Broom: sweeping out Garbage Collection from Big Data systems

Ionel Gog
Jana Giceva
Malte Schwarzkopf
Kapil Vaswani
Dimitrios Vytiniotis
Ganesan Ramalingam
Manuel Costa
Derek G. Murray
Steven Hand
Michael Isard

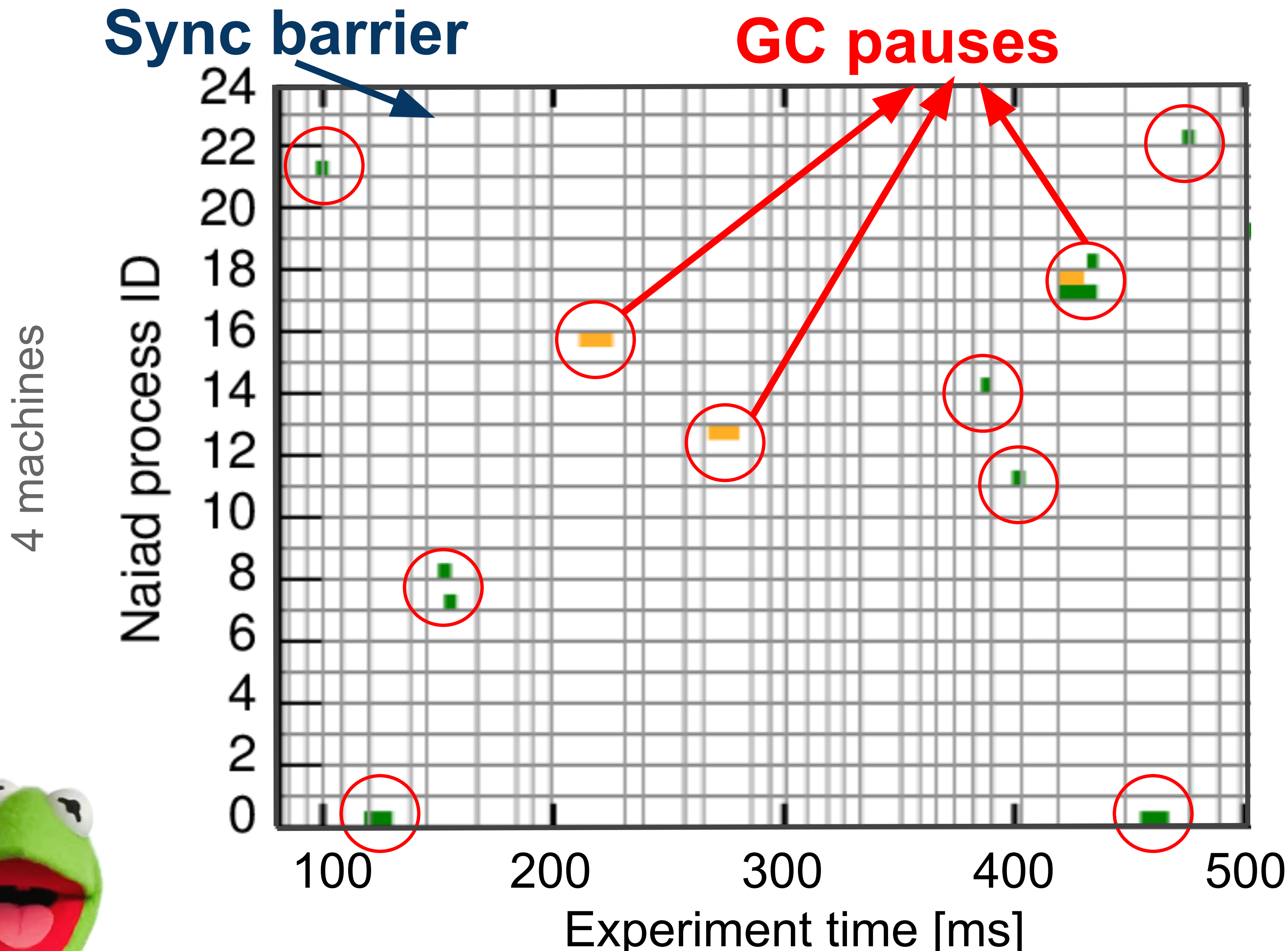
Systems@ETH Zürich
Microsoft Research

Meet Kermit from Sesame, Inc.

- Kermit runs:
  - batch computations
  - graph computations
  - incremental computations

- He uses stateful dataflow systems (e.g., Naiad, Dryad, Spark)

Incremental strongly connected components



Properties of dataflow systems

Run as a collection of actors

Communicate via message passing

Well-defined communication points

```
public class AggregateActor

  Dictionary<Time, Dictionary<K, V>> state;

  void OnReceive(Message msg, Time time)
    // Update state...
    var key = keySelector(msg);
    state[time][key] = Aggregate(state[time][key], msg)

  void OnNotify(Time time)
    Send(outgoingMsg);
    // Clear state for time...
    state.Remove(time);
```

```
public class AggregateActor

  Dictionary<Time, Dictionary<K, V>> state;

  void OnReceive(Message msg, Time time)
    // Update state...
    var key = keySelector(msg);
    state[time][key] = Aggregate(state[time][key], msg)

  void OnNotify(Time time)
    Send(outgoingMsg);
    // Clear state for time...
    state.Remove(time);
```

```
public class AggregateActor

  Dictionary<Time, Dictionary<K, V>> state;

  void OnReceive(Message msg, Time time)
    // Update state...
    var key = keySelector(msg);
    state[time][key] = Aggregate(state[time][key], msg)

  void OnNotify(Time time)
    Send(outgoingMsg);
    // Clear state for time...
    state.Remove(time);
```

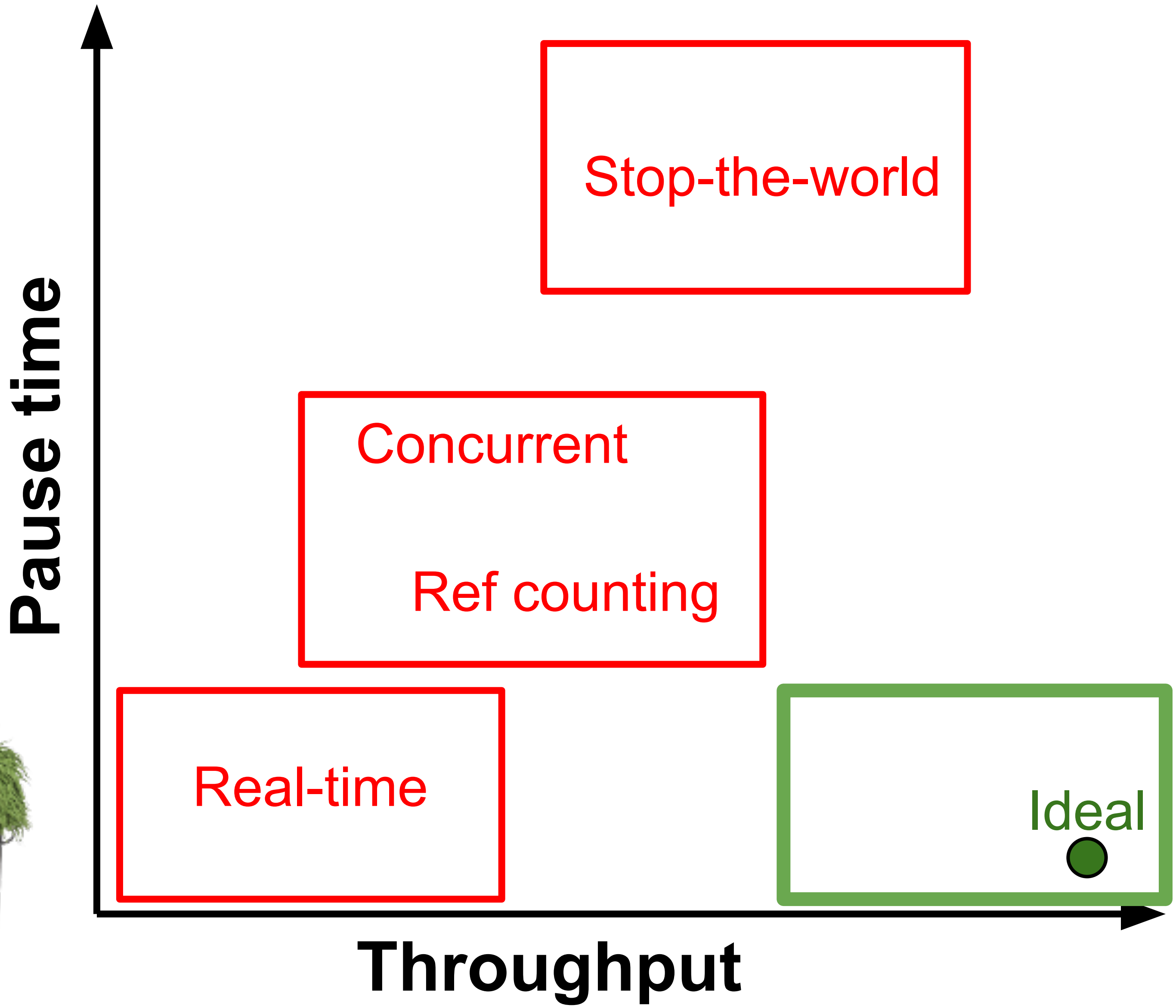
Oscar the grouch:

