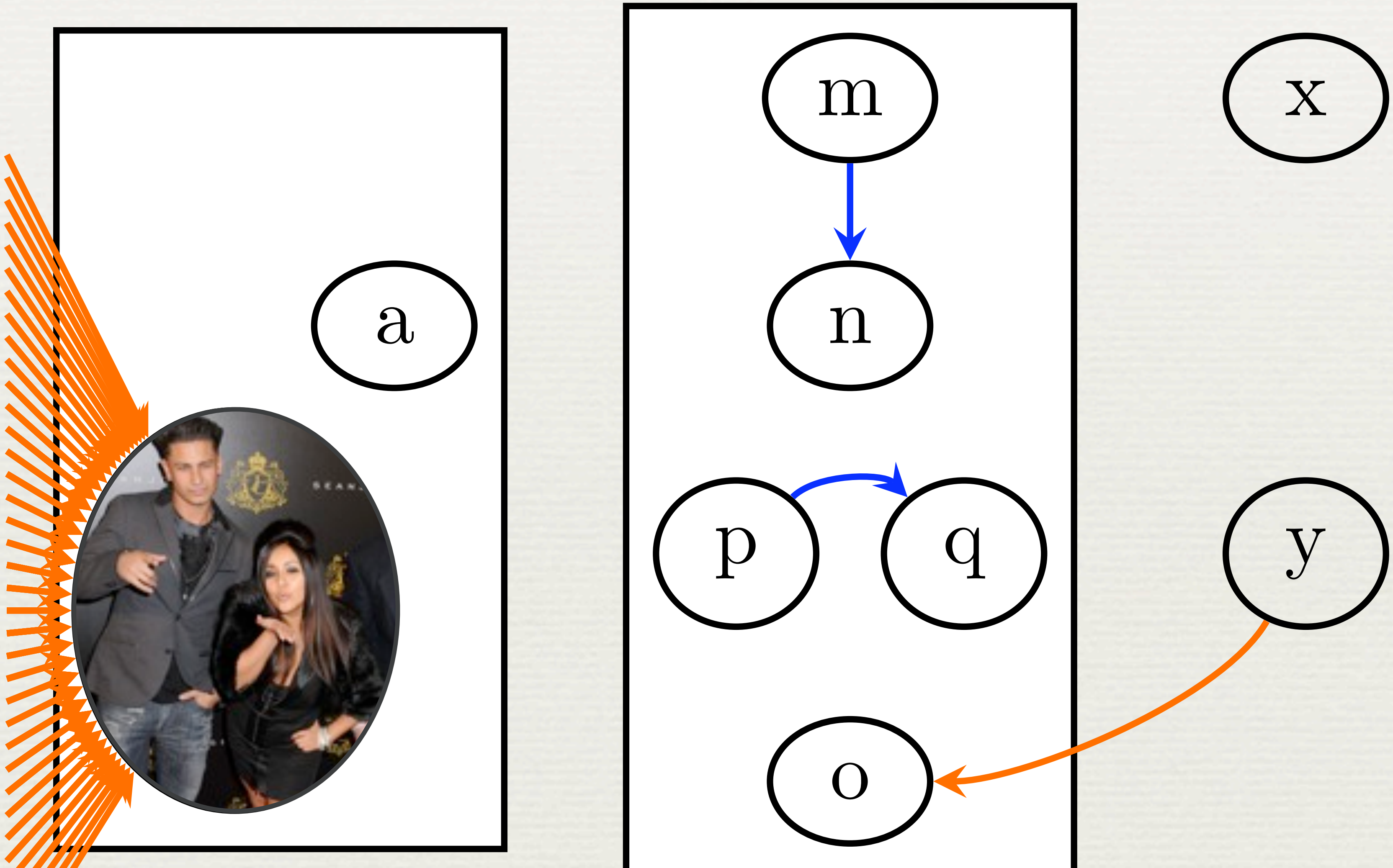


what if (a) were unreachable and in a region with popular objects?

