CMSC838E Final Project: Improving Cache Locality of Garbage Collector

Xiaolong Tian, xt01@umd.edu

1. Introduction

In the class project, we implemented Cheney's algorithm for garbage collection. Cheney's algorithm traverses links in the order of where the references are on the stack. What this means is that we can think of the following: A->B->C, A->D->E, where A, B, C, D, E are reachable boxes and, on the stack, reference to A is before references to B, and D, and they are before references to C and E. Then, A, B, and D will be copied from the from space to the to space, before C and E are copied. However, this order of copying objects is not good for cache locality. Let's take the example of a large linked list. By large, I mean that there are many elements in the linked list such that the list in total occupies numerous cache lines. A cache line is a contiguous memory, and it is loaded by the processor as a unit one at a time. Hence, to maximize the efficiency of using cache, we want each cache line to contain related data. For example, in the case of traversing a linked list, N1->N2->N3->N4, if N1 to N4 are all located in the same cache line, then we only have to load the cache line once to satisfy all memory reads in the traversal. There is only one cache miss at N1. However, if N1, N2, N3, and N4 are all located in different cache lines, each time we go to the next element in the linked list, there is a cache miss, and an entire cache line must be loaded. In the case that references to N1, N2, N3, and N4 are all located in contiguous stack locations, Cheney's algorithm approach will do just fine. However, elements of a linked list may be allocated, inserted, and removed on demand. N1 may first be added to the linked list, after which we allocated a large vector, then N2 will be added to the linked list. We may then allocate some strings and boxes, before allocating and appending N3, etc. Cheney's garbage collection approach will first copy N1, then the vector, then N2, then other objects, then N3, and so on. This will likely result in elements of the linked list in different cache lines.

In this final project, I modified the garbage collector to traverse the links in a depth first approach. Given the same example A->B->C, A->D->E, A, B, C, will be copied from the from space to the to space before D, and E is copied. DFS approaches are particularly good for linked structures such as linked list and trees. We can think of many application use cases where a DFS based traversal will improve cache locality. For example, in an expression tree, we need to traverse down an expression, evaluating its sub expressions. DFS approach will

make sure these subexpressions closer together in memory, whereas with the Cheney's approach, the subexpressions can be far away in memory if the tree is large. There are many other applications which can benefit from a depth first approach such as scene graphs in game engines, directories in file systems, object-oriented hierarchies, just to name a few.

In addition, I add a hot cold object segregation to the DFS based garbage collector. Hot cold object segregation means that we classify the objects into cold objects and hot objects. Cold objects are those that are accessed infrequently and hot objects are those that are accessed frequently. From space and to space are divided into a hot region and a cold region. Hot objects are copied from to the hot region and cold objects are copied to the cold region. Examples of cold objects can be configuration variables that are only used at the beginning, objects from sleeping users, metadata, etc. Examples of hot objects could be loop counters, user IDs, etc. By grouping hot objects and cold objects separately, frequently accessed objects are loaded together in the same cache lines. This results in fewer cache misses and improve cache locality.

2. Implementation

2.1 Depth First Garbage Collection

In Cheney's algorithm, there are two phases. In the first phase, we traverse through the stack, for each pointer, we copy the object from the from space to the to space, and leave a forwarding pointer. If it is already a forwarding pointer, then simply points to the forwarding address. In the second phase, we do the same in the to space for elements of objects. Similarly, in the depth first approach, we traverse the stack looking for pointers, copying the object from the from space to the to space (if it is not already a forwarding address). It is important here to also leave a forwarding pointer. This is because in the case of A->B->C (all boxes), if reference to B is located before reference A in the stack, B and C will be copied first. If there is no forwarding pointer, when reference to A is encountered in the stack, we would copy B and C redundantly leading to incorrect behavior. When an object is copied to the to space, each element of the object, such as fst and snd for a cons, element of a vect, are pushed unto a DFS stack. Elements of strings, which are not capable of being references, are not pushed unto the stack.