In-house

Garbage Collection

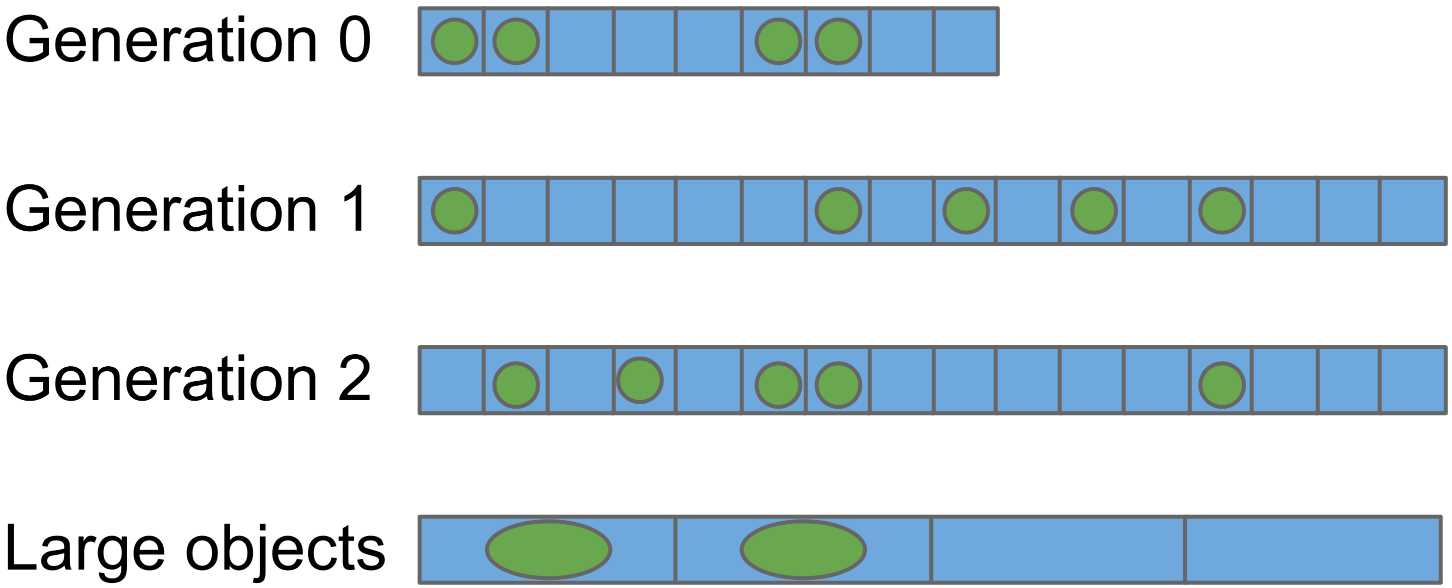
expert

Throughput vs. pause time



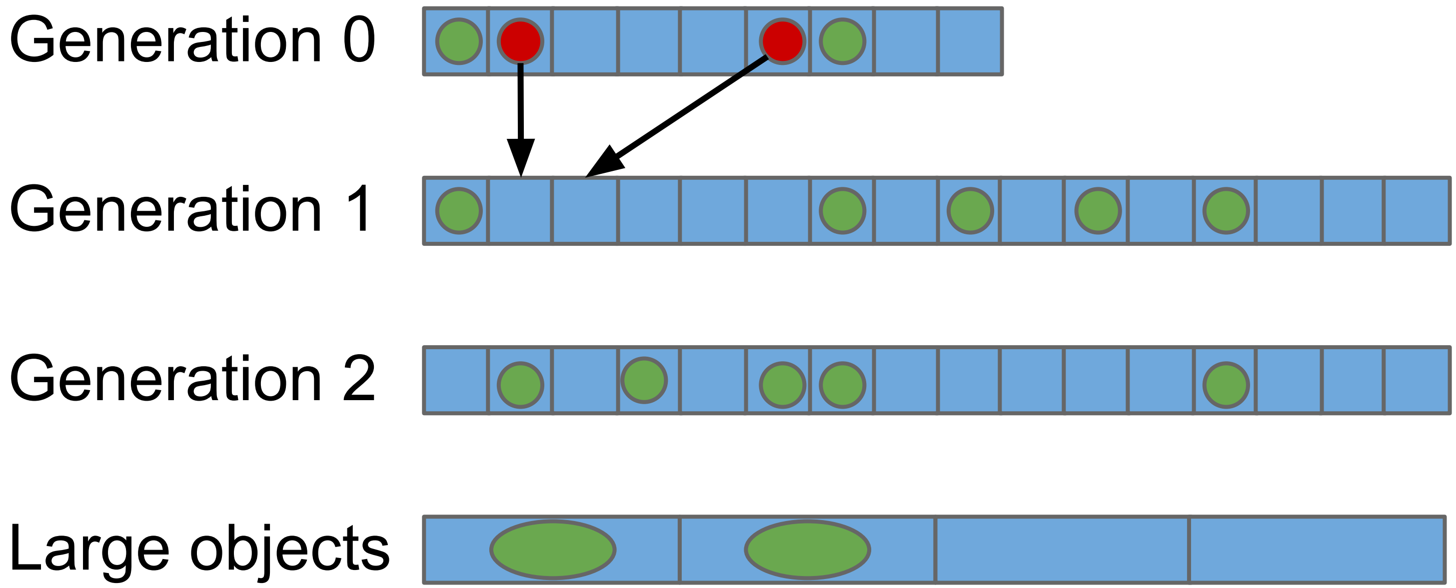
Common language runtime GC

Weak generational hypothesis:
“**most objects die young**”



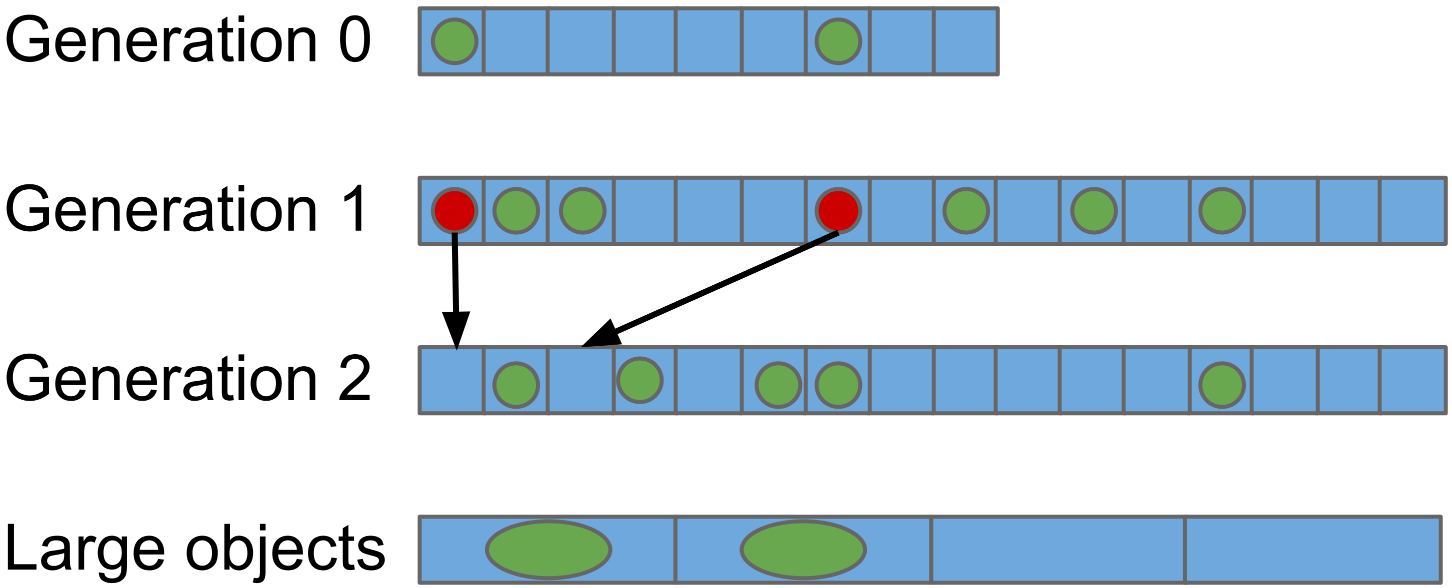
Common language runtime GC

Weak generational hypothesis:
“**most objects die young**”



