NORTHWESTERN UNIVERSITY

Investigation of Generational Incremental Garbage Collector

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Computer Science

By

Yixi Zhang

EVANSTON, ILLINOIS

August 2013

ABSTRACT

Investigation of Generational Incremental Garbage Collector

Yixi Zhang

Interactive applications written in high level programming languages requires acceptable short pauses to achieve quality user experience. Though generational garbage collectors can shorten the pause time by minimizing the objects need tracing, and also decrease the frequency of disruptive pauses, for some programs with intense interactivity requirement, the pauses for older generation garbage collections still destroy the use.

This thesis first explores the basic algorithm of generational and mark-sweep garbage collectors. Then we give an optimal generational incremental garbage collector design, that can provide a combination of short pause time and improved long-term performance. Finally we demonstrate the power of Racket plai/gc2 framework as an experiment platform for garbage collector design and evaluation.

Keywords: Garbage Collector, Generational GC, Incremental GC, Racket

Acknowledgements

First and foremost, I want to thank my girlfriend Chongchong, who has kept me company through many late nights, for her undying patience and support throughout the whole process. I must thank my parents - without their support I cannot be here.

I would like to express my special appreciation and thanks to my advisor, Robby Findler, you have been a tremendous mentor for me. I would like to thank you for encouraging my research and for allowing me to grow as a programmer. Your advice on both research as well as on my career have been invaluable. I'd also like to thank Jay McCarthy, whose knowledge and insight of the inner workings of plai/gc2 framework was very helpful.

And finally, I would like to thank my committee members, professor Peter Dinda, professor Nikos Hardavellas for serving as my committee members even at hardship. I want to thank you for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions, thanks to you.

Table of Contents

List of Figures

CHAPTER 1

Introduction

Users of web browsers must have experienced pauses and freezes after clicking around inside the page. This is because the garbage collector in the browser runtime works and temporarily stops the browser itself from responding to your following operations. Interactivity is very important to all applications that are communicating with users, which are expected to be responsive by either excluding garbage collectors, and moving to manual memory management; or by staying with garbage collectors, but utilizing a more incremental tactic. The former strategy is promoted by Apple on their iOS devices, as the unacceptable long pauses of garbage collections are believe as one of the roots of poor performance on mobile web applications[**6**]. However, on the desktop side, since JavaScript are dominating the web, both Google's Chrome and Mozilla's Firefox have introduced their latest virtual runtimes equipped with a new incremental garbage collector.

On the other hand, high level programming languages' virtual machines are widely using a generational architecture, including Java (JVM), Racket and Erlang[**1**], etc. This thesis is interested in producing a garbage collector design with the combination of generational architect and the incremental tactic, to provide both short pauses and overall effective performance.

Unlike the traditional way to experiment garbage collector designs, which bases on the implementation of interpreter or compiler from scratch, this thesis uses the plai/gc2 framework from Racket[**4**] for our research platform. The first contribution of this thesis

is the implementations of several garbage collectors via plai/gc2. The second contribution is an optimal design of generational incremental garbage collector. The third contribution of this thesis is the extension to plai/gc2 framework, and a series of mutator benchmark programs that evaluates the performance of our design.

1.1. Thesis Overview

The remainder of this thesis is organized as follows:

- Chapter 2 briefly reviews the algorithm of various basic garbage collecting strategies, with an emphasis on the combination of generational and mark-sweep GC.
- Chapter 3 presents the design details of generational mark-sweep garbage collector, with an emphasis on the design of optimal generational incremental garbage collector.
- Chapter 4 offers rudimentary performance measurements and result analysis for a series of benchmark programs written in plai/gc2/mutator language, including the extension we made to the framework to write more complicated programs.

1.2. Code Availability

Complete code for the implementations of garbage collectors, benchmark programs, and the extension to plai/gc2 are available via the Github repository:

```
git@github.com:yixizhang/racket-gc.git
```

CHAPTER 2

Basic Garbage Collection Algorithms

In this section we introduce the algorithms behind various garbage collectors, including mark-sweep GC (both batch and incremental version) and generational GC.

2.1. Batch Mark-Sweep GC

Mark-Sweep garbage collector, as its name, is consist of two phases: detect the live objects from the garbage, and reclaim the garbage[**8**]. All the objects that are reachable by local variables in stacks or registers (the collection of them are called "root set") are considered "useful" or "live" for the on-going computation. Therefore the rest of them are garbage that can be collected after tracing finishes.

Mark-Sweep GC mark objects at different stage during the tracing process with different color. The brief process is as follows. All objects are marked white when tracing starts. Then all objects directly referenced by the root set are marked grey. If there is a grey object, its "children" objects that are referenced by it are marked grey, and itself upgrades to black, means collector finishes the tracing on it and will never trace it again. This process continues until all objects are either black or white. Finally white objects are swept out, memories are reclaimed for further allocations.

This scheme is trivial to implement but comes with drawbacks. Firstly, it introduces fragmentation. In the worst scenario it will require compaction, however, based on our observation and prior works[**10**], it is rare to happen. Secondly, the whole tracing process

must finish in one pass, means the whole heap needs to be traversed at least once (depends on the objects graph). No memory can be reclaimed before all live objects are traced. This is usually called "Stop-the-World", as the running programs are paused to wait for garbage collector does its work.

2.2. Incremental Mark-Sweep GC

Mark-Sweep garbage collectors can be made incremental, which means it only does some units of collection, or at least tracing, and return the control back to mutator, in order to shorten the collection pauses. For implementation, a tracing stack recording all the grey objects (whose children are traced yet) is introduced to save the progress of tracing. So next time when the incremental collection restarts it knows where to continue the work.

Besides that, we should understand the running programs or the mutators in the garbage collector literature can modify the objects graph before incremental GC finishes tracing. This makes incremental GC inevitably conservative. Just imaging some object is "dropped" by the mutator program after it was traced. This kind of objects and their children remain as live till the end of this collection round, which is okay because they will be reclaimed by the next collection round.

Also, destructive operations by mutators, e.g. set-first! and set-rest! in Racket, could set references inside a black object to a white value. Because black objects are never traced twice, its white children are mistakenly collected. This breaks the graph of reachable data structures, as part of the graph is lost. We bring in an invariant, that a black object can never point to a white one[**5**], in order to survive from this kind of situations. The typical

method to maintain the invariant is to have a writer barrier bound to the destructive operations, so whenever a reference inside a black object is overwritten to a white value, the writer barrier upgrades the white value to color grey[**5**].

2.3. Generational GC

Generational GC's are widely used by high level programming language virtual machines, including JVM, Racket virtual machine, and Erlang virtual machine, etc. This model is based on the observation, that most objects live a very short time, while a small percentage of them live much longer[**7**]. So the heap is divided into several subsections, and each section contains values in different "age". When each section is full, the garbage collector only traces reachable objects inside the local subheap, so the pausing time drops substantially due to the decreasing of tracing load.

When the youngest generation is filled up, live objects are copied over to the older generation. During the copying, forwarded locations are saved locally so that when forwarded objects are referenced again, collector can update their reference locations according to the stored information. The memory usage inside older generation increases as live objects from younger generation are copied over, so it is possible that during the copying GC, older generation becomes full and needs reclamation. Only after the collection in older generation finishes and some memory are reclaimed, the copying GC inside younger generation can continue.