Before Refinement



After Refinement



(still popular; not collected)

(still collected; far more reclaimed)

Implementation

Larceny

- Scheme (IEEE/ANSI/R5RS/R6RS)
- Built for compiler and GC research
- Interchangeable collectors
  - stop-and-copy
  - generational
- Full control; enforce system invariants and implement specialized write-barriers

Larceny Regional GC

- Added dynamic region allocation
- Modified write-barrier for SATB marker
- Modified Cheney core
  - Update remembered set, marker state, etc
- Summary sets

Read Felix's Dissertation!

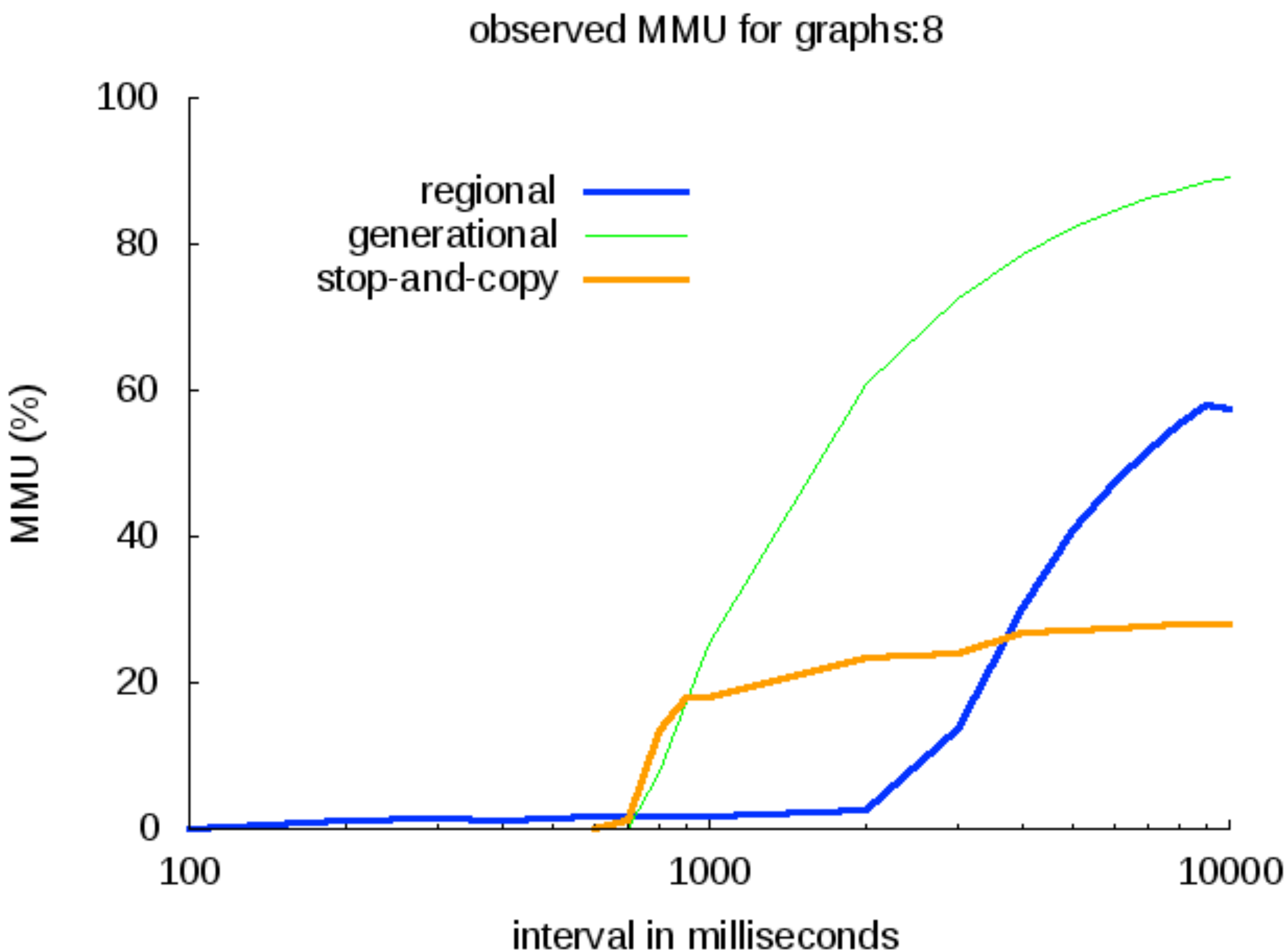
Evaluation

Larceny Benchmarks

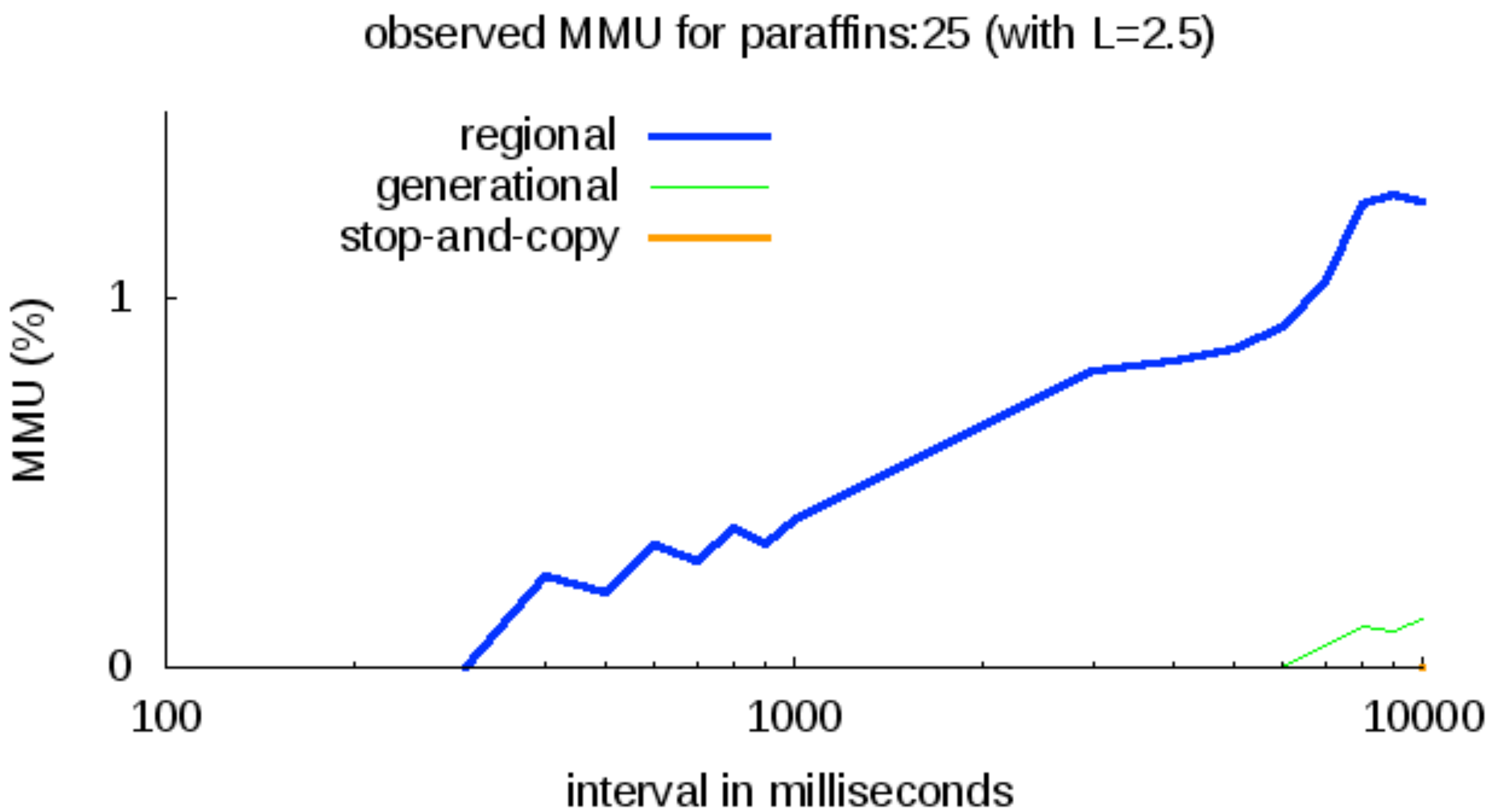
- Standard set of 68 R6RS benchmarks
  - Can regional collector compete with generational?
- Near-worst-case benchmarks
  - Is regional collector scalable?
  - How bad are the worst-case bounds?