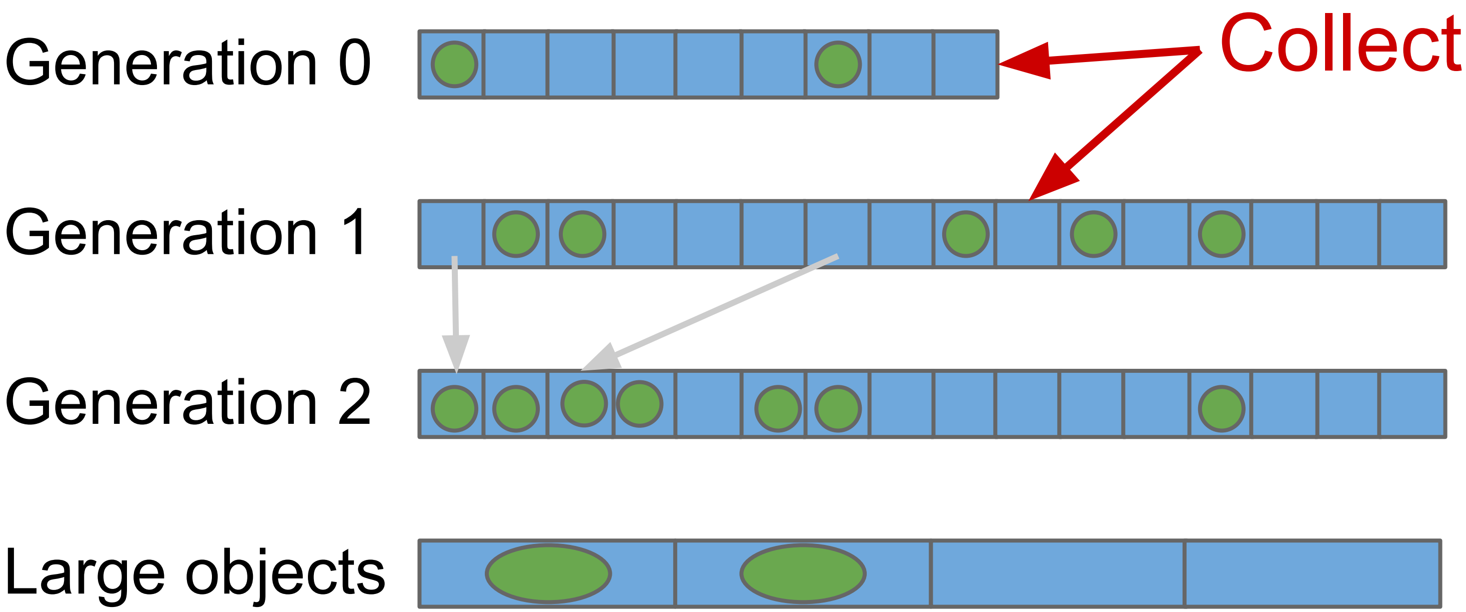
Common language runtime GC

Weak generational hypothesis:
"**most objects die young**"



Common language runtime GC

Weak generational hypothesis:
“most objects die young”



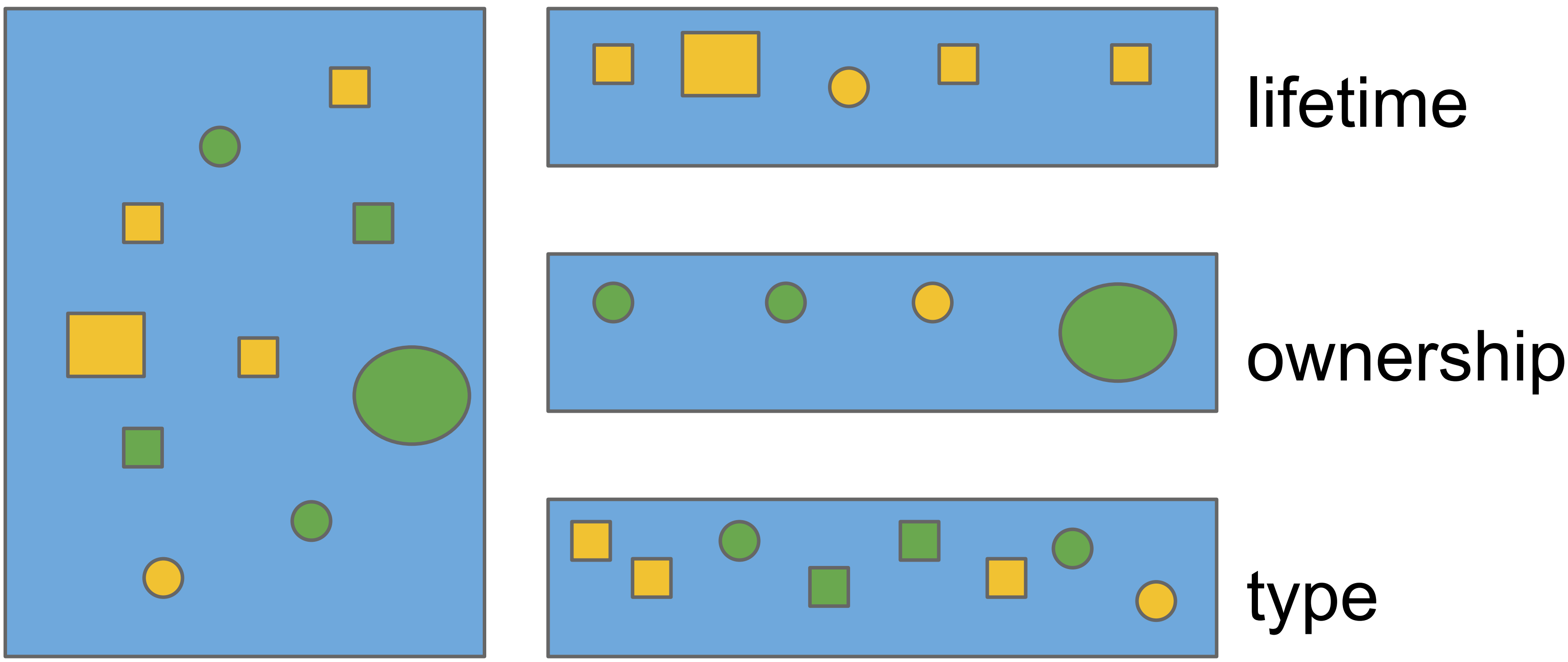
Common language runtime GC

Weak generational hypothesis:
“**most objects die young**”

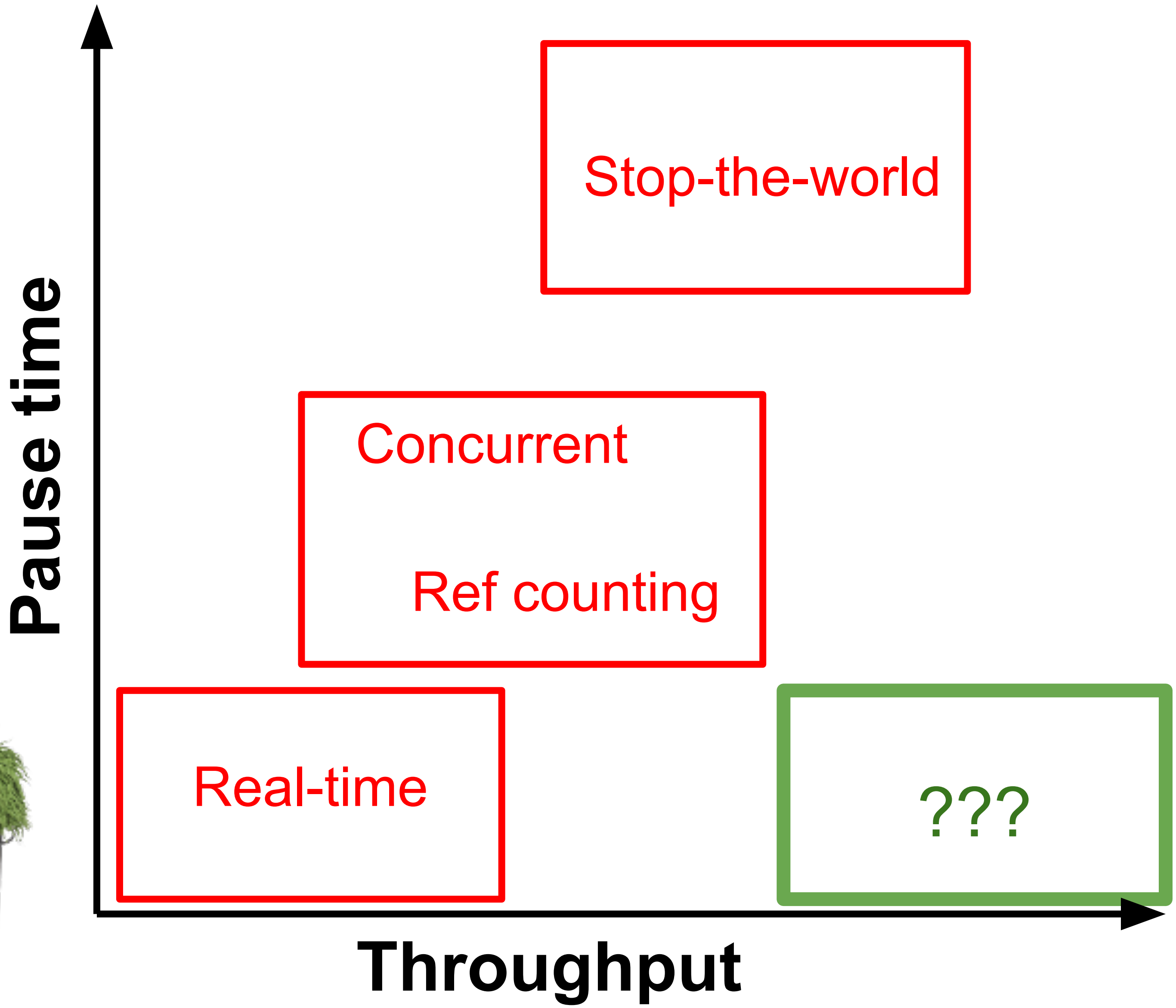
Does not hold in stateful dataflows!
(e.g. Naiad, Dryad, Spark)

why only co-locate objects based on their age?

