B.Sc. in Computer Science and Engineering Thesis

Analysis of Immix Garbage Collector Algorithm by Rust Language

Submitted by

Raihanul Bari Tanvir
1105015

Md. Nafis-Ul-Islam
1105084

Nishat Tasnim
1105101

Supervised by

Dr. Rifat Shahriyar

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

February 2017

CANDIDATES’ DECLARATION

This is to certify that the work presented in this thesis, titled, “Analysis of Immix Garbage Collector Algorithm by Rust Language”, is the outcome of the investigation and research carried out by us under the supervision of Dr. Rifat Shahriyar.

It is also declared that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Raihanul Bari Tanvir
1105015

Md. Nafis-Ul-Islam
1105084

Nishat Tasnim
1105101

CERTIFICATION

This thesis titled, **"Analysis of Immix Garbage Collector Algorithm by Rust Language"**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in February 2017.

Group Members:

Raihanul Bari Tanvir

Md. Nafis-Ul-Islam

Nishat Tasnim

Supervisor:

Dr. Rifat Shahriyar
Assistant Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

ACKNOWLEDGEMENT

We would like to express our deepest and sincere gratitude to our thesis supervisor Dr. Rifat Shahriyar for his guidance in the development and completion of this study. He kindled our interest in the field of garbage collection and automatic memory management. He also played a vital role in improving our writing skill. His suggestions, advices, and guidance have been the most significant source of support for us. Without his continuous and whole hearted supervision, this thesis would not have been materialized.

Finally, We would like to thank our friends and family for their continuous support.

Dhaka
February 2017

Raihanul Bari Tanvir

Md. Nafis-Ul-Islam

Nishat Tasnim

Contents

List of Figures

List of Tables

ABSTRACT

Autonomous memory management system, widely known as Garbage Collection, (GC) is a very unique and challenging working sector of computer science that also offers many areas of improvement. Immix-GC, which is a mark-region type garbage collector, has an efficient marking approach and a pretty good reclamation method. But it suffers from fragmentation problem and a not so good allocation method. One crucial thing here is the choice of programming language to be used in implementing the GC. We provide an approach of improving the memory allocation method where we use Rust Programming language for implementing the Immix Garbage Collector. After detecting the scopes of memory allocation problem and fragmentation problem, our analysis results show the major issues behind these anomalies. We provide an improved approach towards memory allocation method with defragmentation. This results in an efficient memory management process in addition to the time efficiency.

Chapter 1

Introduction

In this world of multi-megabyte RAM configurations and virtual memory, memory is still a scarce and precious resource whose use must be carefully controlled. Programs are written in object oriented languages with automatic memory management for software engineering benefit and maintaining memory usage. Many high level programming language remove the burden of manual memory management from the programmer by offering garbage collection. Garbage collection is a major part of memory management. It refers to the process of deallocation of every unreachable object that is created. A fast yet resistant garbage collector is a must requirement for language runtime. For such a garbage collector, manipulation of raw memory and thread parallelism must be maintained. Therefore, we looked forward to work with the rust language for implementing garbage collector as it is defined as 'a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety'.

1.1 Research Objective

One of the important and basic features of any Garbage Collector is its speed and robustness. This was the main reason for choosing Rust for implementation of such GC. Its type, memory and thread safety features were useful for developing a robust collector. Again an efficient compile-time safety checker could avoid runtime overhead as much as possible. Our initial objective was to analyze scopes for further efficient implementation. This is where we found the fragmentation problem while freeing memory. Again there were insane objects where insane refers to the problem of threading and allocation.

Our primary objective was to find out the reasons behind these anomalies and find effective solution which could improve the performance of the GC that we based our work on [1]

For fragmentation problem, our approach was to find the addresses of the Lines marked as Live then narrow it down to the objects, so that we can reallocate them efficiently. Then we could free the entire lines with less internal fragmentation and utilizing the spaces of the mostly crowded lines.

1.2 Contributions

The principal contributions of this research are:

1. analyzing partially designed and implemented high performance immix garbage collector in Rust,
2. implementing a garbage collector in rust which is a memory and thread-safe language,
3. implementing defragmentation in the implemented immix garbage collector in Rust,
4. comparing performance of Rust and C language implementations of the designed garbage collector.

Chapter 2

Background

This section presents the garbage collection as well as Rust language terminology and features that is used, compared and explored in this research. It presents the process that is tried to be implemented and then enumerates a few key implementation details.

2.1 Garbage Collection

In computer science, garbage collection is a form of automatic memory management. It is the reclamation of chunks of storage holding objects that can no longer be accessed by a program. It was invented by John McCarthy around 1959 to abstract away manual memory management in Lisp [2,3]. Many programming languages require garbage collection as part of the language specification, for example, Java, C#, .NET ; these are said to be garbage collected languages. Languages such as C/C++ requires explicit management of memory allocation and deallocation through functions. But GC frees the programmer from manually dealing with memory deallocation. Dynamically allocated memory is managed automatically by a collector for managed languages. Thus, a program in managed languages are few times slower than a program written in an explicitly managed languages.
Memory management refers to memory allocation and memory deallocation. Memory storage or Heap's storage is classified as free space and reserved Space. Whenever an object gets allocated in the heap, the following Rules have to be followed.

- Memory allocation takes place in a continuous range of the free space.
- The order of objects in memory remains the order in which they were created, for good locality.
- There are never any gaps between objects in the heap.
- The oldest objects are in the lowest address.

Objects are determined by garbage collector (also known as collector) at run time through two steps, which are garbage identification and garbage reclamation.

- Garbage identification: Garbage identification refers to a marking phase where all the unreachable objects or garbages are marked as well as the list of all live objects are found and created. The difficulty in garbage collection is not the actual process of collecting the garbage-it is the problem of identifying the garbage in the first place. There are two techniques for identifying garbage- reference counting and tracing.
  1. Reference counting: This process keeps track in each object of the total number of references to that object. A special field to each object called reference count. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the program. If the reference count is zero, the object is garbage.
  2. Tracing: This process keeps track of the live object by tracing all the objects. The Reference tracing does not occur once the object goes out of scope. The GC starts its work when the memory in the heap is full. The live objects are marked and the unreachable object are not visited in this process. So, the objects which is not visited is identified as garbage.
- Garbage reclamation: Garbage reclamation is the process of freeing the unreachable objects or garbages. After idenfying the grabage in mark and sweep process, all of the heap memory that is not occupied by marked objects is reclaimed. It is simply marked as free, essentially swept free of unused objects. Garbage reclamation can also be done by directly returning the space used by the unreachable objects to a free list when the reference count of any object becomes zero.