One another advantage of generational GC, is there are no fragmentation inside the young generation, that all new objects are allocated sequentially, with no extra effort to find free spaces.

Intergenerational References

Because objects in older generation are not traced, it is important to find references that make objects in younger generation reachable by older values during copying collections. Our implementation keeps an indirection table[**7**], to record the two ends of intergenerational pointers.

During each collection for young generation, the objects from the indirection table are considered reachable, and are copied to older generation. The associated pointers are also updated accordingly by the forwarded locations.

The table is cleaned up after each collection for young generation. If it is filled up before young generation is full, which is possible, considering the case that many objects in older generation point to the same object on the other side, the collector will copy over referenced objects, and clear the table.

The collection for older generation also requires a set of references pointing in the opposite direction. It is taxing to keep another indirection table, as collection for young generation happens more frequently than the older one, which forces the table to be updated or swept more frequently. As the indirection table is implemented with an array of pairs of locations, updates require scanning the table and possibly introduce fragmentation between valid pointers. Our solution is to scan the young generation for the big collection[**3**], to find all intergenerational references from younger side. The overhead of scanning younger generation is mainly determined by the size of smaller heap.

It is intuitively a good choice for generational garbage collectors with a batch GC on the bigger heap, because the scanning is only needed when older generation is full. However, for the incremental GC on older generation, the scanning is more recurring

than we want. So we have a different treatment for the incremental GC, and we will cover that topic in later chapter.

CHAPTER 3

Design of Generational Incremental GC

In this section we explore several design decisions we made for the generational incremental garbage collector.

3.1. Timing for Incremental GC

Incremental GC collects garbage before the heap is full. So to guarantee sufficient progress, we use an allocation clock for each unit of allocation, a corresponding unit of collection is done[**2**]. The safe tracing rate is (M - L)/2L, where M and L are respectively the memory size and the maximum amount of live data[**9**]. The collector starts with a tracing rate 1 with the assumption that live objects take about 1/3 of the heap and modify the rate after every collection given by the data it collects. There's another parameter controls how often to start a tracing round, in order to batch up the work over several allocations for efficiency reasons.

Being afraid of the situation that the older generation will be filled up before younger generation copying GC finishes, it is intuitive to set the parameter controlling the frequency of incremental GC very small. However, the interleaving of GC's on both generations forces the scanning of younger generation, at the beginning of every smaller subheap collection, because we need to treat objects in older generation referenced by live objects in the younger generation as roots as well, in order to maintain the reachability of objects graph.

Incremental GC is introduced in the first place as the garbage collector wants to give control back to mutator after an acceptable short pause, but in this scenario, all incremental works add up as overhead to the small subheap collections in the eyes of mutator programs. If we can delay the incremental GC until the end of younger generation copying GC, the scanning of smaller subheap becomes unnecessary, which saves a substantial portion of scanning overhead. To ensure the sufficient progress of incremental GC on the bigger heap, we can increase the allocation clock parameter, but it is beyond the topic of this thesis. The pseudo code for the collection process is shown in the list below.

Our observation shows the garbage collectors basing on this design have remarkable improvement on both pause time and overall performance compared to other implementations.

3.2. Batch-Up

Since incremental GC happens after the copying GC inside younger generation, so if the size of younger generation is relatively small, frequency of incremental GC unnecessarily rises. Not many objects become garbage since last mutator running round, tracing again all the live values becomes the new primary overhead to the incremental GC. One way to cope this, is to "batch-up" the incremental GC after every two rounds of younger generation copying GC. Our implementations employ the batch-up strategy when the size ratio of two generation is e.g. 1 vs. 10.

3.3. Free List

Because Mark-Sweep GC on the older generation introduces fragmentation, finding free spaces for copying allocations could be an expensive operation, if we do it by scanning the heap. A better way would be to maintain a free list, a linked list that connects all the sequential free spaces. When allocating objects from the smaller subheap, collector searches from the head of free list and returns the first slot that matches the size of allocation. Our observation shows the "FCFS" strategy serves us well, as the fragmentation rates are typically low, and values usually don't have large size differences (most allocations are for flat or pair values).

The coalescing is taken care by the sweeping process, which finds all free space in sequence, and tag them with the size and the next free node location. Another improvement compared to the traditional scanning way is, free spaces are no longer necessary to be "memset" to "free" or 0, which enables us to implement the sweeping process in one scanning pass, even for the incremental GC which requires a relatively low pause time.

3.4. Two-Space in Younger Generation

As objects are moved during smaller subheap GC, we need to save their forwarded locations somewhere, in case some other objects reference forwarded values and need to trace their updated locations. Virtual machines usually save them in an additional list or table, but in our implementations, forwarding locations are stored where values originally locate. If younger generation is organized as one space, these forwarding informations have to be thrown away before any new value can be allocated. In some rare scenarios, that objects happen to be allocated partially at the end of younger subheap, and after

garbage collection mutator has no idea where they are forwarded to. So in this case we would want those forwarding informations live longer.

A two-space design serves for this purpose. The younger generation is divided into two equal spaces. When one space fills up, live objects are copied over to older generation and forwarding fields stay in their original places. Then the other space becomes the new "To Space" for allocations. Forwarding informations will be eventually reclaimed when "To-Space" switches, so in some rarer case, e.g. mutator conses two relatively large objects, two arguments of cons are allocated and probably forwarded before the 'cons itself operates, dividing spaces doesn't help. We believe this happens rarely, even if possible for real-world virtual machines comes with garbage collector, when all arguments of a function can't fit in a single generation of heap memory. In our implementation, the fix to these scenarios is to increase the size of memory.

CHAPTER 4

Implementation and Evaluation

In this section we introduce the implementations of generational incremental garbage collector via Racket's plai/gc2 framework, and the evaluations of GC performances through mutator programs written in Racket.

4.1. Implementation

Racket's plai/gc2 framework[**4**] provides a convenient way to experiment garbage collector design without touching virtual machines. So that programmers don't need to understand low-level concepts, including stacks walking, pointer arithmetic, raw memory manipulation, etc., or more advanced topics, e.g. computation state recording and restoration, intermediate representation of mutator programs, etc. The framework abstracts local variables in running stacks and register values into a convenient root set. All it requires to be implemented are functions related to objects allocation and type detection.

In addition, plai/gc2 provides heap visualization for debugging and evaluation. Figure 4.1 shows the heap when mutator program is running with our generational incremental garbage collector implementation.

The framework supports only flat, pair and closure values out-of-box. In order to evaluate mutator programs require vector, hash table, and self-defined structs, we added vector, struct, and struct instance types to the framework.

To compare the performance between batch and incremental GC, as well as the impact of incremental GC start timing, we implemented three garbage collectors (with their name in benchmark figures inside parenthesis), which are all generational GC, but differ in the GC scheme on older generation.