Flexible object co-location



Throughput vs. pause time



Vmalloc: A General and Efficie
Allocator KIEM-PHONG VO

AT&T Laboratories 180 Park Avenue, Florham Park, NJ

Region-Based Memory Management

Mads Tofte

Region-Based Memory Management in Cyclone *

Dan Grossman Greg Morrisett Trevor Jim[†]
Michael Hicks Yanling Wang James Cheney

Computer Science Department
Cornell University
Ithaca, NY 14853
{danieljg,jgm,mhicks,wangyl,jcheney}@cs.cornell.edu

[†] AT&T Labs Research
180 Park Avenue
Florham Park, NJ 07932
trevor@research.att.com

ABSTRACT

Cyclone is a type-safe programming language derived from C. The primary design goal of Cyclone is to let programmers control data representation and memory management without sacrificing type-safety. In this paper, we focus on

control over data representation (e.g., field layout) and resource management (e.g., memory management). The *de facto* language for coding such systems is C. However, in providing low-level control, C admits a wide class of dangerous — and extremely common — safety violations, such as

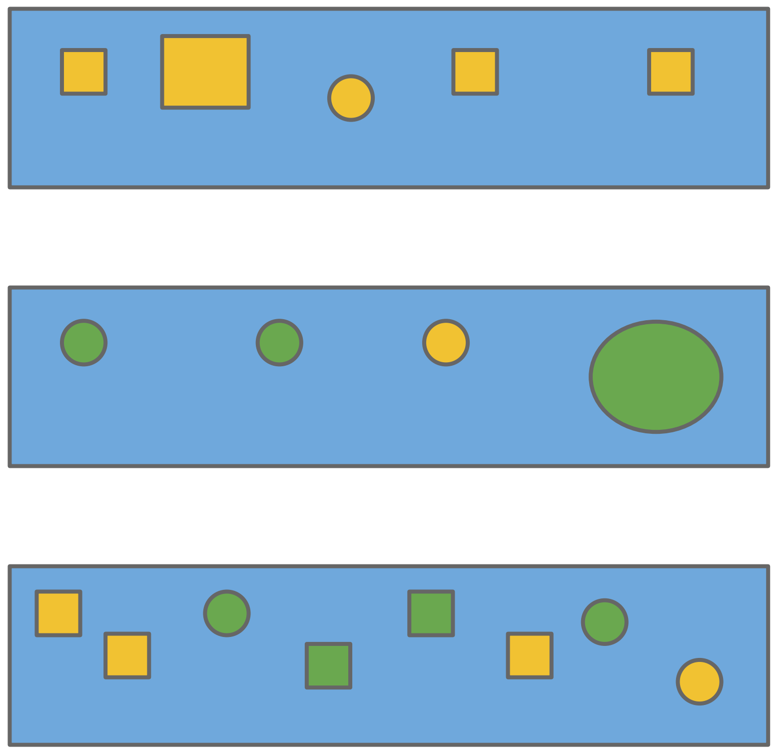
o studies of and algo-
based on garbage col-
deallocation. An al-
memory management,
t has not been well-

```
void f
{
  Regi

  for
    in
    wo
  }
```

Region-based memory management systems structure memory by grouping objects in regions under program control. Memory is reclaimed by deleting regions, freeing all objects stored therein. Our compiler for C with regions, RC, prevents unsafe region deletions by keeping a count of references to each region. Using type annotations that make the structure of a program's regions more explicit, we reduce the overhead of reference counting from a maximum of 27% to a maximum of 11% on a suite of realistic benchmarks. We generalise these annotations in a region type system whose

```
  struct finfo *sa
} *rl, *last = NUL
region r = newregi

while (...) { /* b
  rl = ralloc(r, s
  rl->data = rallo
  ... /* fill in d
  rl->next = last;
}
```

Region-based memory management



region 1 | lifetime

region 2 | ownership

region 3 | type

The goods and bads

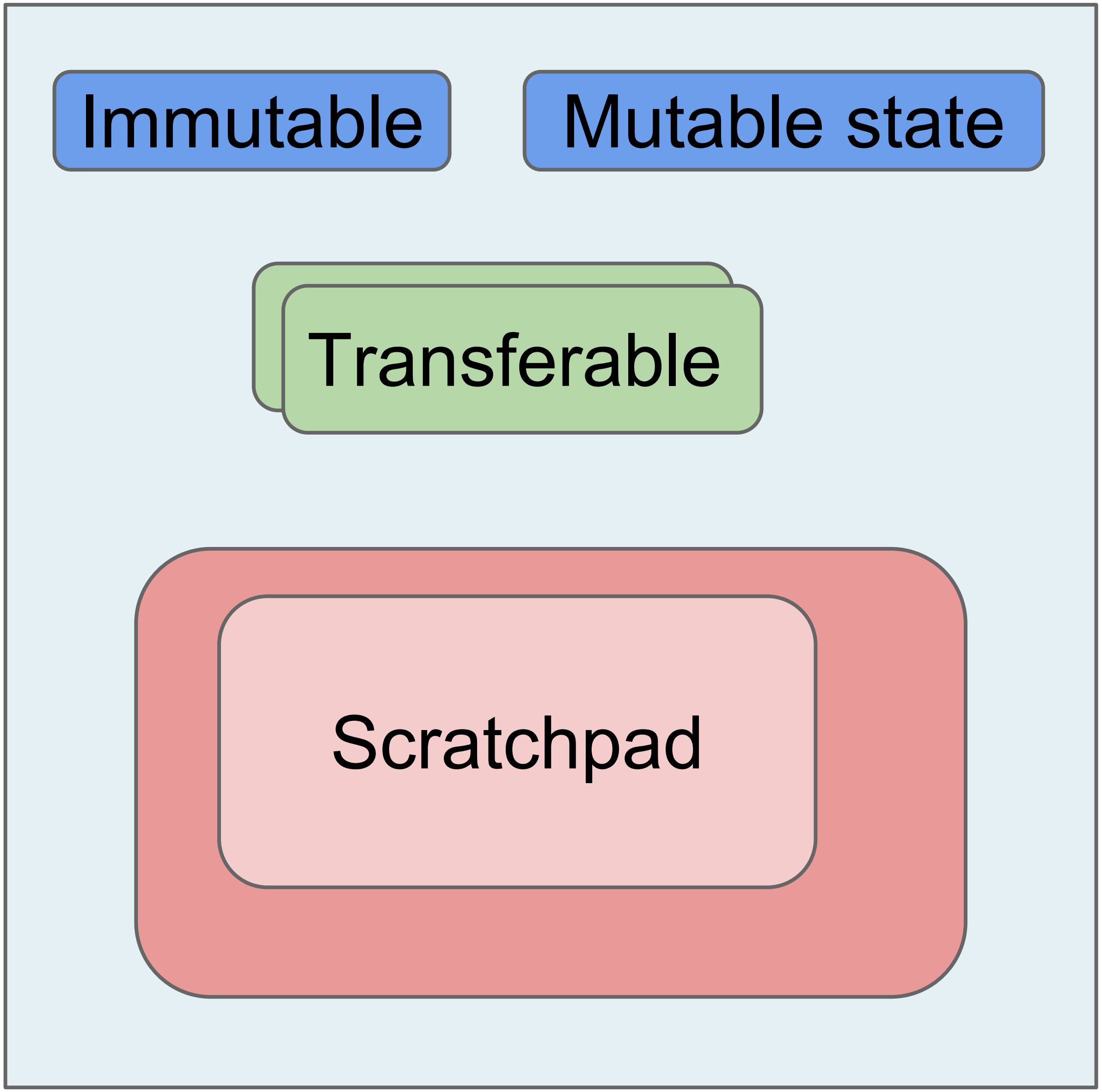
- + Decrease time spent GCing
- + Reduce runtime
- − Difficult to write programs using regions
- − Easy to leak memory
- − Trades memory usage for throughput

What's different?

We target data processing frameworks

- Stateful dataflows run as a collection of actors
- Communication done via message-passing
- Many objects have identical lifetime
- Users are not exposed to the underlying implementation

Memory usage pattern



Overview of Broom

- Three types of regions:
  - **Actor-scoped**
  - **Transferable**
  - **Temporary**
- Implemented in Bartok,
  a research compiler from MSR

Aggregate actor

```
public class AggregateActor
  Dictionary<Time, Dictionary<K, V>> state;

  void OnReceive(Message msg, Time time)
    if (state[time] == null)
      state[time] = new Dictionary<K,V>();

    var key = keySelector(msg);
    state[time][key] = Aggregate(state[time][key], entry)

  void OnNotify(Time time)
    // Clear state for time...
    Send(outgoingMsg);
```

Actor-scoped regions

```
public class AggregateActor
  Dictionary<Time, Dictionary<K, V>> state;

  void OnReceive(Message msg, Time time)
    if (state[time] == null)
```

Lifetime is identical to the actor's lifetime

```
    var key = keySelector(msg);
```

Used to store actor's fields

```
  void OnNotify(Time time)
```

Can be garbage collected

Transferable regions

Lifetime can span over the lifetime of multiple actors

```
void OnReceive(Message msg, Time time)
  if (state[time] == null)
```

Used to pass data among actors

```
  var key = keySelector(msg);
  state[time][key] = Aggregate(state[time][key], entry)
```

A region can be accessed by only one actor at a time

Temporary regions

Lifetime does not span over multiple methods

```
void OnReceive(Message msg, Time time)
  if (state[time] == null)
    state[time] = new Dictionary<K,V>();
```

Used to store temporary data

```
void OnNotify(Time time)
  Send(outgoingMsg);
```

