Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation

Christian Navasca[*] Cheng Cai[*] Khanh Nguyen[*]

Brian Demsky[†] Shan Lu[‡] Miryung Kim[*] Harry Xu[*]

[*] UCLA [†] UCI [‡] THE UNIVERSITY OF CHICAGO

creative commons BY-SA 4.0

October 30, 2019

1.0.0

Previous Approaches Focus on One Aspect of Overhead

Skyway (Serialization and Deserialization)

- 1.4x speedup
- 77% more network traffic

Yak (GC)

- 1.7x speedup
- 12% increased memory usage

Tungsten Processes Native Bytes, But is Limited

Instead of processing Objects, process bytes

- Removes object overhead, greatly improves performance

However, Tungsten is not general

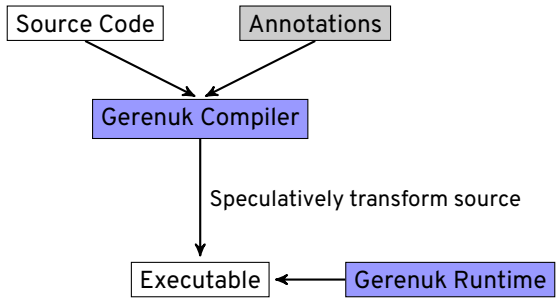
- Only for simple data types
- Adds overhead to certain applications

Can we find a scalable, general solution?

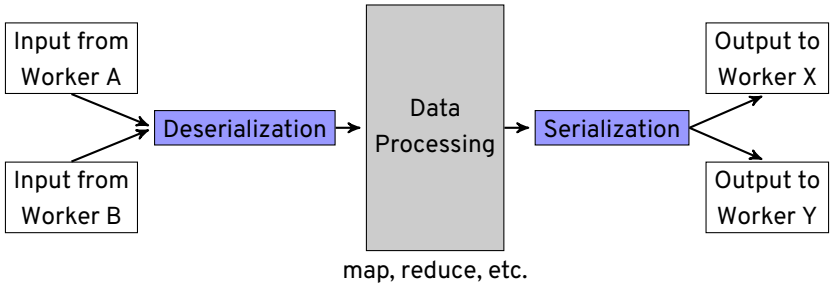
Our Solution: Gerenuk

We ran on 12 applications across two frameworks:

- Improved performance by 1.6x
- Reduced memory usage by 26%

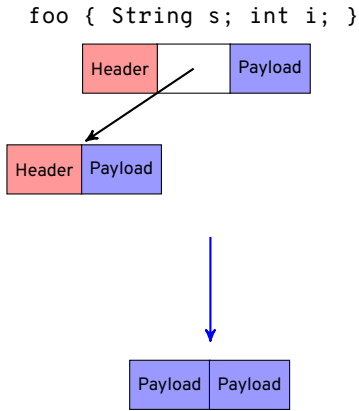


Developers Write Data Processing Applications

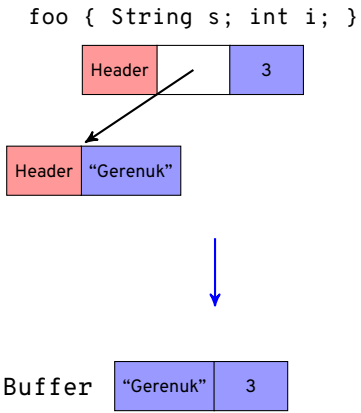


Goal: Remove Objects Through *byte inlining*

Process *inlined bytes* instead of *objects*.



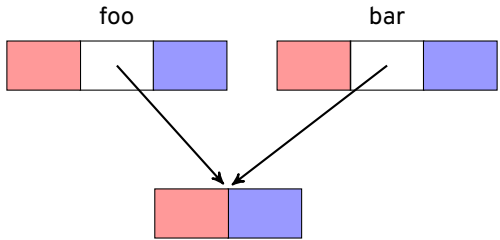
The Gerenuk Compiler Replaces Objects With Addresses




```
String s = foo.s
foo.i = 10
```

```
s = readNative(Buffer + 0, 7)
writeNative(Buffer + 7, 4, 10)
```

Byte Inlining Relies on *Confinement*

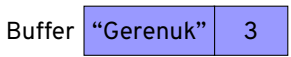


Escaping references are not allowed:

```
v = foo.s
bar = new Baz()
bar.g = v /* foo.s escapes through bar, violation */
```