- Batch GC (batch)
- Incremental Interleaving GC (incremental)
- Incremental Optimized GC (incremental2 or incremental optimized)

4.2. Evaluation

Unlike many profiling interfaces provided by mainstream virtual machine distributors, e.g. JVM, RVM, etc, which enable the observations on GC event level only[**11**], plai/gc2 gives us the chance to fined-grained evaluations on every heap operation. So our benchmarks focus on two aspects: 1) pause of every GC round; 2) largest memory usage. To benchmark the performance for different designs in term of locality, we simulate the memory hierarchy, with specs:

- Directed-mapping (E=1), Write-Back
- Size: L1=64, L2=256, Main memory=1024+
- Cycles: L1=1, L2=10, Main memory=100

Most of our benchmark programs are simply repeatedly allocating objects with different types onto the heap. Considering the program in Listing 4.1, which keeps allocating balanced binary trees. Once a tree is allocated, it becomes garbage because further computations don't need it anymore. However, if during its allocation, either the younger or

older generation is full, all the branch nodes are treated as roots, and all nodes on the tree are live objects during the collection.

```
#lang plai/gc2/mutator
(allocator-setup "../collector.rkt" 2048)
(define (build-one) empty)
(define (traverse-one x1) (empty? x1))
(define (build-tree n)
  (cond
    [(= n 0) (cons #f #f)]
    [else (cons (build-tree (- n 1))
                (build-tree (- n 1)))]))
(define (trigger-gc n)
  (if (zero? n) 0 (begin (build-tree 4) (trigger-gc (- n 1)))))
(define (loop i)
  (if (zero? i)
    'passed
    (let ((obj (build-one)))
      (trigger-gc 5)
      (if (traverse-one obj) (loop (- i 1)) 'failed))))
(loop 10)
```

Listing 4.1. balanced binary tree

By allocating different type of values, we are hoping to observe the performance of different collecting strategies with the impact of various allocation patterns. However, we are still hoping to get more convincing result by running more "useful" programs. For example, the character statistic program shown in Listing 4.2 records the occurrences of all characters from a short paragraph of "Sherlock Holmes". Specially, this program utilizes the vector and struct types we added to plai/gc2 framework to implement hash table.

```
#lang plai/gc2/mutator
(allocator-setup "../collector.rkt" 2048)
;; definition of hash table
...
```

```
(define stats (make-hash))
(define (count ip)
  (let ([c (read-char ip)])
    (cond
      [(eof-object? c ) (void)]
      [else (let ([n (hash-ref stats
                               c
                               (lambda ()
                                 (begin
                                   (hash-set! stats c 0)
                                   0)))])
              (hash-set! stats c (+ 1 n))
              (count ip))])))
(define txt "To Sherlock Holmes she is always the woman. I have seldom
   heard him mention her under any other name. In his eyes she eclipses
   and predominates the whole of her sex. It was not that he felt any
   emotion akin to love for Irene Adler. All emotions, and that one
   particularly, were abhorrent to his cold, precise but admirably
   balanced mind. He was, I take it, the most perfect reasoning and
   observing machine that the world has seen, but as a lover he would
   have placed himself in a false position. He never spoke of the softer
    passions, save with a gibe and a sneer.")
(count (open-input-string txt))
```

Listing 4.2. character statistics

More interestingly, we rewrote the HTML to XML parser from the Racket html library in plai/gc2/mutator, by extending the *import-primitives* and *gc-scheme* functions for the mutator interpreter. The HTML page in Listing 4.3 is parsed, and generates the result shown in Listing 4.4.

```
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <title>0</title>
</head>
<html>
</html>
```

Listing 4.3. html page

```
(#(struct:element
```

```
   #(struct:location 0 0 0)
   #(struct:location 0 0 6)
   head
   ()
   (#(struct:element
      #(struct:location 0 0 9)
      #(struct:location 0 0 76)
      meta
      (#(struct:attribute
         #(struct:location 0 0 41)
         #(struct:location 0 0 75)
         content
         text/html; charset=utf-8)
       #(struct:attribute
         #(struct:location 0 0 15)
         #(struct:location 0 0 40)
         http-equiv
         content-type))
      ())
    #(struct:element
      #(struct:location 0 0 80)
      #(struct:location 0 0 87)
      title
      ()
      (#(struct:pcdata #(struct:location 0 0 76) #(struct:location 0 0
         80)

  ) #(struct:pcdata #(struct:location 0 0 87) #(struct:location 0 0 88)
     0)))))
 #(struct:pcdata #(struct:location 0 0 107) #(struct:location 0 0 108)
)#(struct:element
   #(struct:location 0 0 108)
   #(struct:location 0 0 114)
   html
   ()
   ())
 #(struct:pcdata #(struct:location 0 0 122) #(struct:location 0 0 123)
))
```

Listing 4.4. parsed result

4.3. Results Analysis

Benchmark results are shown in Figure 4.2, 4.3, 4.4 and 4.5. All the numbers are normalized to the results of Batch GC. For example, the incremental optimized GC uses only 45% of the whole heap. Also, we can see that for the collector with a better incremental collection start timing, its efficiency and overall performance are better than the incremental collector having collections on two generations interleaved. By avoiding the scanning younger heap overhead, incremental optimized GC can achieve a short pause time stably around 30% of the batch GC, which is our primary goal of using an incremental collecting strategy. Further more, as most programmers believe that incremental GC must be less effective in overall performance, as it's comparatively more conservative and must afford the overhead of keeping track of progress, our benchmark result shows that incremental optimized GC can decrease the overall pause time down to about 50% of that of the batch GC.

We can gain more details from the following two figures. Figure 4.6 and 4.7 respectively compares the memory usage and overall running time of batch GC and incremental optimized GC, in the eyes of the character statistic mutator program. Especially Figure 4.7 demonstrates that, incremental GC with the right start timing can have not only short pauses, but also perform better in the long term. We attribute this fact to two points that differ within the design of batch and incremental optimized GC. Firstly, via the timing of collecting start, incremental optimized GC saves the need to scanning the younger heap; Secondly, as incremental GC starts after the younger heap copying GC, live objects from the younger generation are freshly black values, so they are not traced by the incremental

collection once again. In contrast, batch GC ignores the "fruit of labor" of copying GC, and all values are traced again from white.

By repeating the HTML page parsing 80 times, we stimulated the performance of garbage collectors running with computation intensive programs. Our benchmark result shows even with the heaviest working load, optimal incremental GC doesn't generate long-term overhead and still maintain a slightly better performance than the batch GC, where incremental GC's longest pause is 44% of the batch GC, and the average pause is 95% of batch GC. Figure 4.8 and 4.9 respectively shows the memory usage and running time performance between the two implementations.

```
#lang plai/gc2/mutator
(allocator-setup "../../collector.rkt" 10240)
(require "html.rkt")
(import-primitives
 open-input-file)

(define (loop i)
  (if (zero? i)
    'passed
    (begin
      (read-html-as-xml (open-input-file "1.html"))
      (loop (- i 1)))))
(loop 80)
```

Listing 4.5. html parser

Our benchmarks also evaluate the impact of subheap size. The result in Figure 4.10 proves that when the younger generation drops to a comparatively small size, its performances fall off substantially. Based on our experiments, batch-up strategy cannot bring the performance of generational incremental GC back to its optimal stage, but can still improve the efficiency by about 25%.