2.1.1 Immix

Immix is an efficient mark-region garbage collection process. Immix has space efficiency, fast collection process and mutator performance. The heap organization in immix maintains blocks which are recyclable. Each blocks are divided into fine-grained lines. Immix blocks are generally 32KB and lines are 128B. This procedure contagiously allocate in regions i.e. in free blocks and lines. Contagious allocation results in excellent locality and for simplicity, objects cannot span regions, it spans lines. Partially marked blocks are recycled first as it maximizes the sharing of freed blocks.Free blocks are allocated at last.
Garbage identification is done just like mark and sweep. The live objects and the their containing regions are marked live and the unmarked regions can be freed. A line containing live objects as well as the object containing lines are marked live and if there exists any line which contains no live object are freed in the reclamation phase.

After this process is done, the blocks becomes free, some of them are partially freed and some are completely freed. The partially freed blocks contains free lines which is sometimes not spacious enough to be allocated for new objects. So, this creates fragmentation problem and opportunistically evacuation of fragmented blocks is done. While allocating a large object, sometimes the allocator cannot proceed with the request because there are lack of contagious free block but there exists partial free blocks. In that case, immix triggers the defragmentation process. During this process, the collector copies live objects and lines from the target blocks to the source blocks.

2.2 Rust Programming Language

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety[1].

- Zero Cost Abstraction: Rust is zero cost abstraction language which means it doesn't impose a cost over the optimal implementation of the task it is abstracting. Rust is focused on safety and speed which is accomplished through many ”zero cost abstraction”. It means abstraction costs as little as possible in order to make them work at low cost. The ownership system in rust is a significant example of zero cost abstraction.

- Move semantics: In rust, binding means to have ownership of what they are bound to. When a binding goes out of scope, Rust will free bound resources. Rust always ensures that there is exactly one binding to any given resource. When ownerships transferred from one owner to something else, it is said to be moved the thing we refer to and then the original binding will not be available to use. It does not need some sort of special annotation to move the ownership, its the default thing that Rust does. There is also an alternative to this method: Copy, which means copying trait and their ownership. Therefore it is not moved like one would assume, following the ownership rules.

- Guaranteed Memory Safety: Memory safety for rust means a programmer can program without a garbage collector and without the fear of segfaults. Memory safety bugs often come down to code access. The weapon against these bugs in Rust is ownership, a discipline for access control that systems programmers try to follow. In Rust, every value has an owning scope and passing or returning a value means transferring ownership (moving it) to a new scope. Values that are still owned when a scope ends are automatically destroyed at that point. The concept of b̈orrowingc̈omes in where a value is not intended to destroy or transfer the ownership permanently. Borrowing is used when temporary access to a vector or value is needed to be granted and then the functions can use the vector or value afterwards. Rust will also check that these leases do not outlive the object being borrowed. This references are valid in a limited scope and has two different types: Immutable reference (&T), which allow sharing but not mutation and mutable reference (&mut T), which allow mutation but not sharing.

- Threads Without Data Races: Data race is any unsynchronized, concurrent access to data involving a write. There is a data race when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized. Message passing, where threads or actors communicate by sending each other messages, is a simple style of concurrency. If a thread creates a vector, sends it to another thread, and then continues using it, the thread receiving this vector could mutate it as the thread continues running. So, the call to the function could lead to race condition or, for that matter. Instead, the Rust compiler will produce an error message on the call. Another way to deal with concurrency is by having threads communicate through passive and shared state using lock. Essentially preventing data races refers to a way that shared states between threads cannot be happened even accidentally, all (mutating) access to state has to be mediated by some form of synchronization. Sometimes, its often important that updates to different locations appear to take place atomically: other threads see either all of the updates, or none of them. In Rust, having &mut access to the relevant locations at the same time guarantees atomicity of updates to them, since no other thread could possibly have concurrent read access. Many languages provide memory safety through garbage collection. But garbage collection doesnt give you any help in preventing data races.

- Trait-based Generics: Like many other languages, Rust features generic functions, functions that can operate on many different types. Traits are the way to tell the Rust compiler about functionality that a type must provide. Theyre very similar in spirit to interfaces in Java, C#. They provide the solution to the problem that is faced. To define a trait, trait keyword is used and this declares a trait name with one method that all types that implement the trait must define. After declaration of a trait, implementations for various types can be defined and modified as required. There are some special traits. Some traits are known to the compiler and represent the built-in operations. Most notably, this includes the ubiquitous trait copy, send, add etc., which invokes the corresponding operations that occur when the related syntax is used [4].

- Pattern Matching: Instead of using complex conditions, Rust language has a keyword ”match”, which enables the programmer to match the pattern of possible options or conditions and act accordingly. Match is also an expression, which means we can use it on the right-hand side of a let binding or directly where an expression is used. One of the many advantages of match is it enforces exhaustiveness checking.

- Type Inference: Acquiring ownership through moving is sometimes expensive compared to the operation as the compiler must do some extra work for acquiring and destroying that ownership. In that case, a existing reference can be borrowed to access and use the object. Borrowing in Rust follows some rules: first, any borrow must last for a scope no greater than that of the owner. Second, there are two kinds of borrows that can be used one at a time, but not both at the same time. The two kinds of borrows are one or more references (&T) to a resource, which allow sharing but not mutation and exactly one mutable reference (&mut T), which allow mutation but not sharing.

- Minimal runtime: The Rust runtime means a collection of code which enables services like I/O, task spawning, task synchronization, message passing, task-local storage etc. Rust is usable with an extremely minimal runtime that approximates having no runtime at all. There is a small amount of initialization code that runs before the users main function. The Rust standard library additionally links to the C standard library, which does similar runtime initialization. Rust code can be compiled without the standard library, in which case the runtime is roughly equivalent to Cs.

- Efficient C bindings: Rust provides a foreign function interface (FFI) to communicate with other languages. Following Rusts design principles, the FFI provides a zero-cost abstraction where function calls between Rust and C have identical performance to C function calls. The libc crate provides many useful type definitions for FFI bindings when talking with C, and it makes it easy to ensure that both C and Rust agree on the types crossing the language boundary.

Chapter 3

GC in Rust

Unlike other high-level modern languages, Rust does not have a garbage collector. One of Rusts key innovations is guaranteeing memory safety (no segfaults) without requiring garbage collection. By avoiding GC, Rust can offer numerous benefits: predictable cleanup of resources, lower overhead for memory management, and essentially no runtime system. All of these traits make Rust easy to embed into arbitrary contexts, and make it much easier to integrate Rust code with languages that have a GC. Rust avoids the need for GC through its system of ownership and borrowing, but that same system helps with a host of other problems, including resource management in general and concurrency. The borrow checker is the part of the compiler that lets us achieve memory safety without garbage collection, by catching use-after-free bugs and the like. For when single ownership does not suffice, Rust programs rely on the standard reference-counting smart pointer type, Rc, and its thread-safe counterpart, Arc, instead of GC.

There are also projects that implement collectors in Rust for Rust. These projects use Rust as the implementation language and their focus is in introducing GC as a language feature to Rust. there are a reference counting stop-the-world GC [5] and simple mark-and-sweep GC for rust [6]. There is also an API that is developed for working as a GC support for Rust [7].

We now focus on the state of GC in rust; in other words, the key aspects of rust while implementing a high performance Garbage Collector. To incorporate with Rust specifications, we have chosen to implement Immix GC [8].