Then, we proceed to process the stack. We pop the top of the stack, check to see if it is a reference. If it is, then we process the reference in the same way, that is copy to the to space (or directly reassign to forwarding address), leaves a forwarding ptr, push its elements unto the stack. This is done iteratively until the stack is empty. When the stack is

empty, we move to the next address in the program stack, and so on. Hence, only one phase is required for the DFS approach.

We give a simple example illustrating this. Consider a vector of four elements, a box pointer, a cons pointer, and two integers. The memory layout for it is drawn in Figure 1.

1) The whole vector is copied to the to space. Each element of the vector is pushed onto the DFS stack.
2) We pop elements of the DFS stack until we encounter a reference, the cons ptr 1.
3) The cons pointed to by cons ptr 1 is copied to the to space, and the first and second of cons is pushed unto the DFS stack.
4) We pop elements of the DFS stack until we encounter a reference, the box ptr 1.
5) The box pointed to by box ptr 1 is copied to the to space, the value box ptr 2, which also happens to be a box ptr, is pushed unto the stack.
6) We pop elements of the DFS stack until we encounter a reference, the box ptr 2.
7) The box pointed to by box ptr 2 is copied to the to space, the value is pushed unto the stack.
8) We pop elements of the stack until it is empty.
9) We move to the next element in the program stack (not shown).