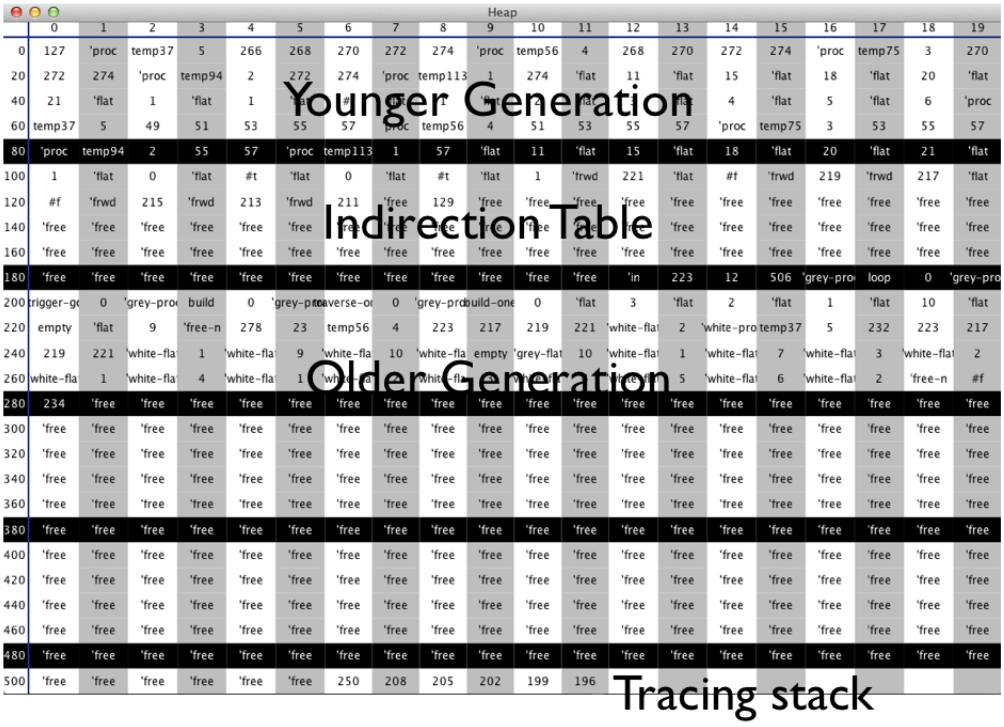

Figure 4.1. Heap Visualization

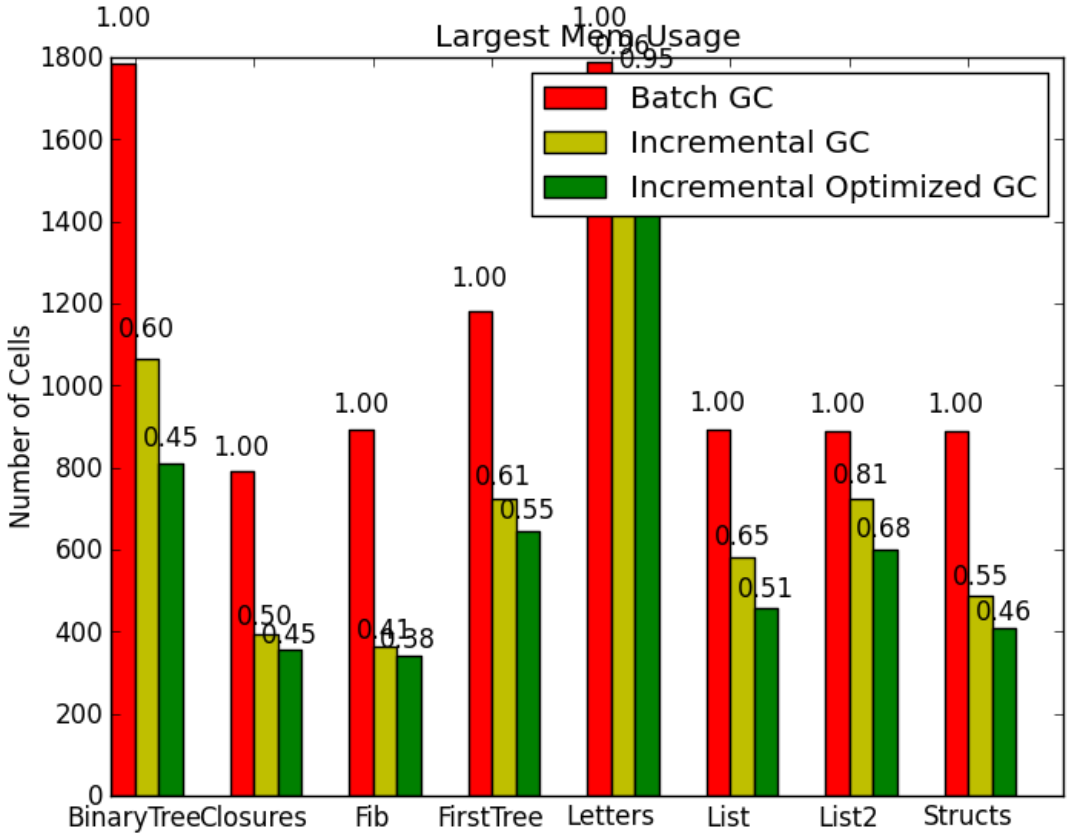

Figure 4.2. Largest Memory Usage

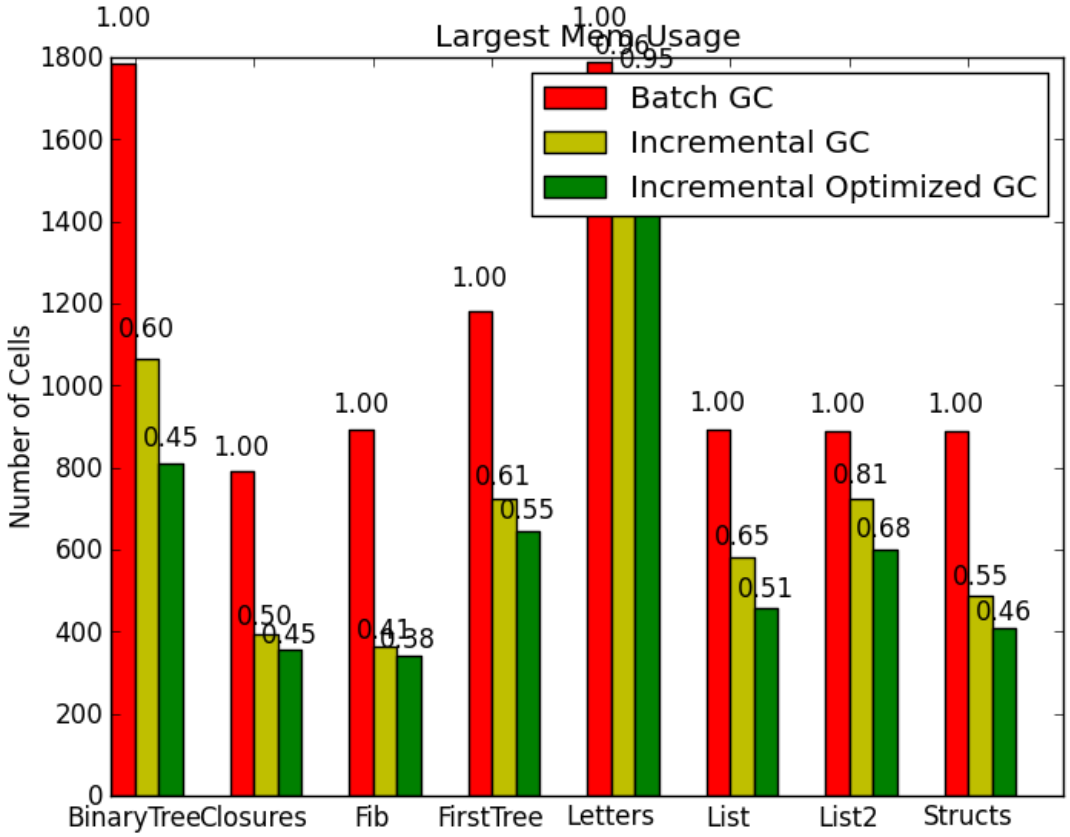

Figure 4.3. Longest Collection Pause

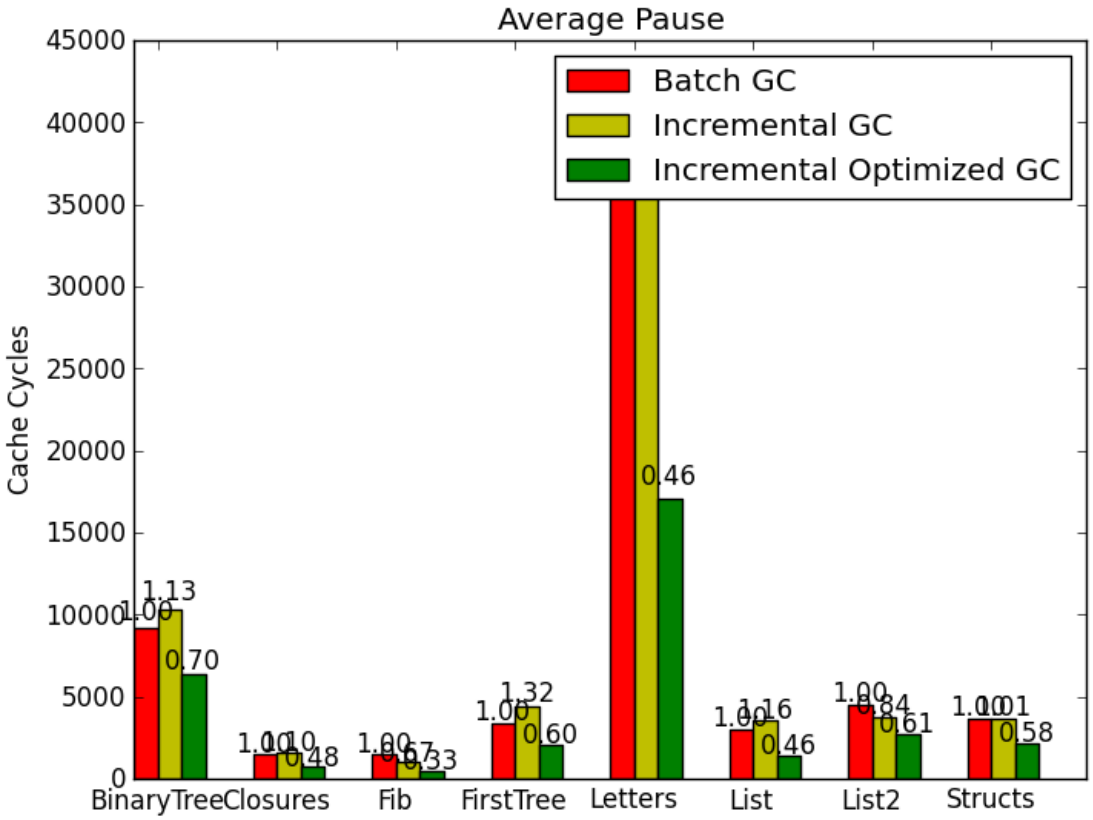

Figure 4.4. Average Collection Pause

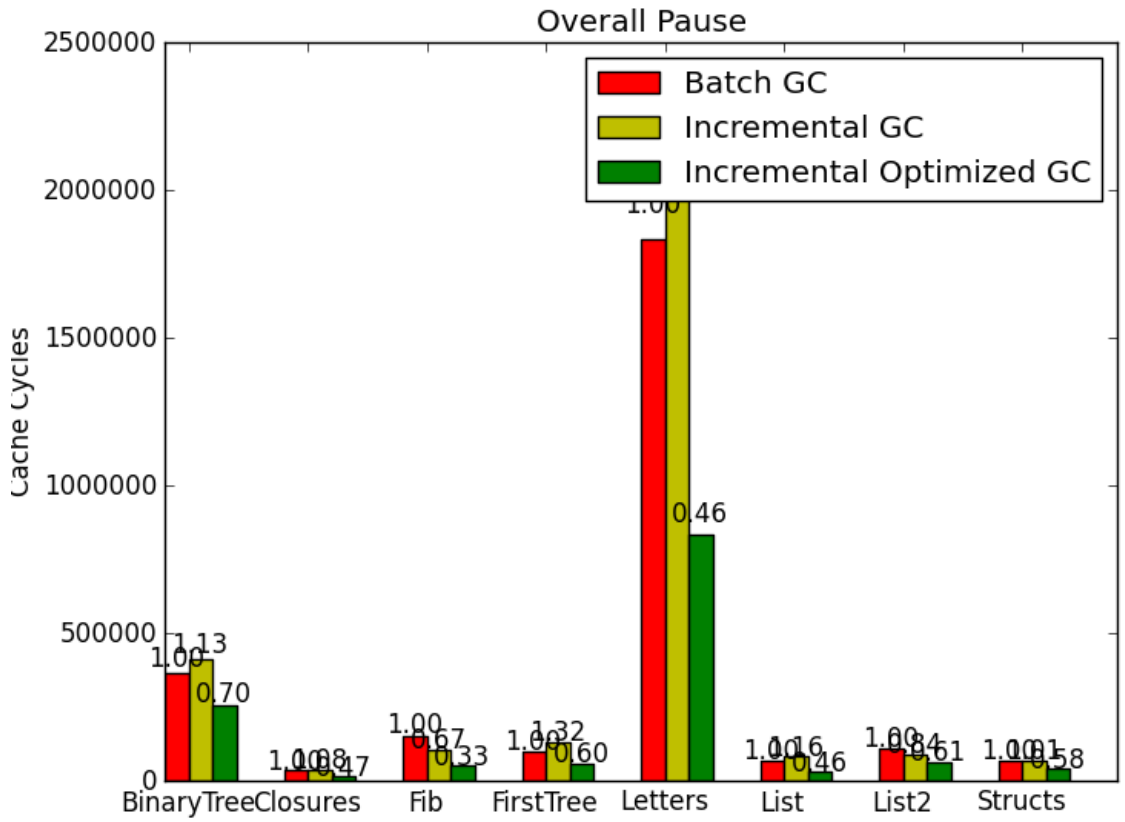

Figure 4.5. Overall Collection Pause

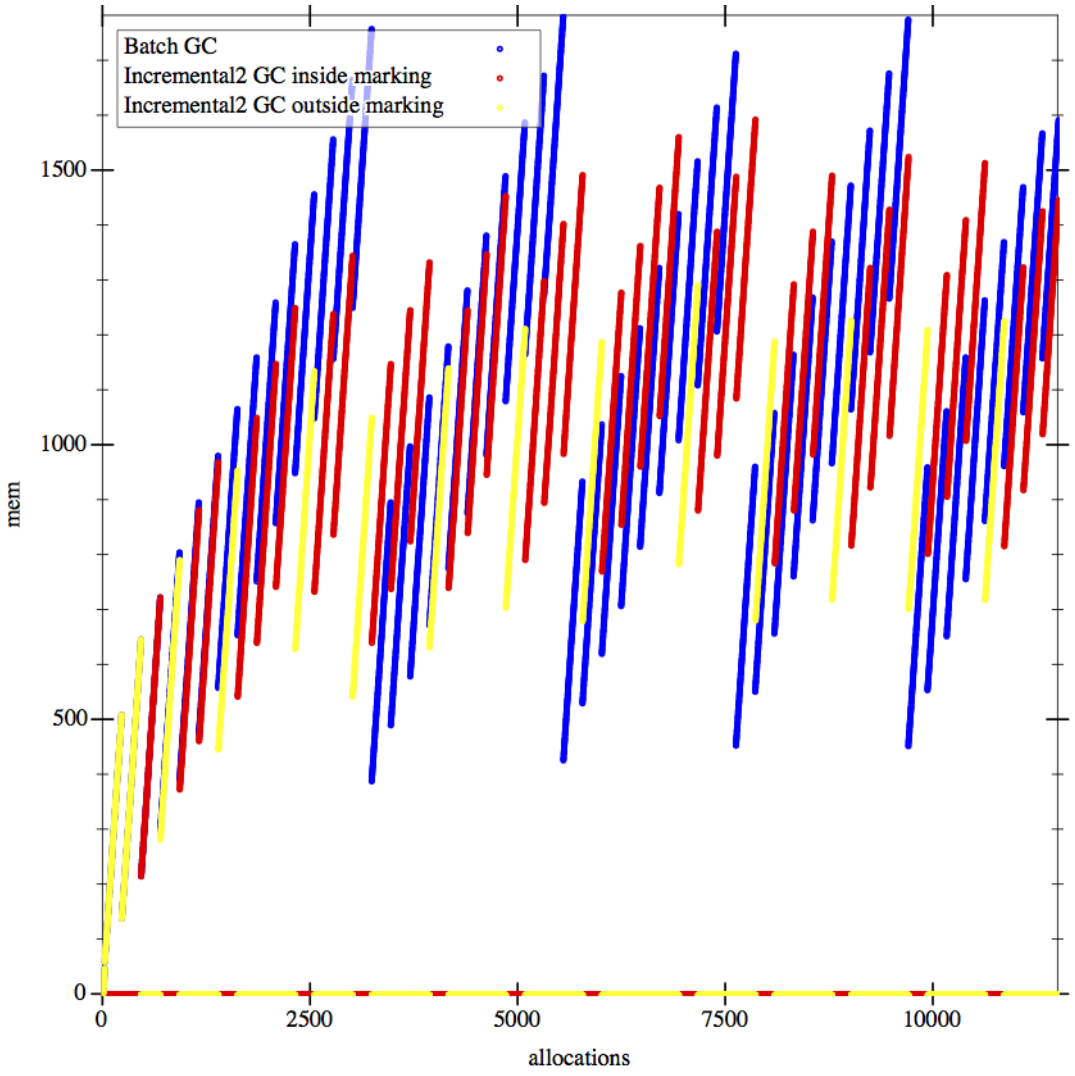

Figure 4.6. Memory Usage

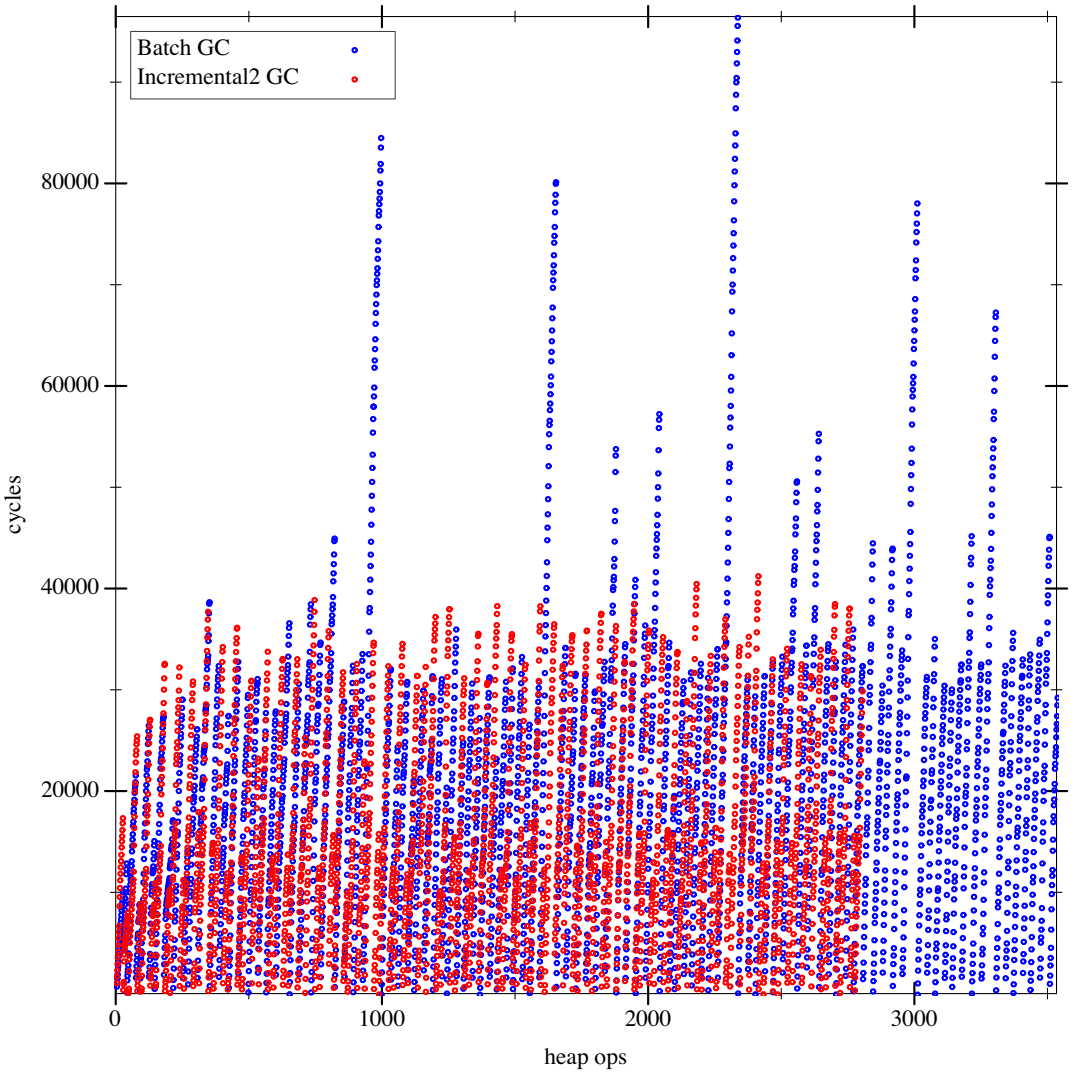

Figure 4.7. Running Time

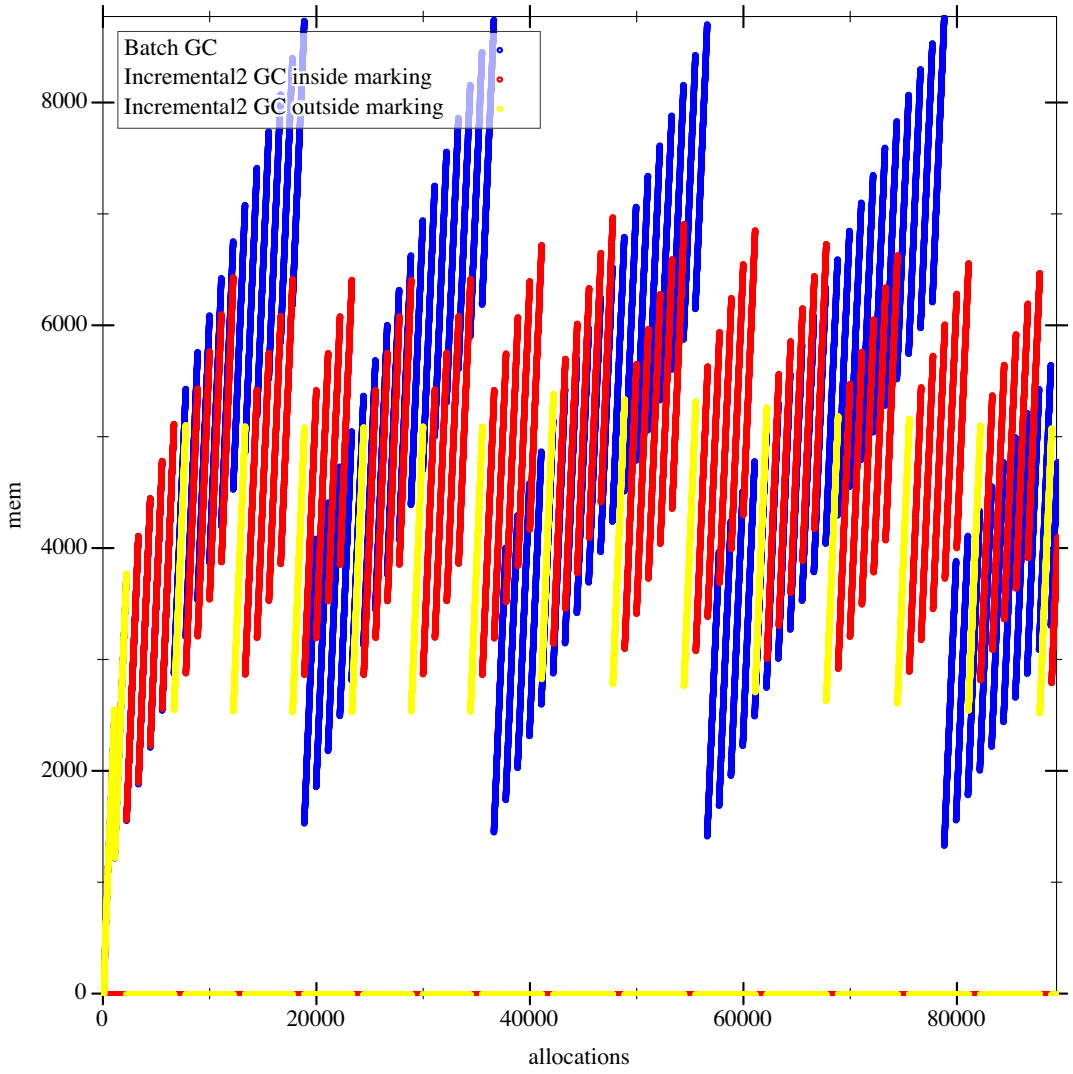

Figure 4.8. Memory Usage of HTML parser

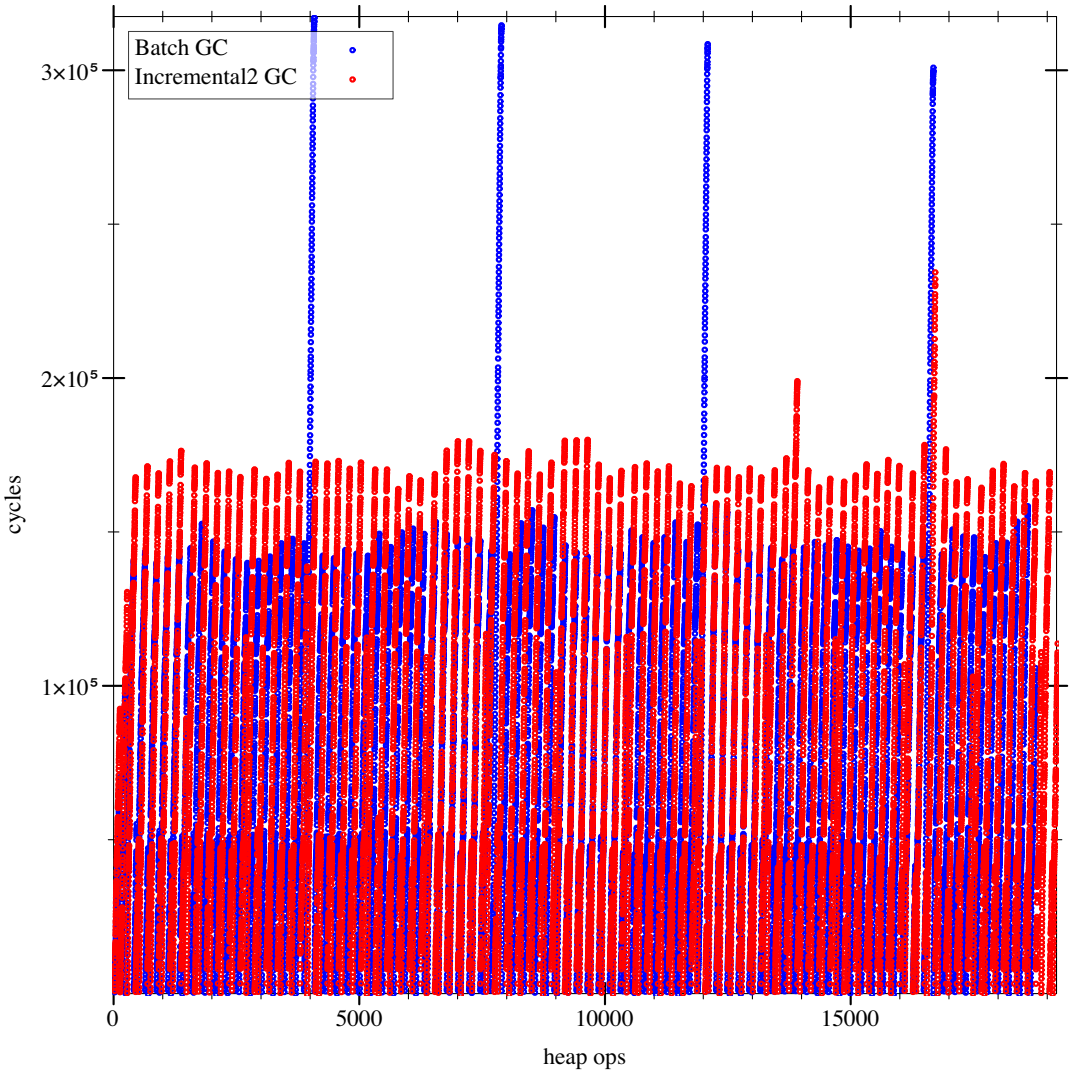

Figure 4.9. Running Time of HTML parser

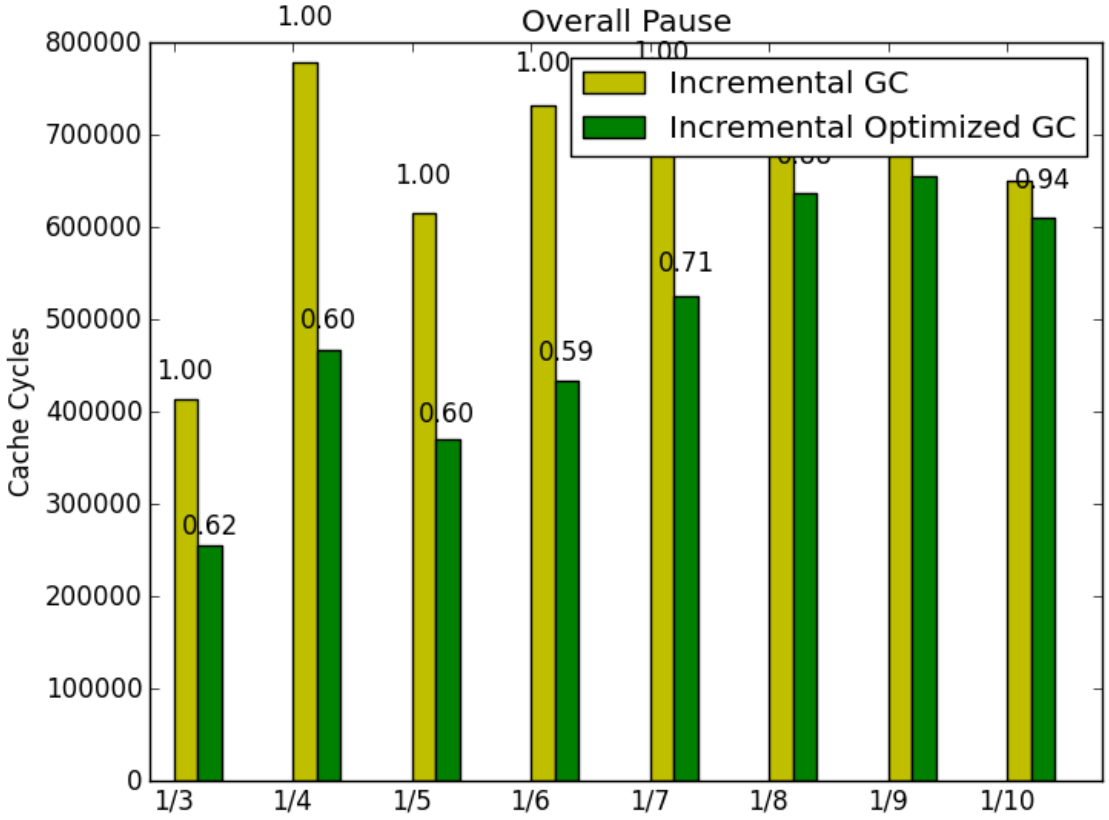

Figure 4.10. Overall Pause with different Generation Size Ratio

CHAPTER 5

Conclusion and Future Work