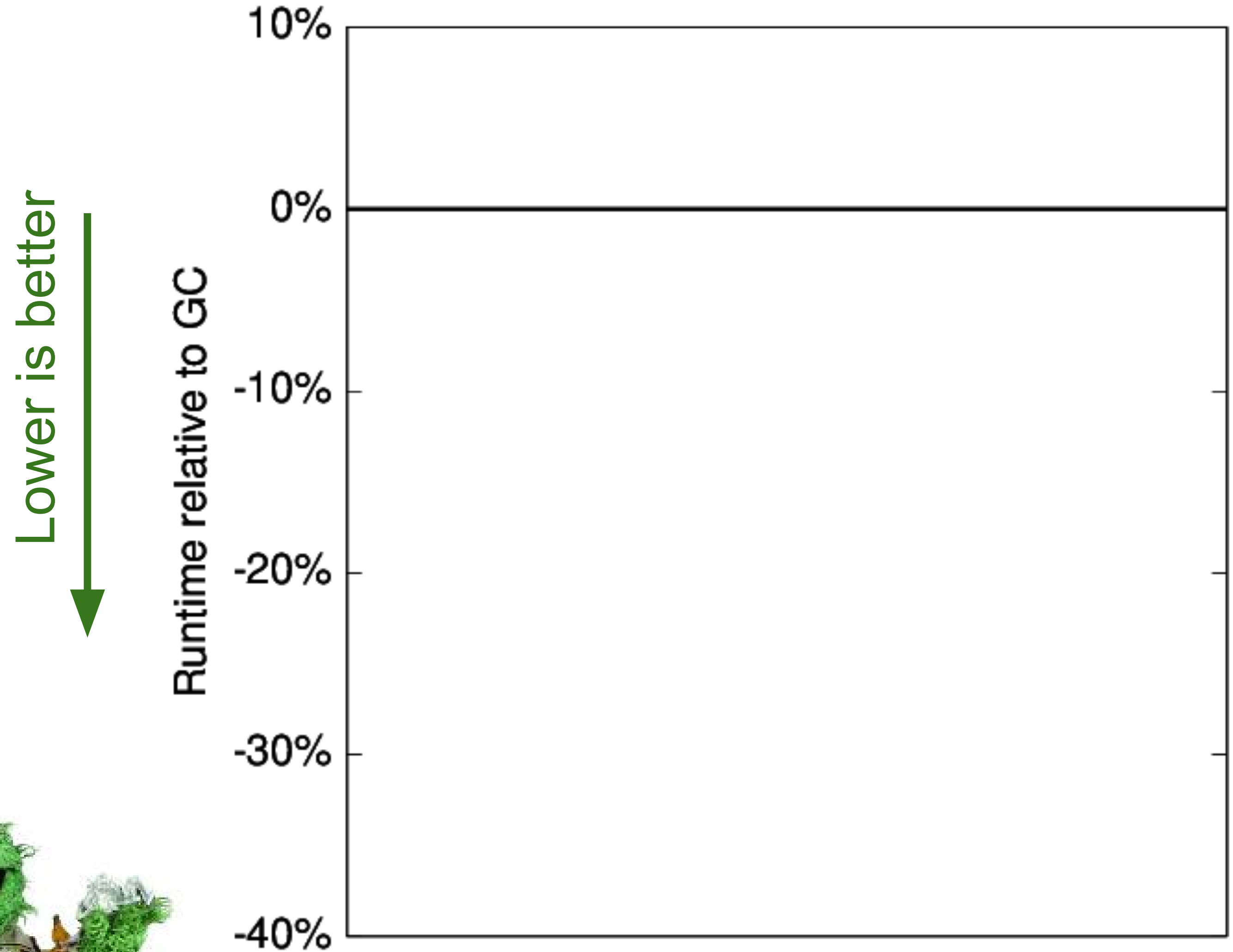
They are not garbage collected

How well does it work?

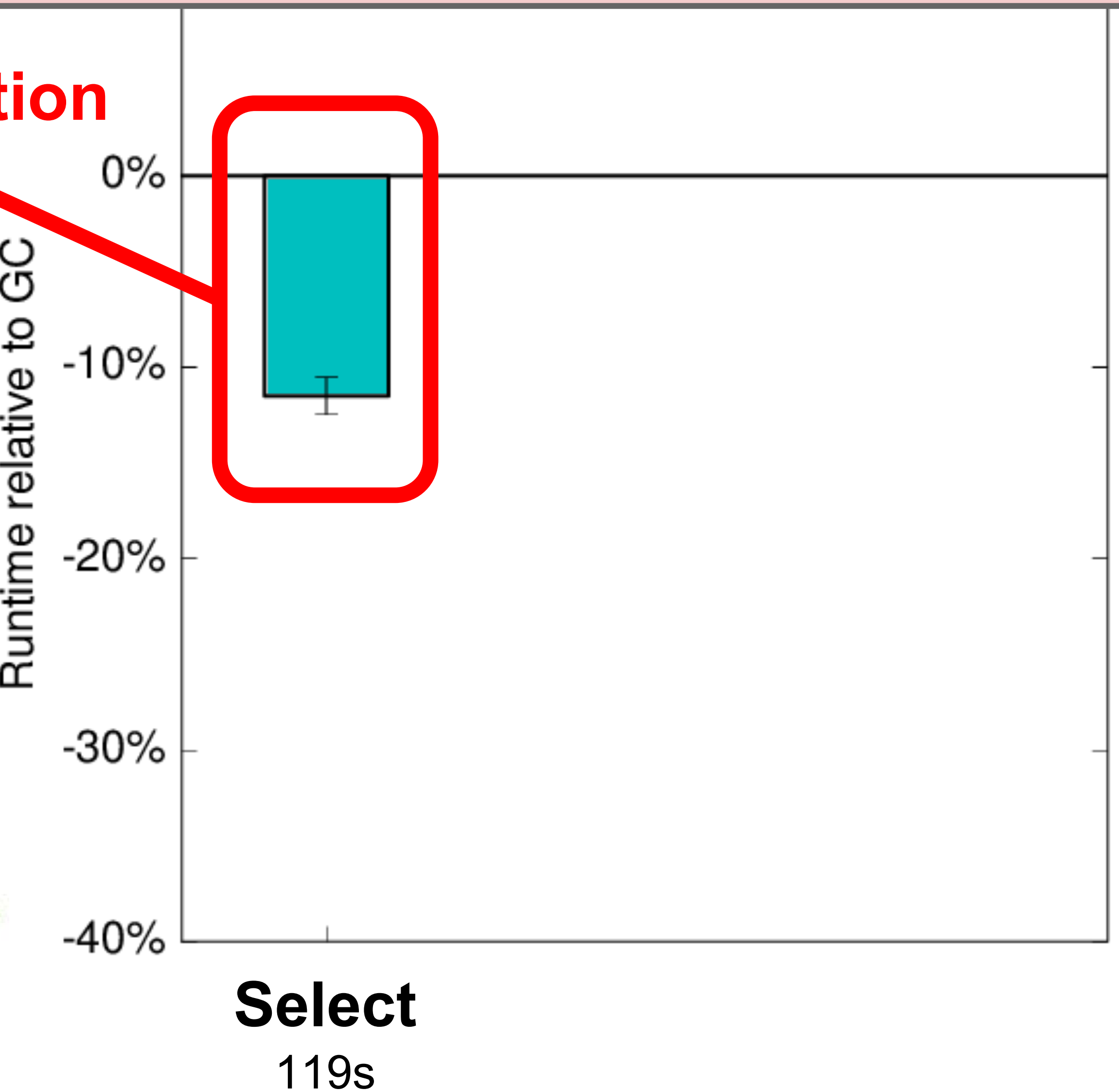
Naiad emulator

- Actor over 40 Naiad time epochs
- 500k-600k documents per epoch
- 10-20 new author entries per epoch