Representative Benchmarks

- Compared to Larceny's generational collector:
  - regional GC is 12% slower overall
  - stop-and-copy GC is 23% slower



observed MMU for graphs:8
regional
generational
stop-and-copy
MMU (%)
0
20
40
60
80
100
100
1000
10000
interval in milliseconds

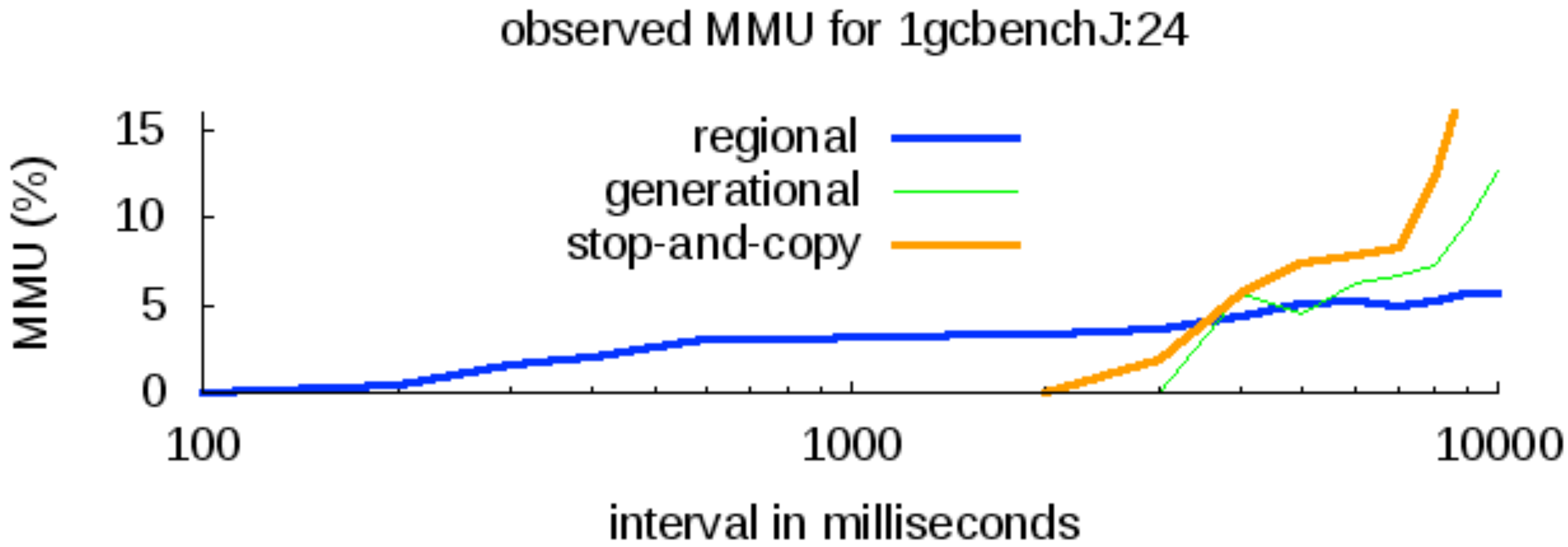

observed MMU for paraffins:25 (with L=2.5)
regional
generational
stop-and-copy
MMU (%)
1
0
100
1000
10000
interval in milliseconds