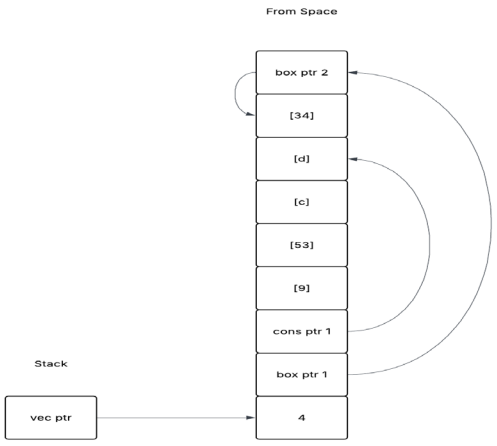

Figure 1 Memory Layout of a Vector

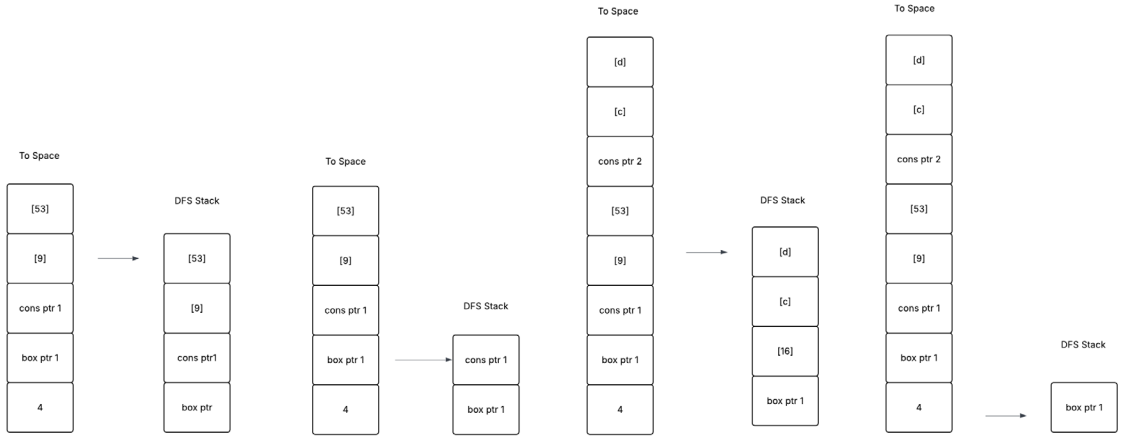

Figure 2 Step 3-5

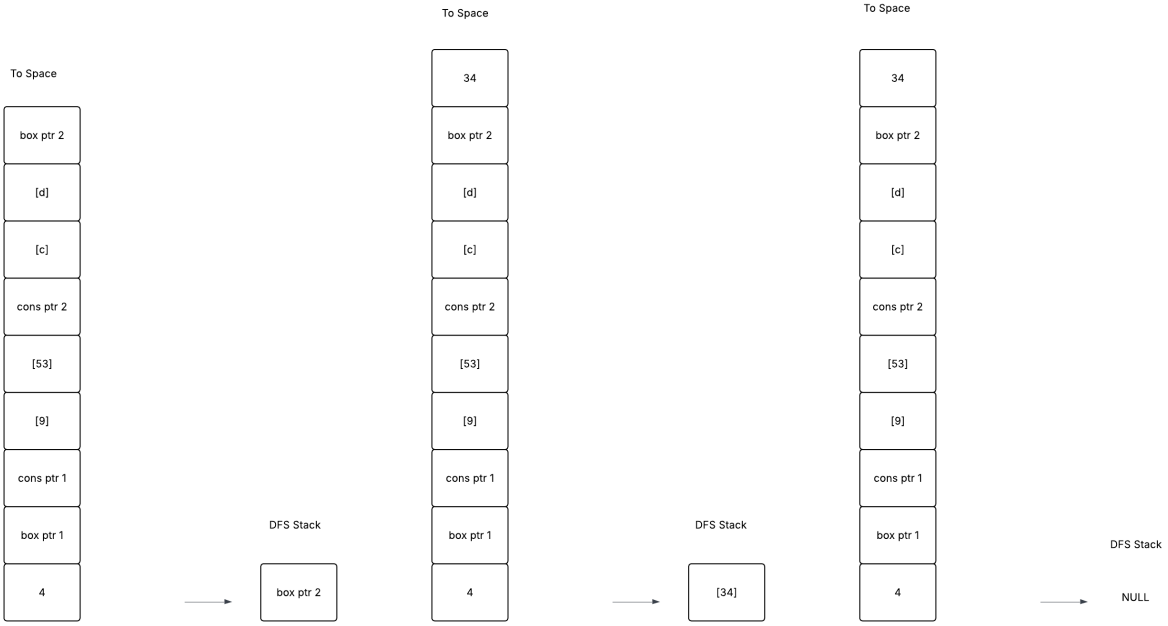

Figure 3 Step 6-8

In Cheney's two phases approach, a queue is needed to store the types of each object being copied from the from space to the to space. However, when we pop the reference (more accurately, the address of the reference) from the DFS stack, we have the reference and can tell what type of object the reference is pointing to. Furthermore, we know that each reference on the DFS stack is element of a vector, a cons, etc., so we could directly go ahead and process it. In this way, queue or stack to store the types is not necessary.

The key steps, checking to see if a reference contains a forwarding address, if not, copy the object to the to space, leave a forwarding address, and push elements unto the DFS stack are the same for every object, on the program stack and on the from space. Hence, this is encapsulated into the C function with the signature,

```
void process_element(val_t interior, type_t interior_type, val_t* curr, val_t* rbx,
val_t** rbx_double)
```

, in file gc_dfs.c. Essentially, the DFS garbage collector implemented by a DFS stack together with the function process_element.

2.2 Hot Cold Object Segregation

To implement hot cold object segregation on top of DFS garbage collection, we must first decide how to classify an object as either hot or cold. In this final project, in order to limit the scope and focus on the garbage collector implementation, we use the same idea that we can assume deciding when to call the garbage collector is handled elsewhere. All objects are initially cold, and we provide a primitive, mark-hot, to mark an object as hot. At the end of garbage collection, all objects are mark as cold.

```
; Example use of mark-hot primitive
(let ([n1 (box 7)])
(begin
  (mark-hot n1)
  (collect-garbage)))
```

