An Experimental Study of Memory Management in Rust Programming for Big Data Processing

Shinsaku Okazaki
Boston University
so4639@bu.edu

Kia Teymourian
Boston University
kiat@bu.edu

ABSTRACT

Planning optimized memory management is critical for Big Data analysis tools to perform faster runtime and efficient use of computation resources. Modern Big Data analysis tools use application languages that abstract their memory management so that developers do not have to pay extreme attention to memory management strategies.

Many existing modern cloud-based data processing systems such as Hadoop, Spark or Flink use Java Virtual Machine (JVM) and taking full advantage of features such as automated memory management in JVM including Garbage Collection (GC) which may lead to a significant overhead. Dataflow-based systems like Spark allow programmers to define complex objects in a host language like Java to manipulate and transfer tremendous amount of data.

Planning optimized memory management is critical for Big Data analysis tools to perform faster runtime and efficient use of computation resources. Modern Big Data analysis tools use application languages that abstract their memory management so that developers do not have to pay extreme attention to memory management strategies.

Many existing modern cloud-based data processing systems such as Hadoop, Spark or Flink use Java Virtual Machine (JVM) and taking full advantage of features such as automated memory management in JVM including Garbage Collection (GC) which may lead to a significant overhead. Dataflow-based systems like Spark allow programmers to define complex objects in a host language like Java to manipulate and transfer tremendous amount of data.

ACM Reference Format:
Shinsaku Okazaki and Kia Teymourian. 2020. An Experimental Study of Memory Management in Rust Programming for Big Data Processing. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Multiple cluster computing analysis tools have been developed, such as Hadoop MapReduce [2], Apache Spark [3], and Apache Flink [1] [4]. These tools have brought reliable and scalable ways to deal massive data. These has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing.

These tools are constructed on top of Java Virtual Machine (JVM). JVM abstracts hardware and memory management from the developer so that the development is fairly easy. In addition, Java or Scala compiled code is platform-independent, which can run on any machine with JVM. However, these advantages may be really critical weakness when it comes to processing big data. JVM abstract away most detail regarding memory management from the system designer, including memory deallocation, reuse, and movement, as well as pointers, object serialization and deserialization. Since managing and utilizing memory is one of the most important factors determining Big Data systems' performance, reliance on a managed environment can mean an order-of-magnitude increase in CPU cost for some computations. This cost may be unacceptable for high-performance tool development by an expert.

To overcome these problems, one can use programming languages with more control on hardware, system languages, for development of Big Data tools. For example, C++ is a general-purpose, statically typed, compiled programming language which supports multiple programming paradigm. It is also a system language which gives full control over hardware. There are several researches or projects [5] where developers and researchers implement Big Data tools with this language. These tools shows significantly better performances than those developed with application languages. Although the evidence of the advantage of building high speed computational tools with C++ has been discovered, the steep learning curve and difficulty of writing memory safe codes are barrier to technology diffusion.

Rust is a system language which gives the similar performance and control of hardware to C++ or C and safety of runtime. The memory-safety, and fearless concurrently in Rust programming make the language one of the ideal candidate for development of Big Data tools. Since the design of the language is different from any other programming languages, implementations that can be selected for algorithms can differ from existing ones. In this paper, we focus on memory management strategy for Big Data processing algorithms in development with Rust.

Even though Rust can be a great candidate to develop Big Data processing tools, there are few study for development on such tools with Rust programming.

Rust has various ways to manage memory. Rust has different variable types for values allocated in sequence of memory region. Each variables take different memory representation that can produce variation of operation time on the variable types.

In addition, Reference Counting takes important role in Rust ownership concept. By using Reference Counting, a value is able

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to have multiple owners. This situation may happen quite often, when we want to acquire complex values from contiguous memory regions. Reference Counting has both advantage and disadvantage. Reference count can share data which might decrease unnecessary copy of data, but checking reference count might be a overhead.