In this work, an object we can't inline contains a violation

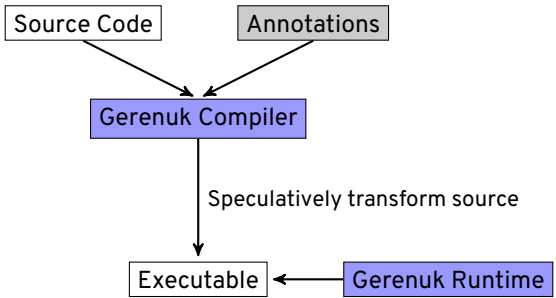
Byte Inlining Relies on *Reference-Immutability*



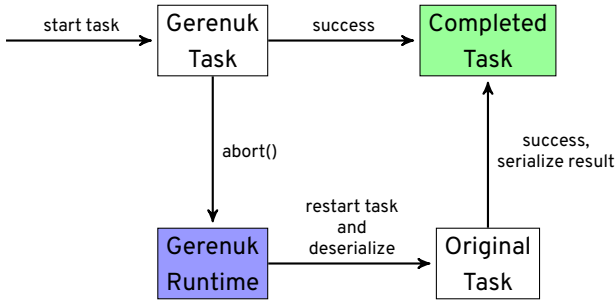
Only primitive-type assignments are allowed:

```
foo.i = 5 /* ok */
foo.s = "LongerString" /* violation */
```

Our Runtime Allows Recovery Through `aborts`



An `abort` Runs the Original Task



This is only applicable to dataflow systems (all tasks are independent)

Our Compiler Uses Static Analysis to Find Violations

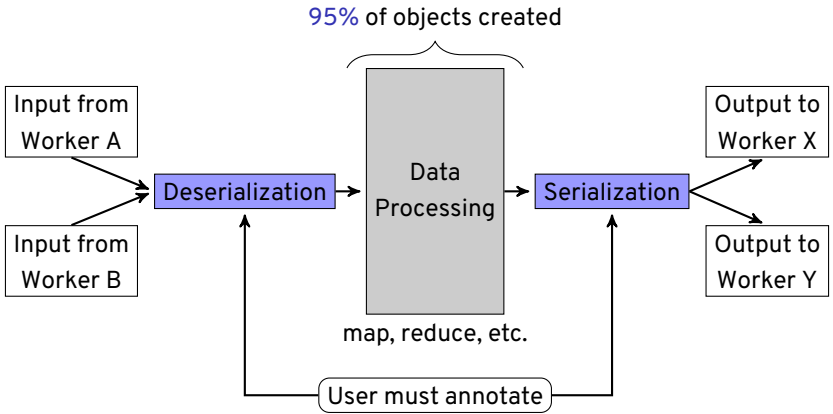
The Gerenuk Compiler inserts `abort` instructions when we detect violations

Two main challenges:

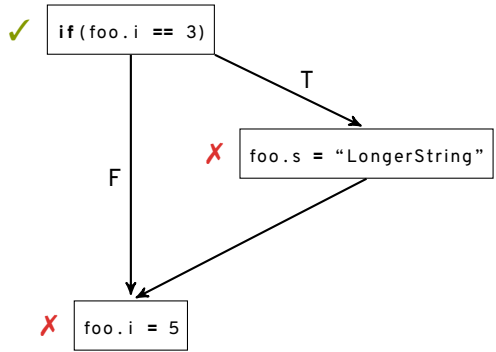
1. Scalability
2. False positives

Insight: Most of the Objects are Data Objects

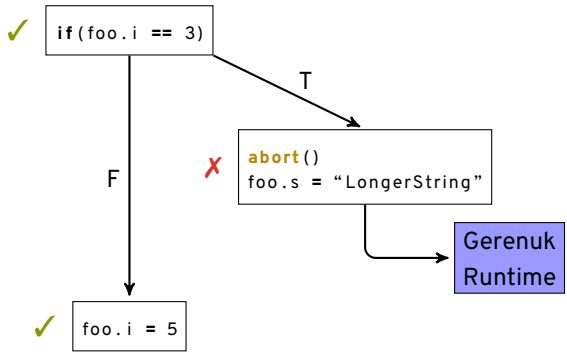
Reduce our scope to only Data Processing



Traditional Static Analysis Must Consider All Paths



aborts enable Speculative Transformation



aborts can be expensive, but should be rare

We Ran 12 Applications Across Two Frameworks

Spark

5 applications (LiveJournal, 37GB Synthetic)

Spark library applications

Hadoop

7 applications (StackOverflow, Wikipedia)

MapReduce jobs found on StackOverflow

11-node cluster, each node contains:

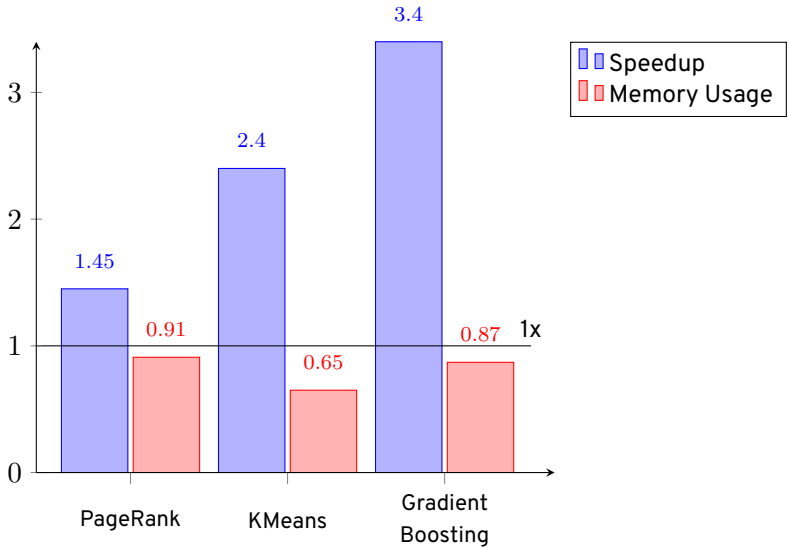
- 2 Xeon(R) CPU E5-2640 v3 processors
- 32GB memory
- 200GB SSD
- CentOS 6.9
- Connected via InfiniBand

Gerenuk Improves Runtime and Memory in Spark and Hadoop

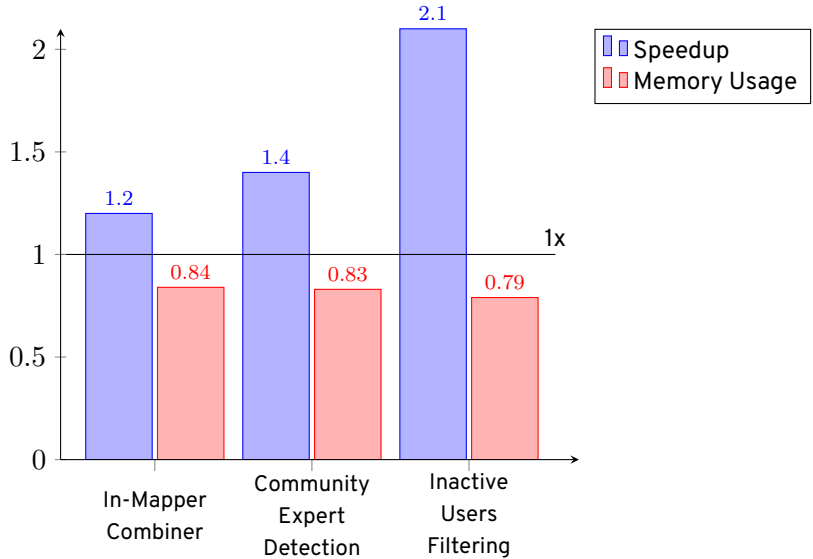
We ran on 12 applications across two frameworks:

- Spark
  - Improved performance by 2x
  - Reduced memory usage by 18%
- Hadoop
  - Improved performance by 1.4x
  - Reduced memory usage by 31%

Gerenuk Improves End-to-End Performance of Spark by 2x

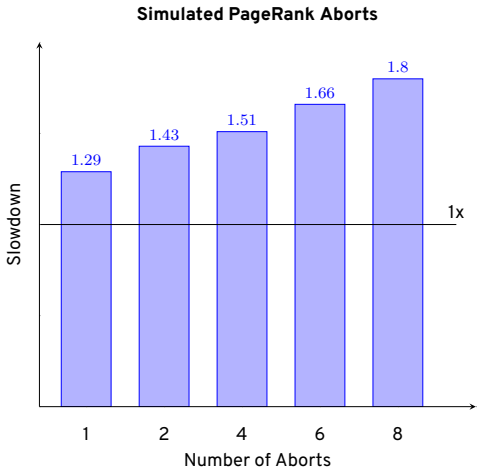


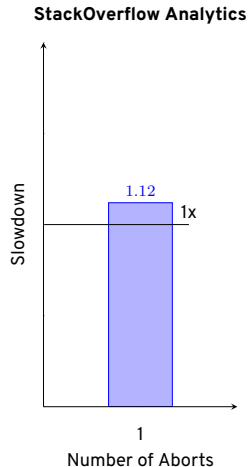
Gerenuk Improves End-to-End Performance of Hadoop by 1.4x



Violations are Costly but Infrequent

No experiments hit *abort* instructions





Summary

We present Gerenuk, which contains:

- A compiler that speculatively transforms a program
- A runtime that handles assumption violations

We ran on 12 applications across two frameworks:

- Spark
  - Improved performance by 2x
  - Reduced memory usage by 18%
- Hadoop
  - Improved performance by 1.4x
  - Reduced memory usage by 31%