Near-worst-case Benchmarks

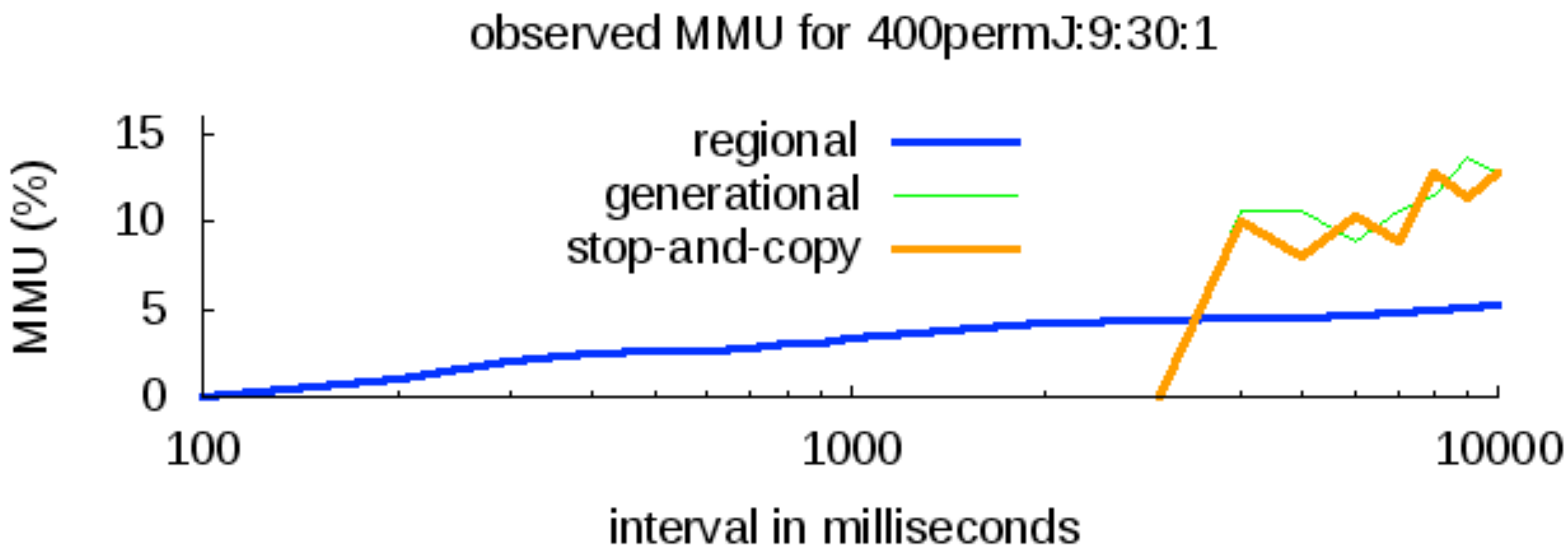
- `5gcbenchJ:24` (not `1gcbenchJ:18`)
- `400permJ:9:30:1`
- `1000queueJ:1000000:50`
- `1000pueueJ:1000000:50:50`