Atomic Reference Counting is ubiquitous in Rust multithreading program. Atomic Reference Count also has similar features to Reference Counting. In addition, it is allowed to use among different threads. This may lead additional overhead from atomic operation.

As we can see, we can choose various memory management strategies in Rust programming. Therefore, we assess following research question in this thesis.

- What are better memory management strategies for complex object processing to perform faster runtime performance.
- How much impact do different variable types in Rust have in order to algorithms' runtime performance?
- How much can algorithms runtime be improved or degraded, if we use Reference Count?
- What are better memory management strategies for faster Big Data processing in Rust multithreading?
- How can we improve runtime performance of common Big Data algorithms by Rust memory management?

2 RELATED WORK

Rust is a system programming language which provides memory safety without runtime checking like GC and necessity of explicit memory de/allocation. To ensure memory safety, Rust provides restrictive coding patterns and checks lifetime of value and memory safety at compile-time. The restrictive patterns also enables a developer to write fearless concurrent code that is free of data races. Main concepts of Memory Management in Rust are ownership, move, and borrowing.

2.1 Ownership

In ownership feature or Rust, each value has a variable called owner. This owner has information about the value, such as location in memory, length and capacity of the value. For example, the object representation of Vec<i32> is shown in Figure 1. The upper boxes represent owner variable in stack frame. The lower boxes represent contiguous memory allocated to store i32. Its capacity is specified 10, but 7 values of i32 are stored. Therefore, there are still spaces to store 3 values of i32 without reallocation of memory. This owner can live on the scope associated with its lifetime. When the owner is dropped, the value will be dropped too. This feature is similar to how RAII in C++ works. However, acquisition of owner out of the scope where it was constructed is available in Rust with the concept of move.

2.2 Move

In Rust, for most types operations like assigning a value to a variable, passing it to a function, or returning it from a function do not copy the value: they move it. With move, a value can be transferred from one owner to another. The previous variable does not have ownership of the value; it is moved to a newly assigned variable. To understand how this assignment implementation is unique from

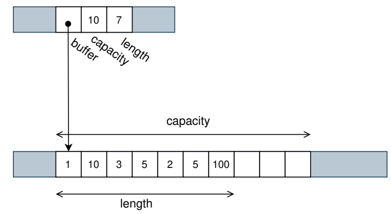

Figure 1: Representation of Rust Vec<i32>

other programing languages, Rust code example for Vec of strings are shown.

```
let s = vec!["lemon".to_string,
             "orange".to_string,
             "apple".to_string];
let t = s;
let u = s;
```

The representation of the original Vec of String in Rust is the almost same in C++ vector of string. Figure 2 however shows different behavior when Rust assigns s to t. The value is moved to t from s so that the source variable s is uninitialized. The Rust code actually throws compile error, because we are assigning uninitialized variable at the last line.

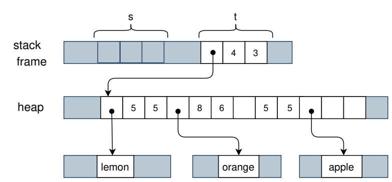

Figure 2: Representation of Rust Vec<String> after assignment to another variable

2.3 Borrowing

Borrowing lets code use a value temporarily without affecting its ownership so that it reduces unnecessary movement of ownership. One use case is when value is used in function and needed to be passed to the argument. If the argument takes ownership and the function does not return the value, the ownership of value goes out of scope and the memory is deallocated. One can pass reference of the value to the argument instead of owner. The reference goes out of scope, but ownership remains the same.

3 CONCLUSION

REFERENCES

[1] 2020. *Apache Flink*. https://flink.apache.org/

[2] 2020. *Apache Hadoop*. http://hadoop.apache.org/

[3] 2020. *Apache Spark*. http://spark.apache.org/

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[5] Jia Zou, R. Matthew Barnett, Tania Lorido-Botran, Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, and Chris Jermaine. 2018. Pliny-Compute: A Platform for High-Performance, Distributed, Data-Intensive Tool

Development. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1189–1204. https://doi.org/10.1145/3183713.3196933