This thesis introduced a generational incremental garbage collector design that provides both short pauses and improved long-term performance. This work offers two main contribution. First, the optimal design gives the right timing to start the incremental GC on the older generation. Second, our implementation and evaluation demonstrate that plai/gc2 is a powerful experiment platform for GC designs.

5.1. Future Work

5.1.1. Benchmarks

We hope to be able to implement more complicated and interesting benchmark programs using plai/gc2/mutator. Though the limited benchmarks can show the improvement of our optimal garbage collector design, it is important to measure performance of the our implementations in the context of interactive applications in general. Unfortunately, till the point we write this thesis, plai/gc2 is still buggy in some places, and the limited Racket primitives it supports, as well as the current way to import primitives are still insufficient to implement complicated programs easily.

5.1.2. Dynamic Tracing Rate

The current implementation of generational incremental garbage collector does the same amount of generational copying during every incremental tracing round. For some programs, this rate is comparatively high, and unnecessarily increases the pausing time. For the other programs, this rate might not guarantee sufficient progress for every tracing round. The right direction to improve on this problem is to use dynamic tracing rate, which varies based on the memory usage garbage collectors observe along the process.

5.1.3. Advanced GC Data Structures

Without advanced data structures like sets and tables, the current implementation of garbage collectors cannot utilize techniques like implicit reclamation[**10**], and etc. So many interesting designs, yet not as vital as the incremental GC timing, were not evaluated in this thesis.