Four distinct elements of Rust as a GC implementation language are:

i The encapsulation of Address and Object Reference types

ii Managing ownership of address blocks

iii Managing global ownership of thread-local allocation

iv Utilizing Rust libraries to support efficient parallel collection.

3.1 Address Type Encapsulation

Memory managers manipulate raw memory. But there is significant importance of abstracting over both raw addresses and references to user objects [9, 10].

Such abstraction offers type safety and disambiguation with respect to implementation-language (Rust) references. Addresses and object references are two distinct abstract concepts in GC implementations. An address represents an arbitrary location in the memory space managed by the GC, while an object reference maps directly to a language-level object, pointing to a piece of raw memory that lays out an object and that assumes some associated language-level per-object meta data. Converting an object reference to an address is always valid, while converting an address to an object reference is unsafe. Abstracting and differentiating addresses is important, but since addresses are used pervasively in a GC implementation, the abstraction must be efficient, both in space and time. This abstraction adds no overhead in type size. So while the types have the appearance of being boxed, they are materialized as unboxed values with zero space and time overheads compared to an untyped alternative, whilst providing the benefits of strong typing and encapsulation.

3.2 Ownership Management

Efficient memory management in multi-threaded languages demand thread-local allocation techniques. Here a global pool is maintained for allocating raw memory and recovering it after use by thread-local allocators [11]. Once objects are allocate memory from these raw blocks, they may share all threads. Furthermore, at collection phase a parallel collector may have no concept of memory ownership.

Rusts ownership semantics is the key part of Rusts approach to delivering both performance and safety. The ownership semantics to this scenario guarantees that each block managed by GC is in a state among usable, used, or being allocated into by a unique thread. For this, block objects are created, each of which uniquely represents the memory range and meta data. The global memory pool owns the blocks and arranges them into a list of usable blocks and a list of used ones. Whenever an allocator attempts to allocate, it acquires the ownership from the usable Block list, gets the memory address and allocation context from the block , then allocates into the corresponding memory. When the thread-local memory block is full, the block is returned to the global used list, and waits there for collection. The Rusts ownership model ensures that allocation will not happen unless the allocator owns the block. During collection, the collector stakes away memory from used blocks, and classifies them as usable for further allocation if they are free.

3.3 Global Access

A thread-local allocator avoids costly synchronization. It is because mutual exclusion is ensured among allocators. Parts of the thread-local allocator data structure may be shared at collection time. But Rust will not allow for a mixed ownership model like this except by making the data structure shared. This defeats the main goal of the thread-local allocator.

This is solved by breaking the thread allocator into two parts, a local part and a global part. The thread-local part includes the data that is accessible strictly within current thread and all shared data goes to the global part. This allows efficient access to thread local data, while allowing shared thread data to be accessed globally.

3.4 Library Support for Parallel Collections

Parallelism is a very necessary feature in Garbage Collection. This mostly depends on the implementation of fast, correct, parallel work queues [12]. In a marking collector such as Immix, a work queue is used to manage pending work. When a thread finds new marking work, it adds a reference to the object to the work queue, and when a thread needs work, it takes it from the work queue.

It is a relief that Rust provides a rich selection of safe abstractions that perform well as part of its standard and external libraries (known as crates in Rust parlance). Using an external crate is as simple as adding a dependency in the project configuration, which greatly benefits code re-usability. The use of standard libraries is deeply problematic when using a modified or restricted language subset, as has been commonly used in the past [13–15].

Mainly two crates are used in Rust, std sync mpsc, which provides a multiple producers single-consumer FIFO queue, and crossbeam sync chase_lev , which is a lock-free Chase-Lev work stealing deque that allows multiple stealers and one single worker [16].

The parallel collector starts single-threaded, to work on a local queue of GC roots; if the length of the local queue exceeds a certain threshold, the collector turns into a controller and launches multiple stealer collectors. The controller creates an asynchronous mpsc channel and a shared deque ; it keeps the receiver end for the channel, and the worker for the deque. The sender portion and stealer portion are cloned and moved to each stealer collector. The controller is responsible for receiving object references from stealer threads and pushing them onto the shared deque, while the stealers steal work from the deque, do marking and tracing on them, and then either push the references that need to be traced to their local queue for thread-local tracing or, when the local queue exceeds a threshold, send the references back to the controller where the references will be pushed to the global deque. When the local queue is not empty, the stealer prioritizes getting work from the local queue; it only steals when the local queue is empty. Using those existing abstract types makes implementation straightforward and robust.

3.5 Abusing Rust

Since Rust has a very restrictive semantics, to achieve the goal of implementing an efficient collector, these restrictions are bypassed, which is known as Abusing Rust. Most of the time, the collector design is adaptive to take the full advantage of Rusts safety and performance. However, there are a few places where Rusts safety model is too restrictive. This can only be overcome by diving into unsafe code, where the programmer bears responsibility for safety, rather than Rust and its compiler.

During Allocation Phase, the line mark table, which is the storage for line addresses and their state value mark, may be accessed by multiple allocator threads, exclusively for the addresses that they are allocating into. Since every allocator allocates into a non-overlapping memory block, they access non-overlapping elements in the line mark table. If we were to create the line mark table as a Rust array of u8 , Rust would forbid concurrent writing into the array just as its nature. To bypass this, keeping the strictness of Rust in mind are to either break the table down into smaller tables, or to use a big lock on the whole table. The later is impractical.

Again, during collection phase, two collector threads may race to mark adjacent lines, or even the same line. The naive approach ensures that to set the line to live and storing to a byte is atomic. However, in Rust, it is strictly forbidden to modify a shared objects non-atomic fields without going through a lock.

This is overcome by generalizing the line mark table as an AddressMapTable and then wrapping the necessary unsafety into the AddressMapTable implementation which almost entirely comprises safe code. Here we rely on the Rust compiler to generate an x86 byte store which is atomic. Otherwise, there are reasonable compiler optimizations that could defeat the correctness of the code [17]. The nightly Rust releases would allow to use a relaxed atomic store to achieve the goal. This exposes a shortcoming in Rusts current atomic types where we desire an AtomicU8 type, along the lines of the existing AtomicUsize . This need is reflected in the recently accepted Rust RFC #1543: Add more integer atomic types [18].

Chapter 4

Analysis

4.1 Heap Size vs. Number of calls to GC

Heap size and number of calls to garbage collection is inversely proportional to each other. As heap size increases the number of calls to garbage collector reduces. We have varied the size of the heap space. Additionally we added a hash table that contains the object (its address to be exact) as key and true/false binary value as value. True indicates that it is marked by the GC in its tracing part and false means it is not. So after a call to GC, an object with false Boolean either indicates a garbage or a new object that has been allocated by a mutator after the GC call. The graph is shown in Figure 4.1 and Figure 4.2.