Longest GC Pause

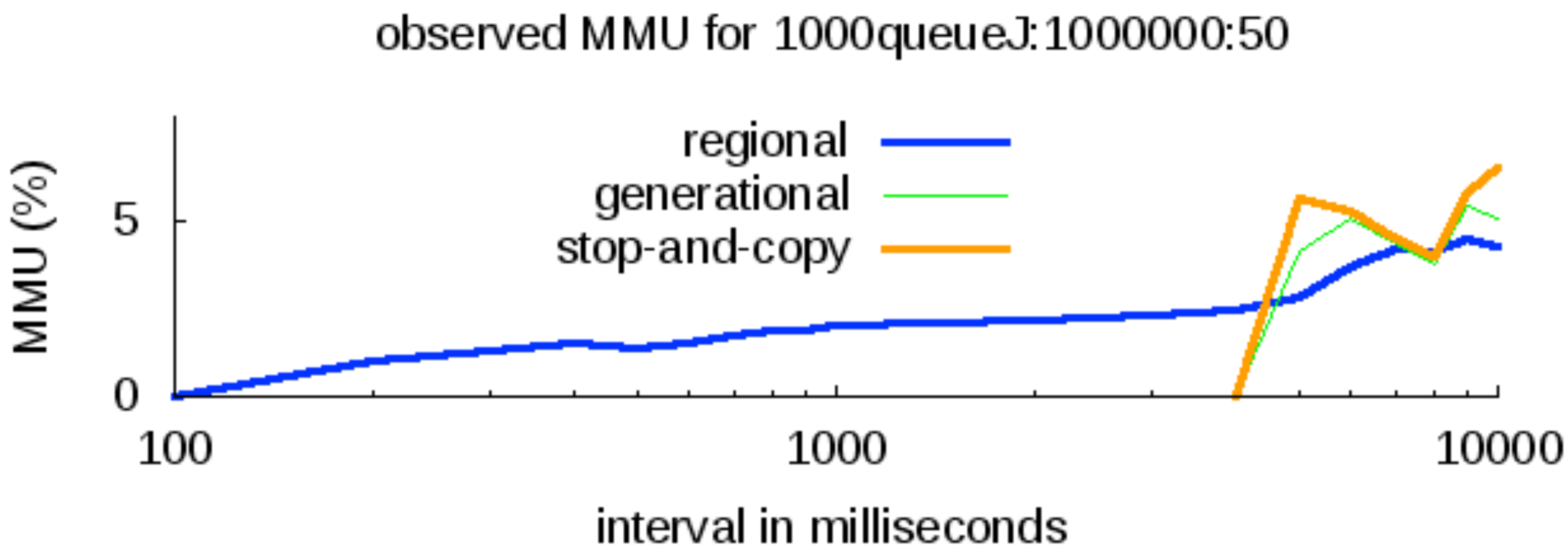
		gcbench	perm	queue	pueue
Scheme	stop&copy	2.94	3.44	4.62	4.74
Scheme	generational	3.13	3.23	4.28	4.45
Java	generational	2.78	2.93	3.24	3.32
Java	concurrent m/s	15.45	0.50	0.45	5.94
Java	garbage-first	2.13	4.68	4.29	5.84
Scheme	regional	0.12	0.13	0.09	0.21



