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
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

Sync barrier

GC pauses

Experiment time [ms]

4 machines

Naiad process ID
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

- Throughput
- Pause time

- Stop-the-world
- Concurrent
- Ref counting
- Real-time
- Ideal

I love trash
Common language runtime GC

Weak generational hypothesis: “most objects die young”

- Generation 0
- Generation 1
- Generation 2
- Large objects
Common language runtime GC

Weak generational hypothesis: “most objects die young”

- Generation 0
- Generation 1
- Generation 2
- Large objects
Common language runtime GC

Weak generational hypothesis:
“most objects die young”
Common language runtime GC

Weak generational hypothesis:
“most objects die young”

Generation 0

Generation 1

Generation 2

Large objects

Collect
Common language runtime GC

Weak generational hypothesis: “most objects die young”

Collect

Does not hold in stateful dataflows! (e.g. Naiad, Dryad, Spark)
why only co-locate objects based on their age?
Flexible object co-location

lifetime

ownership

type
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 improper control of references to memory. 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 main novelty is the use of existentially quantified abstract stores.
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

- Immutable
- Mutable state
- Transferable
- Scratchpad
Overview of Broom

- Three types of regions:
  - Actor-scoped
  - Transferable
  - Temporary

- Implemented in Bartok, a research compiler from MSR
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);
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);
    }
}

Lifetime is identical to the actor’s lifetime

Used to store actor’s fields

Can be garbage collected
Transferable regions

Lifetime can span over the lifetime of multiple actors

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

Used to pass data among actors

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

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

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

Send(outgoingMsg);
Temporary regions

Lifetime does not span over multiple methods

```csharp
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);
    }
}
```

They are not garbage collected

Used to store temporary data
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?

Lower is better
How well does it work?

Select: stateless actor

13% reduction

Lower is better

Runtime relative to GC

Select

119s

31
How well does it work?

**Aggregate:** stores partial aggregation results

- **20% reduction**
- Lower is better

<table>
<thead>
<tr>
<th></th>
<th>Select</th>
<th>Aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>119s</td>
<td>175s</td>
</tr>
</tbody>
</table>

- **32**
Join: highly stateful actor

36% reduction

Runtime relative to GC

-40% -30% -20% -10% 0%

Select 119s Aggregate 175s Join 37s

Lower is better
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
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

Temporary → Actor-scoped → Transferable

Temporary ← Actor-scoped → Transferable
Naiad primer

(message, time)

Actor 1

Actor 2

Actor 3
Naiad primer

OnRecv(message, time)

Actor 1

Actor 2

Actor 3
Naiad primer

Actor 1

Actor 2

Actor 3
Naiad primer

OnNotify(time)

OnRecv(message, time)

Actor 1

Actor 2

Actor 3
Minor vs. major collections

![Minor vs. Major Collections Graph]

- **Minor**
  - TPC-H Q17
  - Shopper

- **Major**

Young generation heap size [MB; $\log_2$]
Percentage of time spent GCing

![Graph showing the percentage of time spent garbage collecting (GC) as a function of young generation heap size. The x-axis represents the young generation heap size in MB, logarithmically scaled. The y-axis shows the time spent on GC in percentages. Two curves are shown: one for TPC-H Q17 in blue, and one for shopper in red. The curves indicate a decrease in GC time as the heap size increases.](image-url)