Table 4.1: Comparison with GC calls, marked true and false objects in hash vs. heap size

HeapSize (MB)	Number of times GC is called	Total hash size	Marked false in hash table	Marked true in hash table
75	5	2625374	2562619	62755
85	5	2986333	2816925	169408
100	4	3501972	3486884	15088
130	3	4555870	4425928	129942
160	2	5596373	5596231	142
220	1	7693098	7693035	63
300	1	10483200	10476124	7076
350	1	12231168	12188796	42372
400	1	13977600	13892042	85558
450	0	15333862	15333862	0

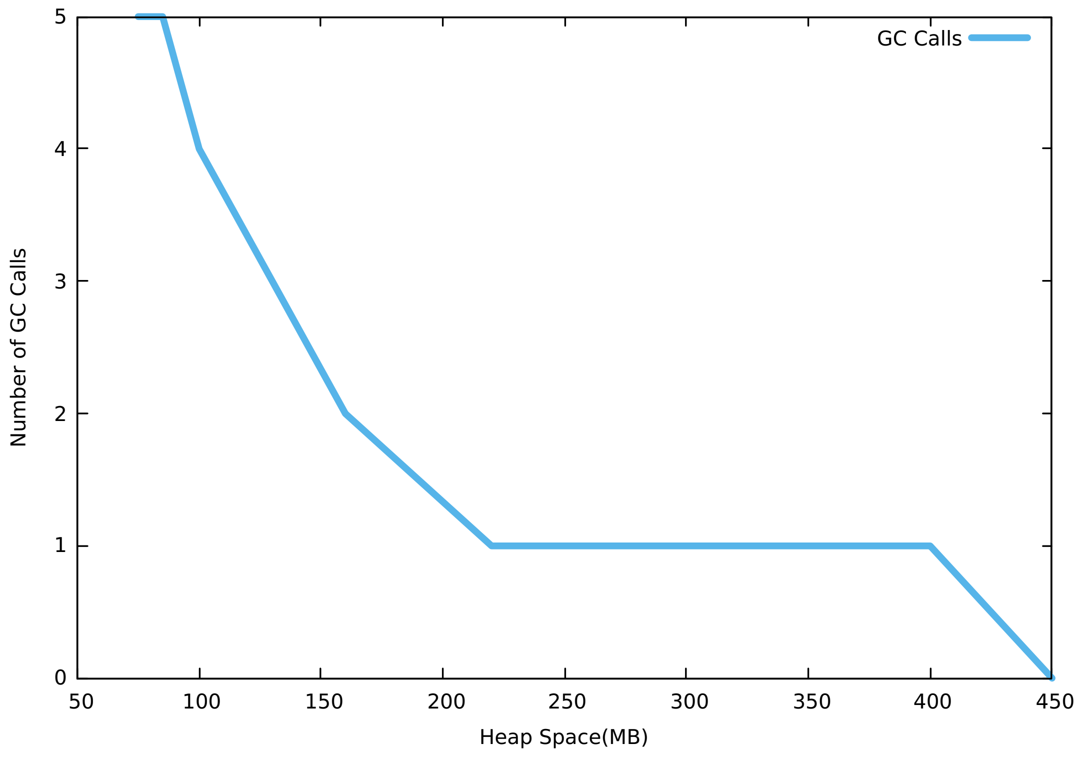

Figure 4.1: Heap Space Vs. True and False Marked Objects

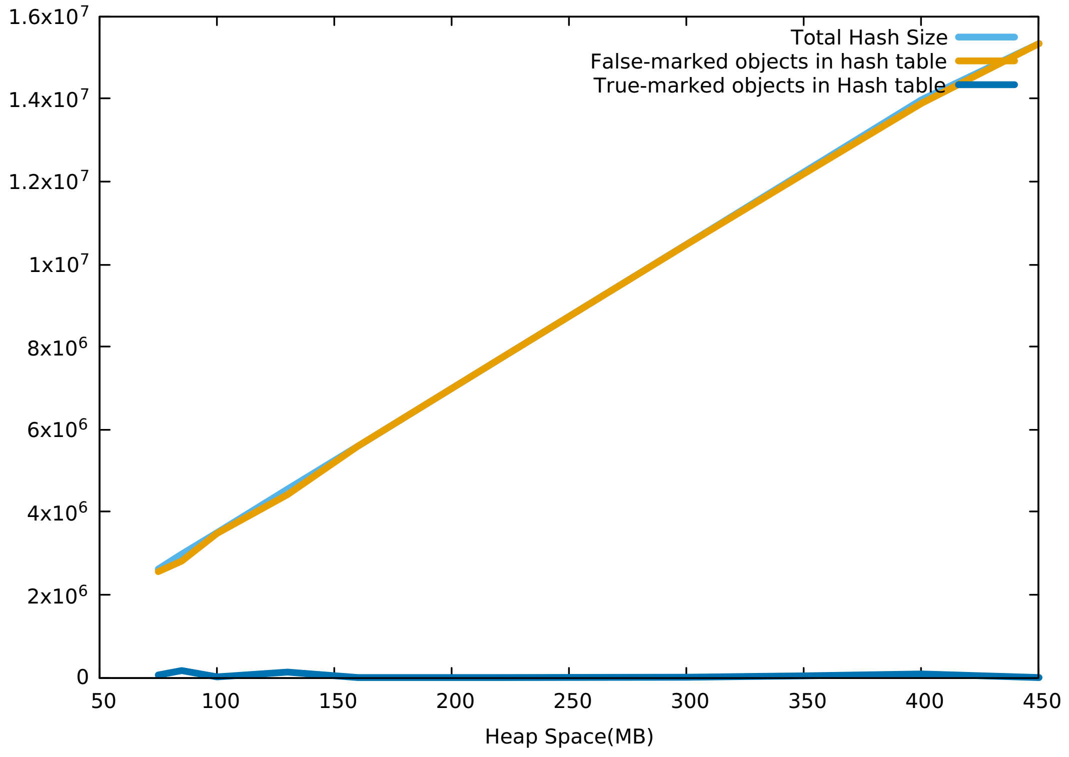

Figure 4.2: Heap Space Vs. Number of GC Calls

4.2 Used Blocks and Usable Blocks

In the immix garbage collection, we know that the heap space is divided into chunks of area known as blocks. Similarly, blocks are divided into smaller chunks of area called lines [immix 2008 paper ref]. In the implementation of immix garbage collection, the blocks are maintained as two lists. One is the list of used blocks and another is the block of usable blocks. From the name it can be understood that list of used blocks contain the blocks where the live objects are contained. Similarly, the list of used blocks contains the blocks that free at the moment. During the process of reclamation, what it does is take the block out of used blocks list and insert it into usable block list. So, usable blocks list can be thought as a free list. We have listed counts of true-marked objects and false-marked objects under usable and used blocks list under varying heap space.