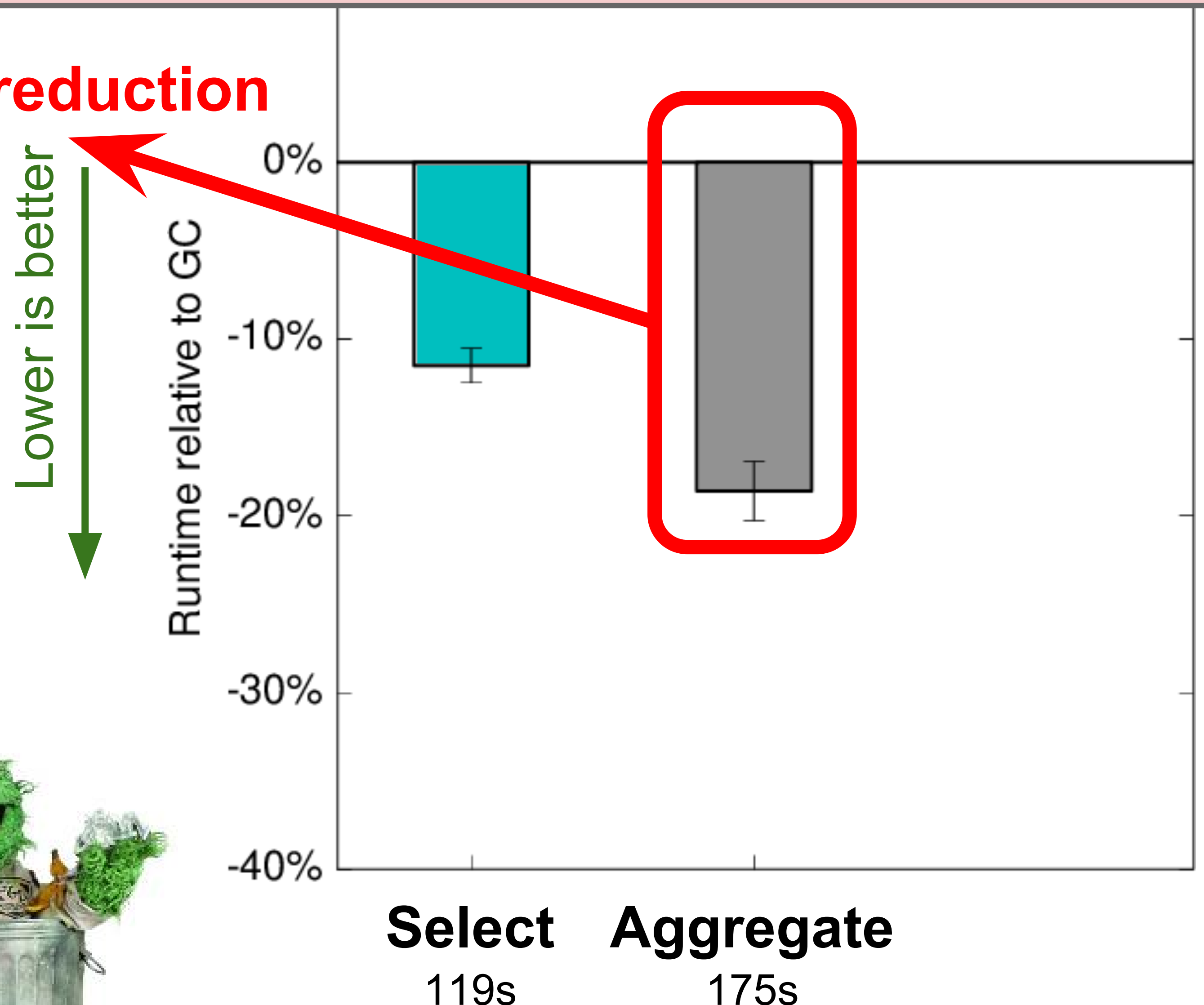
How well does it work?



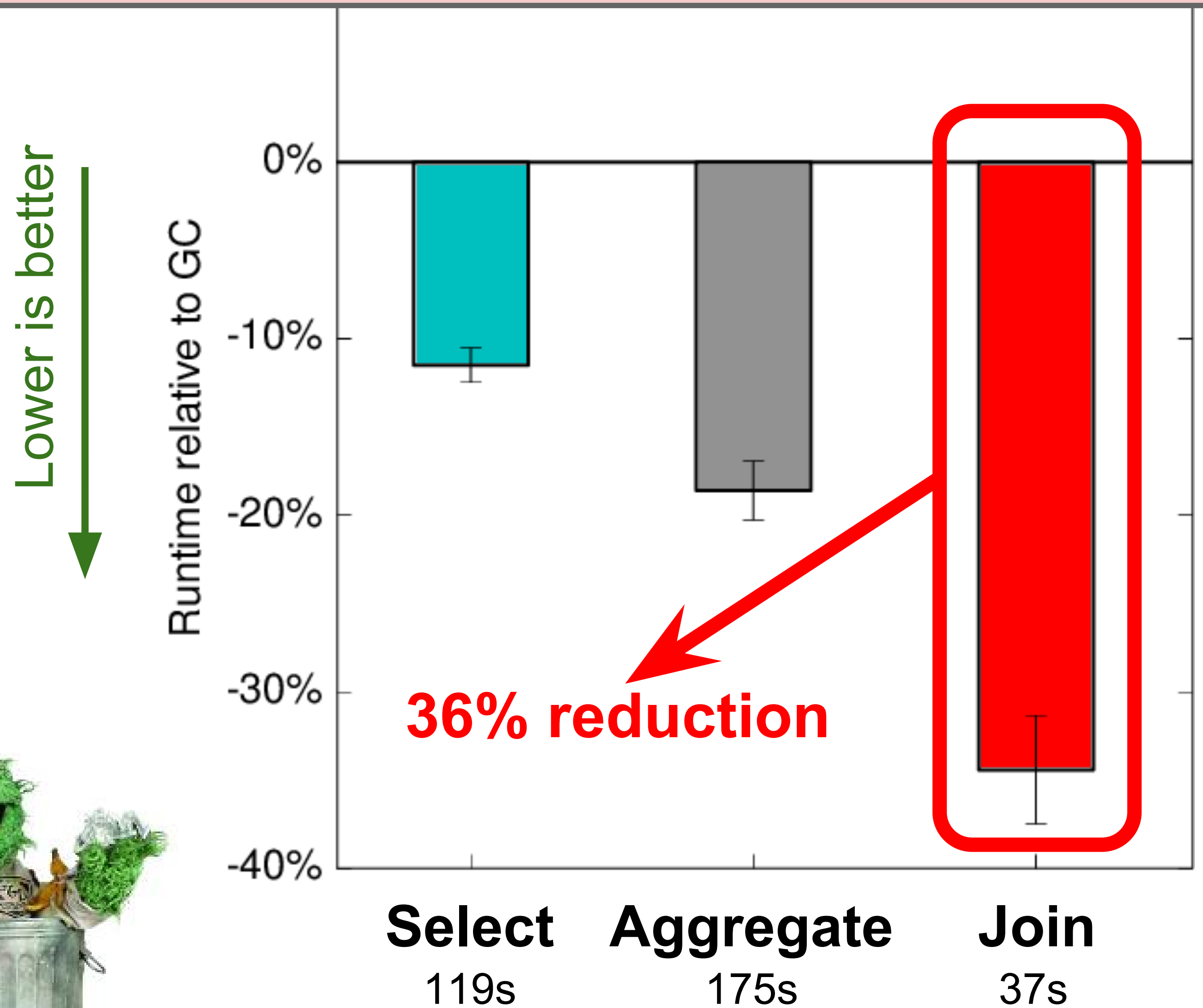
Select: stateless actor



Aggregate: stores partial aggregation results



Join: highly stateful actor



Summary and future work

- Regions work well for stateful dataflow systems
- Preliminary results show 11-36% runtime reduction
- Future work: Type safety and automatic region usage inference

@ICGog

Backup slides

Project Tungsten: Bringing Spark Closer to Bare Metal

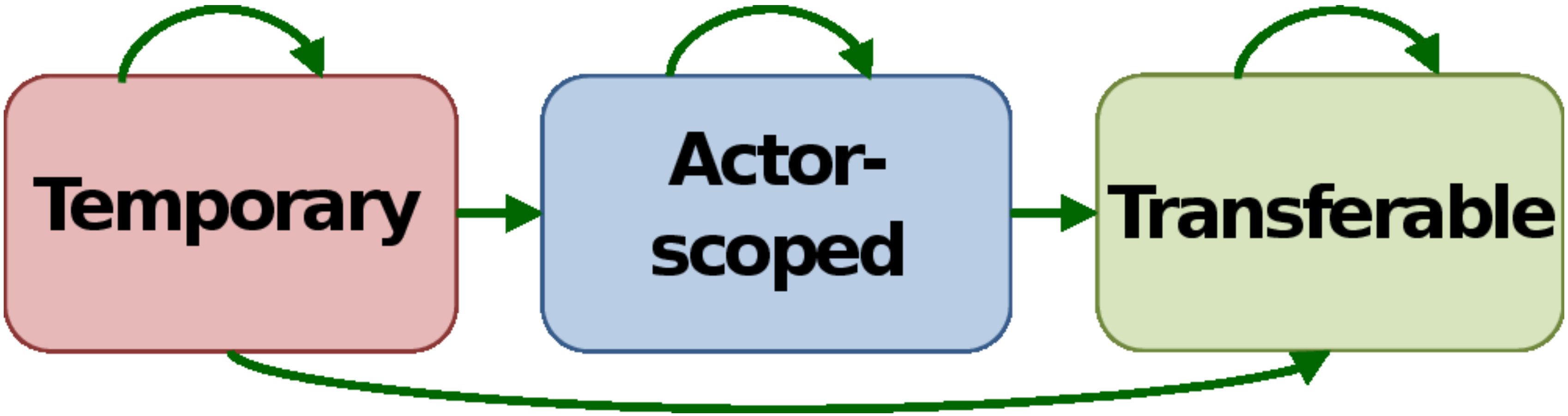
April 28, 2015 | by Reynold Xin and Josh Rosen

In a previous blog post, we looked back and surveyed performance improvements made to Spark in the past year. In this post, we look forward and share with you the next chapter, which we are calling ***Project Tungsten.*** 2014 witnessed Spark setting the world record in large-scale sorting and saw major improvements across the entire engine from Python to SQL to machine learning. Performance optimization, however, is a never ending process.