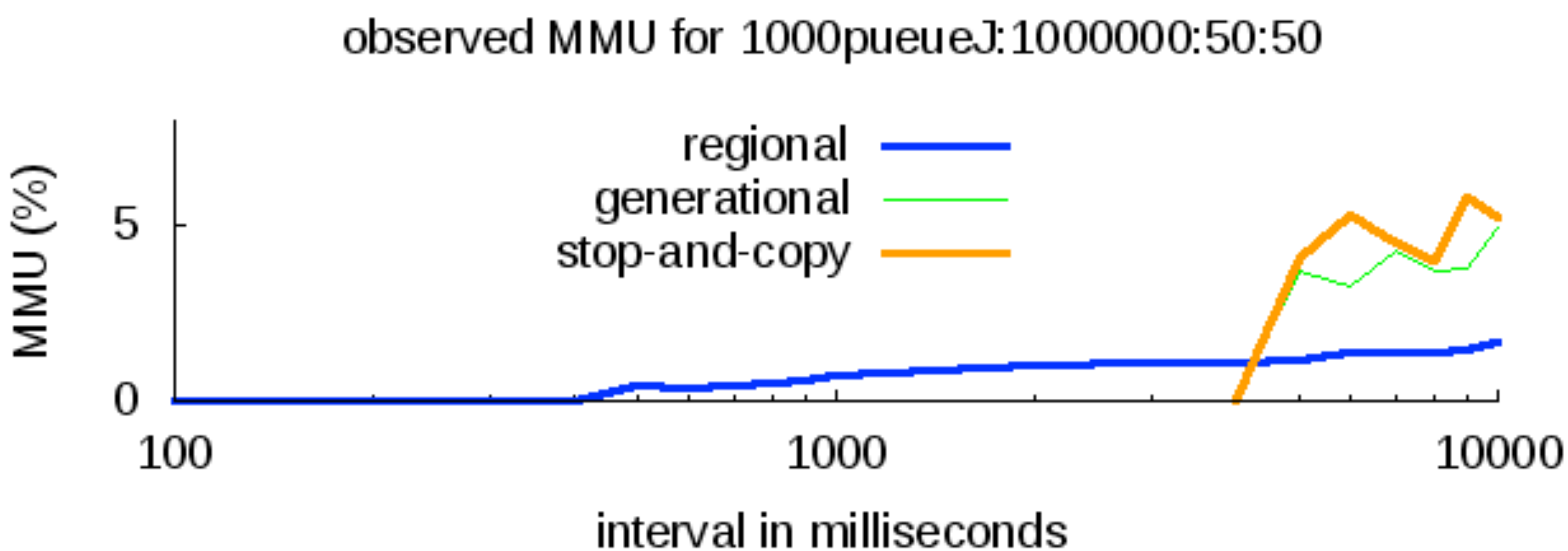
observed MMU for 1gcbenchJ:24
MMU (%)
0
5
10
15
100
1000
10000
interval in milliseconds
regional
generational
stop-and-copy



observed MMU for 400permJ:9:30:1
MMU (%)
0
5
10
15
100
1000
10000
interval in milliseconds
regional
generational
stop-and-copy



observed MMU for 1000queueJ:1000000:50
regional
generational
stop-and-copy
MMU (%)
5
0
100
1000
10000
interval in milliseconds



observed MMU for 1000pueueJ:1000000:50:50
regional
generational
stop-and-copy
MMU (%)
5
0
100
1000
10000
interval in milliseconds

Compared to G1

Pause times?

Compared to G1

Pause times? Better!

Compared to G1

Pause times? Better!

MMU?

Compared to G1

Pause times?	Better!
MMU?	Better!

Compared to G1

Pause times?	Better!
MMU?	Better!
Throughput?	

Compared to G1

Pause times?	Better!
MMU?	Better!
Throughput?	Varies.

Larceny v0.98b1

www.larcenists.org

Related Work (fundamental)

- Generational GC [Lieberman&Hewitt '83]
- Generation scavenging [Ungar '84]
- Scalability 1 & 3 [Blelloch&Cheng '99]
- MMU [Cheng&Blelloch '01]

Related Work (inspirations)

- Concurrent refinement [Detlefs et al '02]
- Garbage-first [Detlefs et al '04]
- Older-first [Clinger&Hansen '97, Stefanovic et al. '02, Hansen&Clinger '02]

Related Work (implementations)

- MarkCopy windows [Sachindran&Moss'03]
- Parallel Incremental Compaction [Ben-Yitzhak et al '02]
- Metronome [Bacon et al '03]
- Pauseless GC, C4 [Click et al '05, Tene et al '11]

Future Work

- Scalability of other algorithms
- SATB marking and summarization could be concurrent with the mutator
- VMs other than Larceny

Conclusion

- *Scalability is important*
  - no fiddling ($\exists\forall$ instead of $\forall\exists$)
  - achievable: regional collector
- Novel, elegant solutions for popularity & float
- Evaluated performance on representative and near-worst-case benchmarks

thanks

www.larcenists.org