Table 4.2: Comparison with true marked and false marked objects with varying heap space.

heap space	true-marked objects in total	true found in used blocks	true found in usable blocks	false-marked objects in total	false found in used blocks	false found in usable blocks
75	34441	32833	1630	2648730	2287266	359670
85	71844	62412	9471	2962037	507081	2453273
100	8211	5416	2816	3525362	1381169	2142720
130	135243	128426	6871	4439068	1735687	2702259
160	390	95	309	5620173	4168965	1450510
220	4142	3984	170	7700049	7664789	35332
300	9535	5467	4076	10475190	4851775	5624515
350	18705	13663	5053	12222179	3112891	9111025
400	89235	84666	4608	13893772	1359977	12536144
450	0	0	0	15333862	15337296	0

In here we can see that, the number of true marked in usable and used block list and false marked objects in used blocks is zero. Which means, for the heap size = 450MB, no calls to GC were made and the objects in the hash tables were not marked and they are false and contained in the blocks which are in used blocks list.

Also there is an aberration that has a high presence in here. The total number of true-marked or false-marked objects turns out to be somewhat less than the summation of object numbers of that type in used and usable block lists. It seems that some object is being counted twice or more than twice in the used block and/or usable blocks. We will refer to them as insane objects. The graph is shown in Figure 4.3 and Figure 4.4.

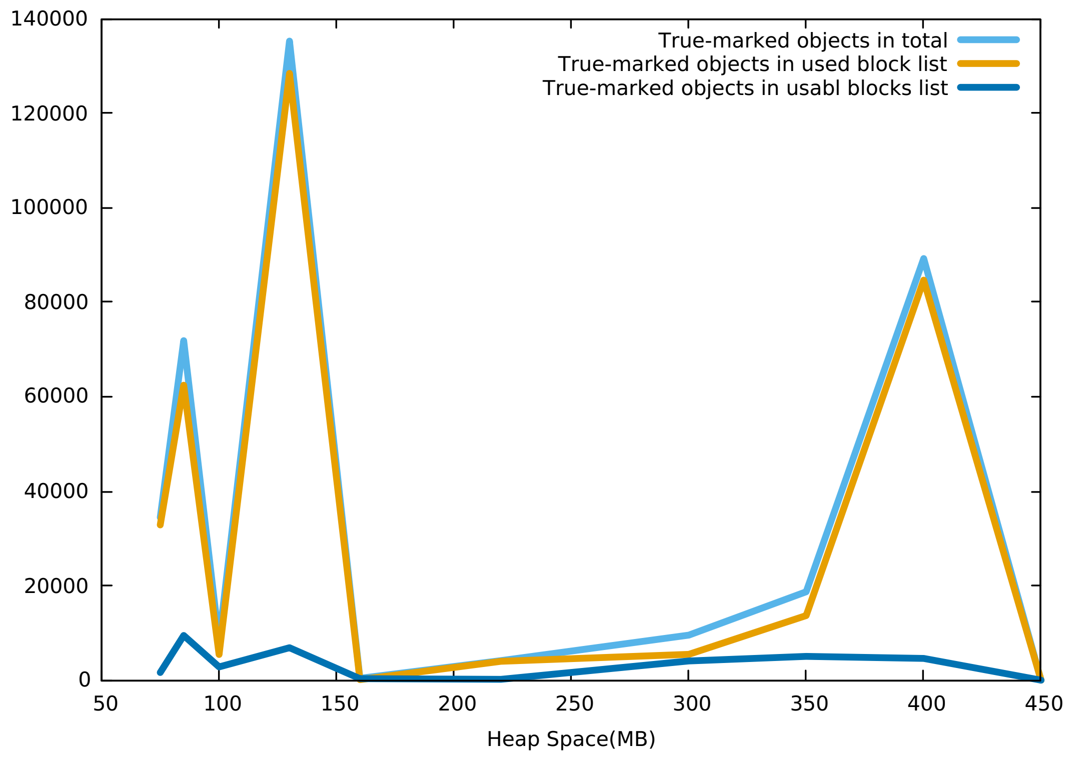

Figure 4.3: Heap Space vs True marked objects in used, usable blocks list and total

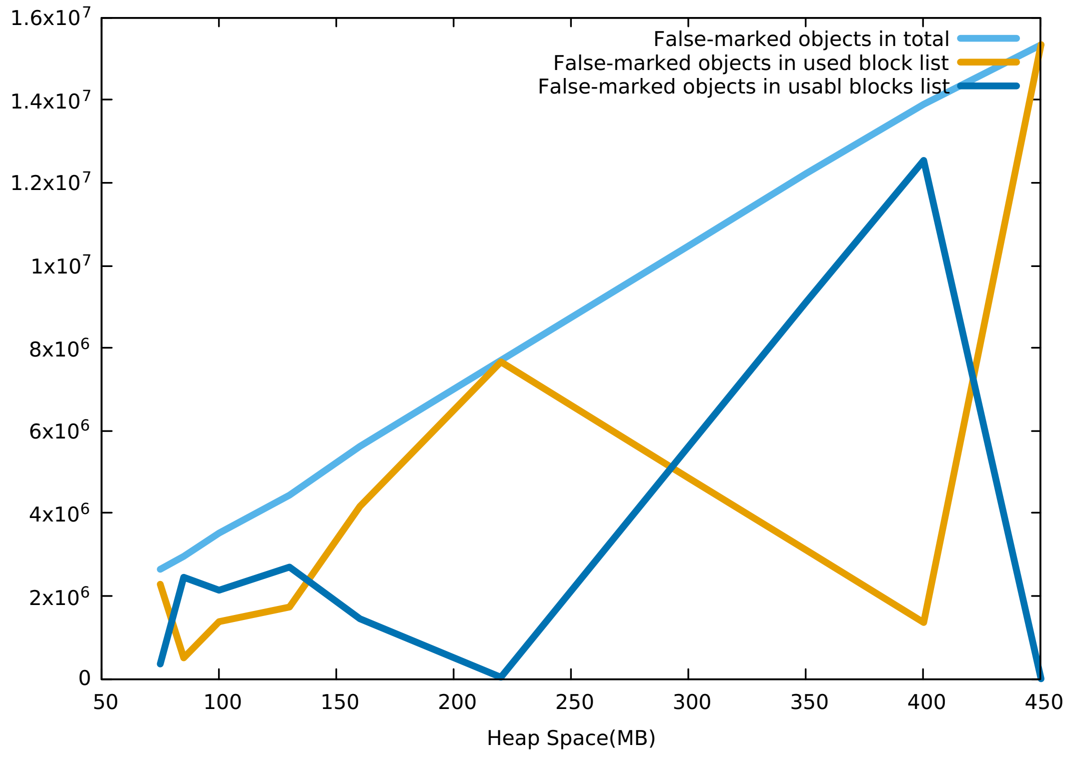

Figure 4.4: Heap Space vs False marked objects in used, usable blocks list and total

4.3 Fragmentation Problem

Fragmentation is a problem of inefficient memory space usage, reducing capacity or performance and often both. It depends on the storage allocation system and the type of fragmentation. In our Garbage Collection approach, it is more of a memory freeing fragmentation problem rather than memory allocation fragmentation. There are three different forms of fragmentation:

- External Fragmentation
- Internal Fragmentation
- Data Fragmentation

Ours suffers from internal fragmentation problem while attempting to free memory.
When memory is allocated to a process slightly larger than the requested size, the end of the memory space is wasted. This is called internal fragmentation. But as mentioned above, our problem is similar with this in case of memory freeing.
In Immix, we mark the the block as used where there is at least one line marked as true and a line is marked true if there is at least one live object. In the scenario, where there is just one live object in a line, it is marked true. But there can be free space in other lines. If we could just transfer that object to the free space of other mostly crowded lines, the line could be free and marked false for collection as free memory. Thus is gives rise to a fragmentation issue.
In the implementation that we based our work on [1] had this problem. It was an area where there was scope for major optimization. For this we had to find the corresponding line addresses of different types of blocks. Then we could finally retrieve the count of live objects in a line with its address and thus we could generate a statistic of how much space was utilized efficiently. Each line could have maximum of 12 objects[still not sure]. In our analysis, we checked if a line had more than 9 true objects. If so, we called it a Good Allocation. Then we generated a statistic how much of the valid lines had Good Allocation. It varied with the heap space size. The graph is shown in Figure 4.5.

Table 4.3: Statistic of Good Line Allocation(Less Fragmentation) with respect to Heap Size

Heap Size	Good Allocation in Line
70	96.75%
80	98.58%
90	84.89%
100	89.10%
150	72.16%
160	69.32%
170	66.46%
180	66.12%
190	63.59%
200	60.49%
210	54.12%
220	52.57%
230	92.45%
240	96.74%
250	95.32%
300	98.90%

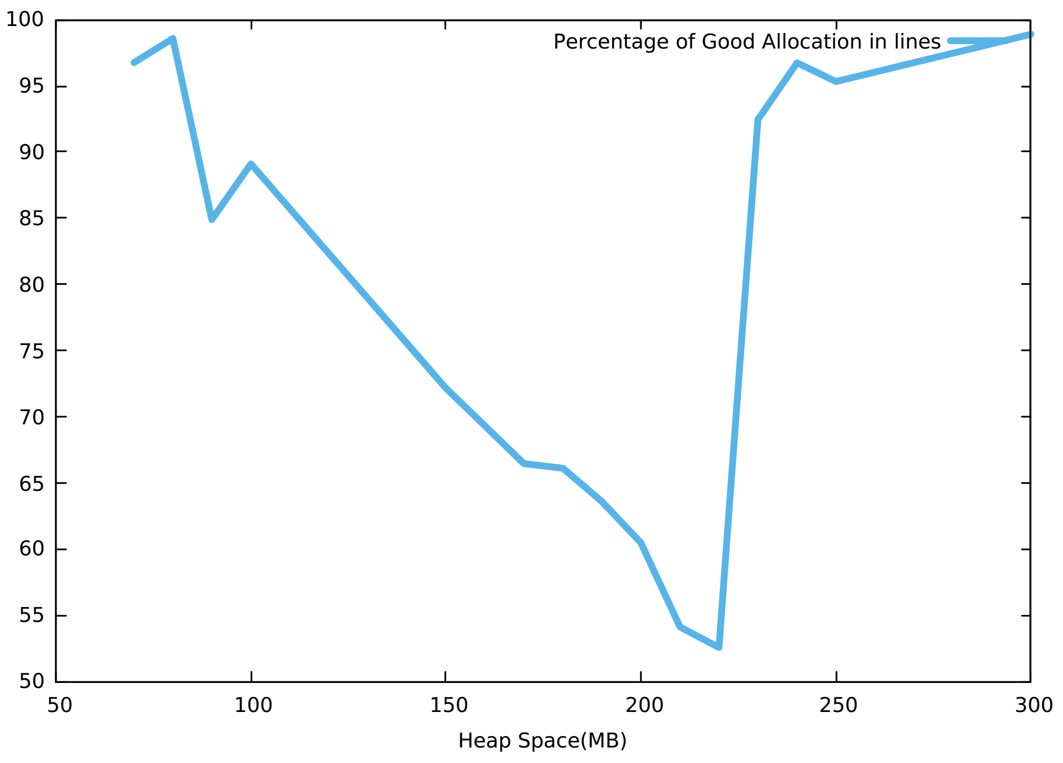

Figure 4.5: Heap Space vs Percentage

4.4 Insanity of objects

The objects that are found more than once in the list of usable blocks or used blocks, we are re-ferring them as insane objects. We tried to navigate why this happens. One of the things that we found was that sometimes objects are allocated on the same memory area twice consecutively. This might be a reason. We gathered the number of insane objects for varying number of heap space.

Table 4.4: Statistics of Insane true-marked objects in used and usable blocks list compared to various heap space size

Heap size	Insane objects as true-marked in used blocks list	Insane objects as true-marked in usable blocks list
75	3214	82
85	8874	237
100	398	156
130	4814	180
160	30	10
220	139	5
300	340	169
350	2296	159
400	3570	258
450	0	0

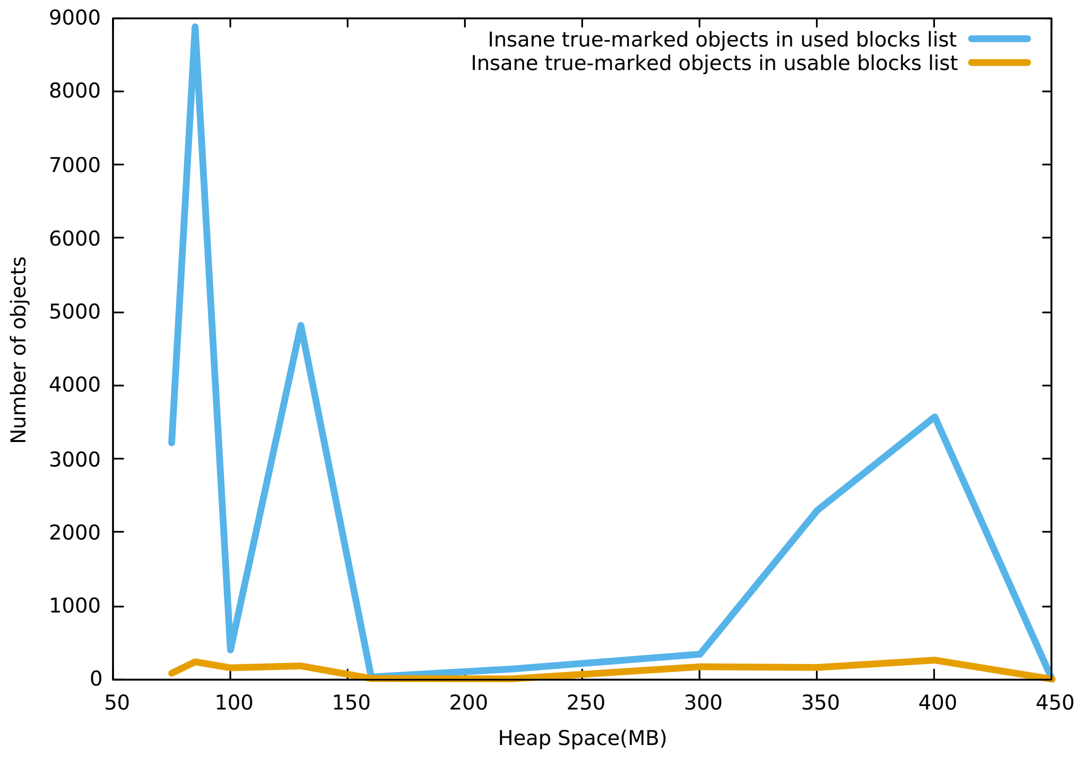

Figure 4.6: Heap Space Vs Insane Object count

Chapter 5

Future Works

Our current phase of research has some major findings and analysis on the scopes of improvement in implementing a GC using rust. This gives some significant opportunities to work with this topic in the future. They are described below.