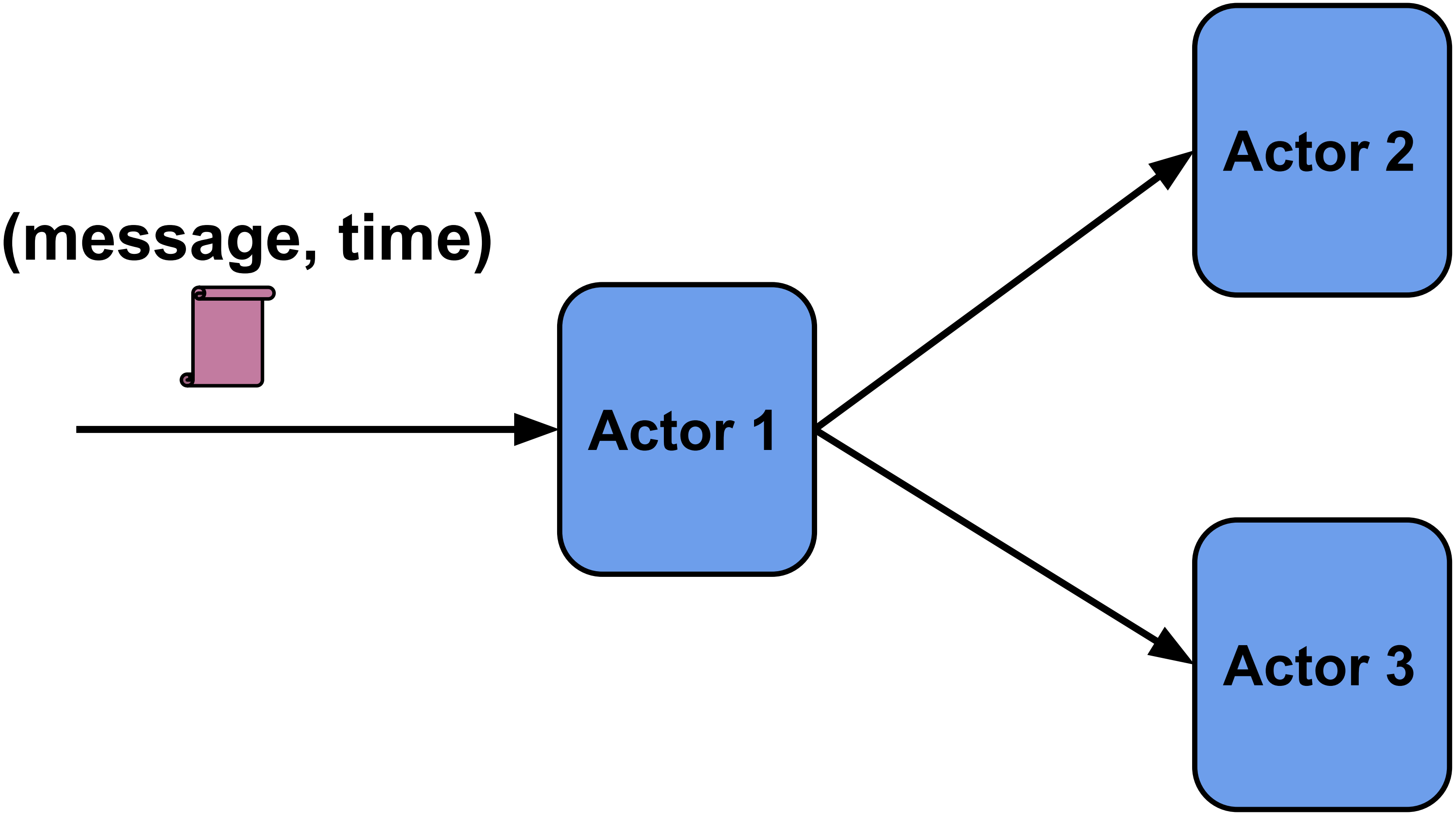
Project Tungsten will be the largest change to Spark's execution engine since the project's inception. It focuses on substantially improving the efficiency of ***memory and CPU*** for Spark applications, to push performance closer to the limits of modern hardware. This effort includes three initiatives:

1. ***Memory Management and Binary Processing:*** leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection
2. ***Cache-aware computation***: algorithms and data structures to exploit memory hierarchy
3. ***Code generation***: using code generation to exploit modern compilers and CPUs