References

[1] Armstrong, J., and Virding, R. One pass real-time generational mark-sweep garbage collection. In *Memory Management*. Springer, 1995, pp. 313–322.

[2] Baker Jr, H. G. List processing in real time on a serial computer. *Communications of the ACM 21*, 4 (1978), 280–294.

[3] Bartlett, J. F. *Mostly-copying garbage collection picks up generations and C++*. Technical Note TN-12, Digital Equipment Corporation Western Research Laboratory, 1989.

[4] Cooper, G. H., Guha, A., Krishnamurthi, S., McCarthy, J., and Findler, R. B. Teaching garbage collection without implementing compiler or interpreters. In *Proceeding of the 44th ACM technical symposium on Computer science education* (2013), ACM, pp. 385–390.

[5] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM 21*, 11 (1978), 966–975.

[6] Hertz, M., and Berger, E. D. Quantifying the performance of garbage collection vs. explicit memory management. In *ACM SIGPLAN Notices* (2005), vol. 40, ACM, pp. 313–326.

[7] Lieberman, H., and Hewitt, C. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26*, 6 (1983), 419–429.

[8] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM 3*, 4 (1960), 184–195.

[9] Wilson, P. R. Uniprocessor garbage collection techniques. In *Memory Management*. Springer, 1992, pp. 1–42.

[10] Wilson, P. R., and Johnstone, M. S. Real-time non-copying garbage collection. In *ACM OOPSLA Workshop on Memory Management and Garbage Collection* (1993).

[11] Zhang, X., and Seltzer, M. Hbench: Java: An application-specific benchmarking framework for java virtual machines. *Concurrency and Computation: Practice and Experience 13*, 8-9 (2001), 775–792.