5.1 Defragmentation

We can implement a defragmentation algorithm to improve the good allocation percentage where it is necessary. We found the percentage of good allocation lines were not satisfactory for some heap sizes. A well effective defragmentation algorithm which is known as Lightweight Opportunistic defragmentation or opportunistic defragmentation is sometimes used in immix to tackle the problem [8]. The algorithm tries to determine the source block and the destination block to defragment based on the statistical data obtained from the previous collection cycle. Then in the source block, live objects are opportunistically evacuated and moved to the destination objects and change the forwarding pointer that records the address of the new location. We have narrowed down the addresses of the badly allocated objects and other line addresses can also be found easily with our implementation. There are various heuristics from which we can select to choose the candidate source block and destination block. The most popular heuristic uses a histogram of live objects per blocks.

5.2 Removing Insanity

Insanity problem needs to be checked to gather better results and measure more accurate performance based on the result. Insane objects can lead to faulty data during some measurement of performance. Our findings showed that there were double allocations which should not have happened. To tackle this problem, the allocator functions need to be modified. Specially the recursive calls might need to be worked around in future implementation.

5.3 Turning it into a Mark-and-Sweep Algorithm

This also is an interesting idea to work on, which can use the specialty of Rust Language to incorporate with a tracing type collector. Immix normally collects free blocks and free lines in partially free blocks during its reclamation phase. It takes the new free block from the used blocks list and inserts it into usable blocks list. For the free lines in partially free blocks, lines in the partially free blocks are marked free using the line map. This reclamation stage can be changed from per block basis to per object basis which is simply a mark-and-sweep garbage collector. [19]

Chapter 6

Conclusion

Rust language is memory and type safe language which doesn't need a garbage collector for its own. Thus it is considered to be suitable for system programming. We analyzed Rust language and implementation as well as comparison of immix garbage collection in Rust and C language. Rust programming language is quite restrictive. It is hard to implement garbage collection without difficulty and without violating Rusts restrictive static safety guarantees. But, this restrictive language gives some benefit while implementing system programming and so, Rust language seems to be a good pick for standalone garbage collector. Although there were some difficulties that we faced while fixing the environment for rust. Many Ubuntu versions were used in this regard and finally Ubuntu 16.04.1 matched the requirements this language needed. Otherwise this language has many benefits to be worked with.

References

[1] Y. Lin, S. M. Blackburn, A. L. Hosking, and M. Norrish, “Rust as a language for high performance gc implementation,” in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pp. 89–98, ACM, 2016.

[2] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part i,” *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[3] M. Davis, “Mccarthy john. recursive functions of symbolic expressions and their computation by machine, part i. communications of the association for computing machinery, vol. 3 (1960), pp. 184–195.,” *The Journal of Symbolic Logic*, vol. 33, no. 01, pp. 117–117, 1968.

[4] P. Walton, “A gentle introduction to traits in rust.” Last accessed on August 8th, 2012, at 00:00:00PM. [Online]. Available: http://pcwalton.github.io/blog/2012/08/08/a-gentle-introduction-to-traits-in-rust/.

[5] bacon rajan, “A reference counted type with cycle collection for rust.” Last accessed on August 4 2015, at 02:08:00PM. [Online]. Available: https://github.com/fitzgen/bacon-rajan-cc.

[6] Manish, “A simple tracaing (mark and sweep) garbage collector in rust.” Last accessed on February 5, 2017, at 0:00:00PM. [Online]. Available: https://github.com/Manishearth/rust-gc.

[7] D. F. Bacon and V. Rajan, “Concurrent cycle collection in reference counted systems,” in *European Conference on Object-Oriented Programming*, pp. 207–235, Springer, 2001.

[8] S. M. Blackburn and K. S. McKinley, “Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance,” in *ACM SIGPLAN Notices*, vol. 43, pp. 22–32, ACM, 2008.

[9] S. M. Blackburn, P. Cheng, and K. S. McKinley, “Oil and water? high performance garbage collection in java with mmtk,” in *Proceedings of the 26th International Conference on Software Engineering*, pp. 137–146, IEEE Computer Society, 2004.

[10] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev, "Demystifying magic: high-level low-level programming," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 81–90, ACM, 2009.

[11] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen, "Implementing jalapeño in java," *ACM SIGPLAN Notices*, vol. 34, no. 10, pp. 314–324, 1999.

[12] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko, "A parallel, incremental and concurrent gc for servers," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 129–140, 2002.

[13] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, *et al.*, "The jikes research virtual machine project: building an open-source research community," *IBM Systems Journal*, vol. 44, no. 2, pp. 399–417, 2005.

[14] A. Rigo and S. Pedroni, "Pypy's approach to virtual machine construction," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 944–953, ACM, 2006.

[15] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An approachable virtual machine for, and in, java," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 30, 2013.

[16] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pp. 21–28, ACM, 2005.

[17] H.-J. Boehm, "How to miscompile programs with" benign" data races.," in *HotPar*, 2011.

[18] "Rust rfc 1543. add more integer atomic types," 2016. Last accessed on May, 2016, at 00:00:00PM. [Online]. Available: https://github.com/rust-lang/rfcs/pull/1543.

[19] T. Endo, K. Taura, and A. Yonezawa, "A scalable mark-sweep garbage collector on large-scale shared-memory machines," in *Supercomputing, ACM/IEEE 1997 Conference*, pp. 48–48, IEEE, 1997.