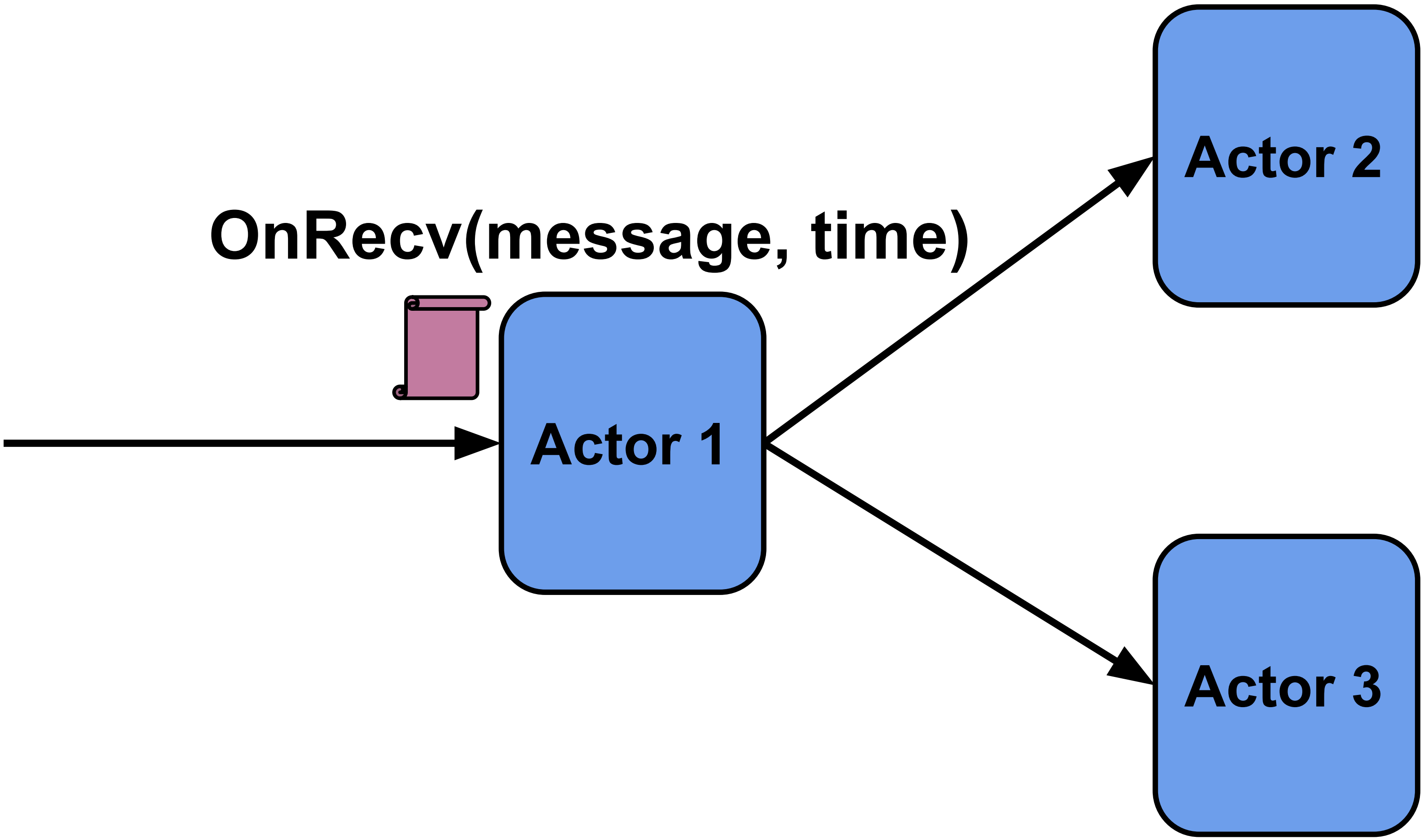
Allowed points-to relationship



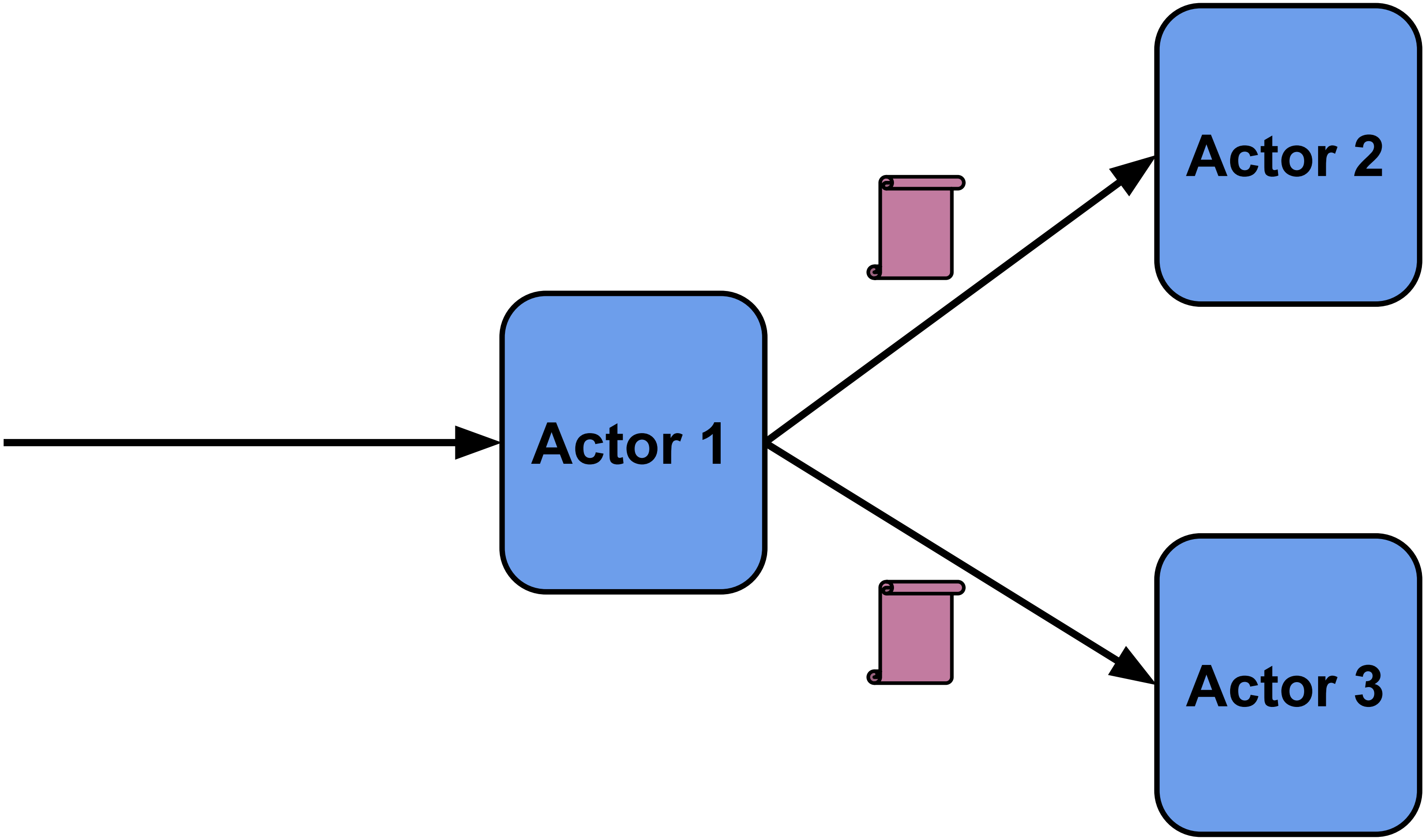
Naiad primer



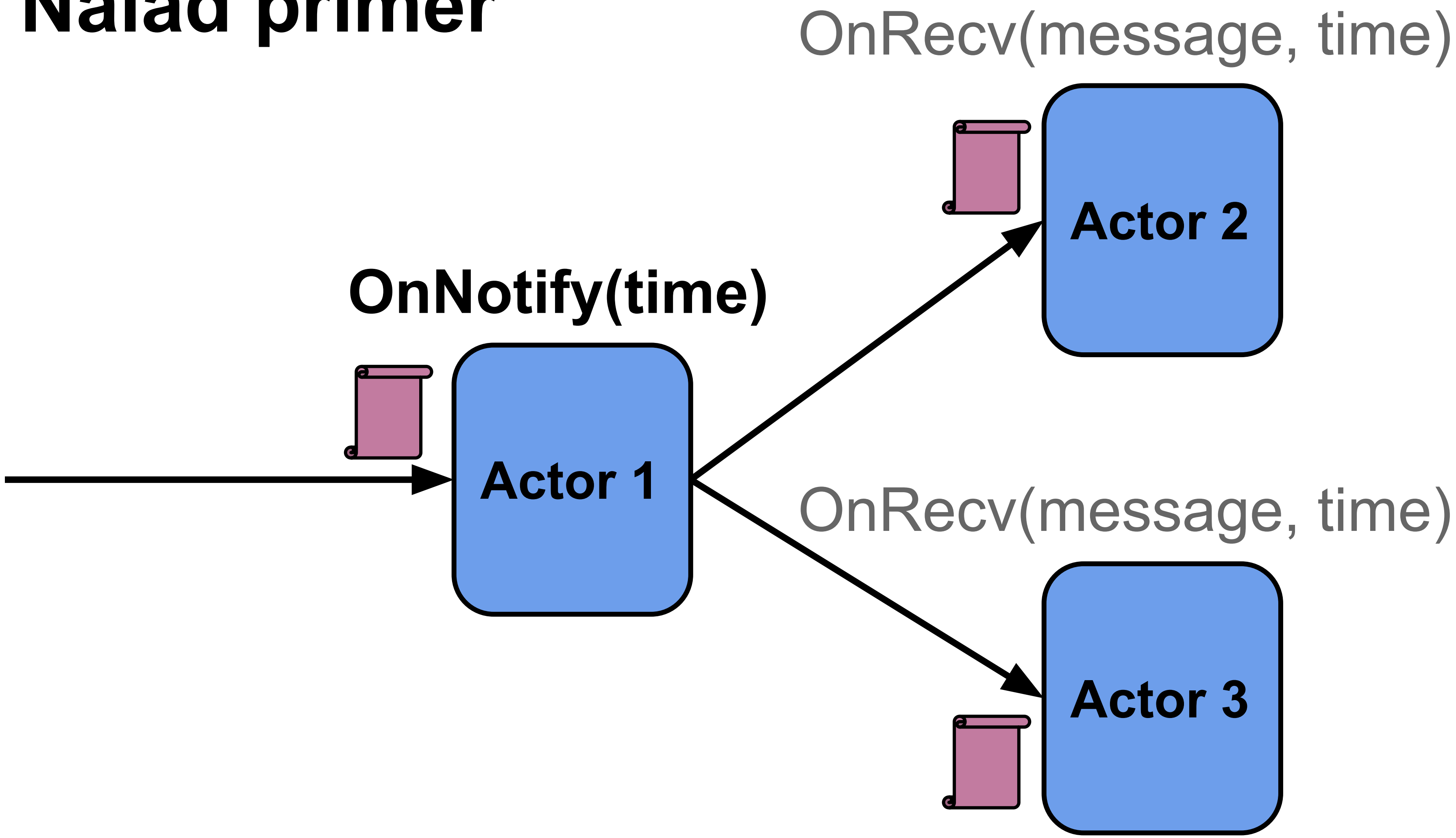
Naiad primer



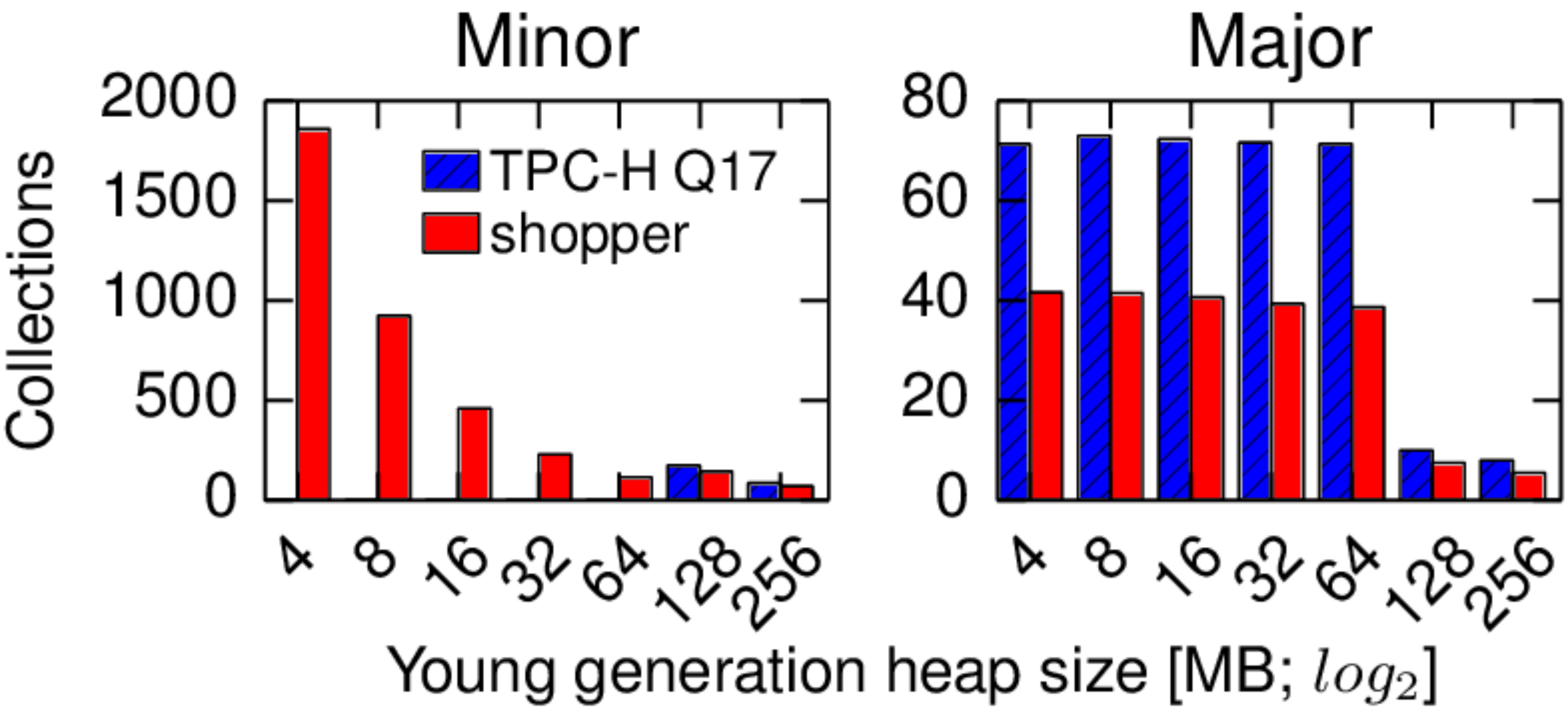
Naiad primer



Naiad primer



Minor vs. major collections



Percentage of time spent GCing