Appendix A

Codes

A.1 Mutators Alloc Function

This is the allocator function of mutator. We insert the newly allocated objects in our hashmap as key and as value we insert false.

```
pub fn alloc(&mut self, size: usize, align: usize) -> Address {
 let start = self.cursor.align_up(align);
 let end = start.plus(size);
 let mut hash = &myHashMap;
 if end > self.limit {
  let start2 = self.try_alloc_from_local(size, align);
  if hash.write().unwrap().contains_key(&start) == false{
   hash.write().unwrap().insert(start2,false);
   ALLOC_COUNT.store(ALLOC_COUNT.load(atomic::Ordering::SeqCst)+1
                                        ,atomic::Ordering::SeqCst);
  }
  start2
 }
 else {
  self.cursor = end;
  hash.write().unwrap().insert(start,false);
  ALLOC_COUNT.store(ALLOC_COUNT.load(atomic::Ordering::SeqCst)+1
                                        ,atomic::Ordering::SeqCst);
  start
 }
}
```

A.2 Hash Map

This portion is the declaration of our hash-map. We declared it in the in the file where the space, blocks and lines are delared.

```
lazy_static!{
 pub static ref myHashMap : RwLock<HashMap<Address,bool>> = {
  let mut ret :HashMap<Address,bool> = HashMap::new();
  RwLock::new(ret)
 };
}
```

A.3 Tracing

This is portion where we mark our objects in the hashmap from false to true if GC traces the object and it is contained in the hashmap.

```
let mut hash = &myHashMap;
for elements in roots.iter() {
 let addr = elements.to_address();
 let mut base = addr;
 if hash.write().unwrap().contains_key(&base) == true {
  let v = hash.write().unwrap()[&base];
  if v {
  }
  else {
   hash.write().unwrap().remove(&base);
   hash.write().unwrap().insert(base,true);
  }
 }
}
```

A.4 Main Code Portion

We use this code to find out the counts of true-marekd and false-marked (according to our self defined hashmap) objects in total, in usable blocks list, in used blocks list. We also use this code to find out the counts of insane counts which are showed in the analysis section.

```
let mut myhash = myHashMap.write().unwrap();
let mut myhashLine = myHashMapForLine.write().unwrap();
let mut usedBlocks = immix_space.used_blocks.lock().unwrap();
let mut usableBlocks = immix_space.usable_blocks.lock().unwrap();
let mut freeList = lo_space.write().unwrap();
let mut count = 0;
let mut count2 = 0;
let mut count3 = 0;
let mut count4 = 0;
let mut count5 = 0;
let mut count6 = 0;
let mut count7 = 0;
let mut count8 = 0;
let mut count9 = 0;
let mut sanity = 0;
let mut insane1 = 0;
let mut insane2 = 0;
let mut insane3 = 0;
let mut insane4 = 0;

for (key, val) in myhash.iter() {
 count = count + 1;
 if *val {
  count2 += 1;
  sanity = 0;
  for element in usedBlocks.iter() {
   let mut block = element;
   let line_table_index1 = key.diff(block.start())
                                        >> LOG_BYTES_IN_LINE;
   let end = block.start().plus(BYTES_IN_BLOCK);
   if *key >= block.start() && *key <= end {
    let len = BYTES_IN_LINE;
    for i in 0..LINES_IN_BLOCK {
     let mut address=block.start().plus(i*BYTES_IN_LINE);
```

```
       if *key >= address && *key <= address.plus(BYTES_IN_LINE){
        if sanity==0 {
         count8 = count8+1;
         sanity = 1;
        }
        else {
         insane1 += 1;
        }
            if myhashLine.contains_key(&address) == true {
             let mut v = myhashLine[&address]+1;
             myhashLine.remove(&address);
             myhashLine.insert(address,v);
            }
            else{
             myhashLine.insert(address,1);
            }
       }
      }
         break;
    }
   }
   sanity = 0;
   for element in usableBlocks.iter() {
    let mut block = element;
    let end =  block.start().plus(BYTES_IN_BLOCK);
    let len = BYTES_IN_LINE;
    if *key >= block.start() && *key <= end {
     let len = BYTES_IN_LINE;
     for i in 0..LINES_IN_BLOCK {
      let mut address=block.start().plus(i*BYTES_IN_LINE);
      if *key >= address && *key <= address.plus(BYTES_IN_LINE){
       if sanity==0 {
        count6 = count6+1;
        sanity = 1;
       }
           else {
        insane2 += 1;
       }
           if myhashLine.contains_key(&address) == true {
```

```
        let mut v = myhashLine[&address]+1;
        myhashLine.remove(&address);
            myhashLine.insert(address,v);
       }
       else{
        myhashLine.insert(address,1);
       }
      }
     }
         break;
    }
   }
  }
  else{
   count3 = count3 + 1;
   for element in usedBlocks.iter() {
    let mut block = element;
    let end =  block.start().plus(BYTES_IN_BLOCK);
    let len = BYTES_IN_LINE;
    sanity = 0;
    if *key >= block.start() && *key <= end {
     if sanity==0 {
      count5 = count5+1;
      sanity = 1;
     }
     else {
      insane3 += 1;
     }
     break;
    }
   }
   sanity = 0;
   for element in usableBlocks.iter() {
    let mut block = element;
    let len = BYTES_IN_LINE;
    let end =  block.start().plus(BYTES_IN_BLOCK);
     if *key >= block.start() && *key <= end {
          if sanity==0 {
           count7 = count7+1;
           sanity = 1;
          }
          else {
           insane4 += 1;
          }
     }
     break;
   }
  }
  let line_table_index = key.diff(immix_space.start())
                                         >>LOG_BYTES_IN_LINE;
  let markValue = immix_space.line_mark_table()
                                         .get(line_table_index);
  if markValue != LineMark::Free {
   count4 += 1;
  }
  if markValue == LineMark::Free {
   count9 += 1;
  }
 }
 println!("------------------------------------------");
 println!("alloc␣in␣gc␣called␣{}␣", unsafe { OBJ_COUNT } );
 println!("hash␣size␣{}␣", count);
 println!("true␣found␣size␣{}␣", count2);
 println!("true␣found␣in␣used␣blocks␣{}␣", count8 );
 println!("true␣found␣in␣usable␣blocks␣{}␣", count6 );
 println!("false␣found␣size␣{}␣", count3);
 println!("false␣found␣in␣used␣blocks␣{}␣", count5 );
 println!("false␣found␣in␣usable␣blocks␣{}␣", count7 );
 println!("false␣not␣in␣free␣lines␣{}␣",count4);
 println!("false␣in␣free␣lines␣{}␣",count9);
 println!("␣size␣of␣used␣blocks␣{}", usedBlocks.len());
 println!("␣size␣of␣usable␣blocks␣{}", usableBlocks.len());
 println!("insane␣in␣usedblocks␣true␣{}␣",insane1);
 println!("insane␣in␣usableblocks␣true␣␣{}␣",insane2);
 println!("insane␣in␣usedblocks␣false␣{}␣",insane3);
 println!("insane␣in␣usableblocks␣false␣␣{}␣",insane4);
 let mut good = 0;
 let mut bad = 0;
```

```
for (key,val) in myhashLine.iter(){
 if (*val >= 11){
  good += 1;
 }
 else {
  bad += 1;
 }
}
let total = good+bad;
let percentage = 100.0 * good as f64 / total as f64;
println!("true␣lines␣{}␣false␣lines␣{}␣total␣{}␣percentage␣{}",
                                        good,bad,total,percentage);
```

Generated using Undegraduate Thesis LaTeX Template, Version 1.2. Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

This thesis was generated on Thursday 16th February, 2017 at 2:25pm.