That is, we assume that we know which object is hot and which object is cold. This may be done by the compiler generating instructions dynamically or future additional work on top of our existing implementation. The primitive is added and linked in the same way as the collect-garbage primitive by adding another matched pattern to compile-ops.rkt.

```
['mark-hot
   (seq (Mov rdi rax)
      pad-stack
      (Call 'mark_hot)
      unpad-stack
      (Mov rax (value->bits (void))))]
```

The mark-hot primitive relies on a bit map to store information about hotness of objects. We could have just used a bool or int array of heap size. However, using a bit map gives us the assurance that minimum memory overhead is occurred. Note that bit map is a very common data structure with well-known implementation and not essential to the garbage collection algorithm (only an optional memory optimization). I did not write the bit map

implementation. It is obtained online with slight modifications and included as a library. No other code is obtained online. The mark-hot primitive will pass the address of the object as input, the offset from the from space is calculated and the bit from the offset is set to 1.

During garbage collection, the to space to divided evenly into a hot region and a hot region. Currently, both of the cold region and the hot region grows upwards. Only the top of the hot region is returned from the garbage collection routine. Obviously, this is not optimal since only half of the from space (consisting of the hot region) can be used to allocate new objects. More complicated schemes can be thought of. For example, the cold region could be allocated on the top of the to space and grow down. This will ensure memory efficiency.

Recall the process_element routine from the discussion of DFS garbage collection. It checks to see if a reference contains a forwarding address, if not, copy the object to the to space, leave a forwarding address, and push elements unto the DFS stack are the same for every object, on the program stack and on the from space. We could simply modify this to check if the reference points to an hot object or a cold object, and copy to the hot or the cold region accordingly.

2.3 Code Directory Structure

A header file bit_array.h is added. Some other files are also modified, such as compile-ops.rkt and parse.rkt to add in the mark-hot primitive. The implementation for DFS garbage collector is in gc_dfs.c file. The implementation for DFS + hot cold object segregation is in gc_hc.c file. The original implementation for Cheney's algorithm is in gc_cheney.c file. The Makefile will compile the code in gc.c file. Hence, to use and test any of the implementation, first copy the file to gc.c and then type make. The test codes given in this report are contained in test.rkt file. To try one of the test cases, comment out all other test cases and generate executable using test.rkt file.

3. Results

In the results section, we evaluated a number of use cases, comparing Cheney's, the DFS approach, as well as hot cold object segregation. We show that DFS approach and hot cold object segregation can put related objects closer in memory, hence resulting in better cache locality.

For a particular use case, we give an example of a program that allocates some objects. We then give illustrations of the memory spaces before and after garbage collection by Cheney's and DFS. We also give the memory printed out by the program from running both garbage collectors in the appendix.txt file. Note that however, the memory printed out by

the program may not directly correspond to the illustrations. This is because the garbage collection may flip it upside down, two children of a node may be in different order, etc. However, the differences are negligible and do not impact the main points of our discussion.

User Case 1: Binary Tree

Binary tree is a commonly used data structure. Consider the following full and complete binary tree in Figure 4 built by the Racket program in Code 1.

```
; Code 1, Full tree
(let ([n1 (cons 6 7)])
 (let ([n2 (cons 14 15)])
 (let ([n4 (cons 9 10)])
 (let ([n5 (cons 3 4)])
 (let ([n8 (cons 1 2)])
 (let ([n9 (cons 19 23)])
 (let ([n11 (cons 34 11)])
 (let ([n12 (cons 2 8)])
 (let ([n3 (cons n1 n2)])
 (let ([n6 (cons n4 n5)])
 (let ([n10 (cons n8 n9)])
 (let ([n13 (cons n11 n12)])
 (let ([n7 (cons n6 n3)])
 (let ([n14 (cons n10 n13)])
 (let ([n15 (cons n7 n14)])
  (begin
    (let ((y (box 8)))
          2
    )

  (collect-garbage)))))))))))))))
)
```

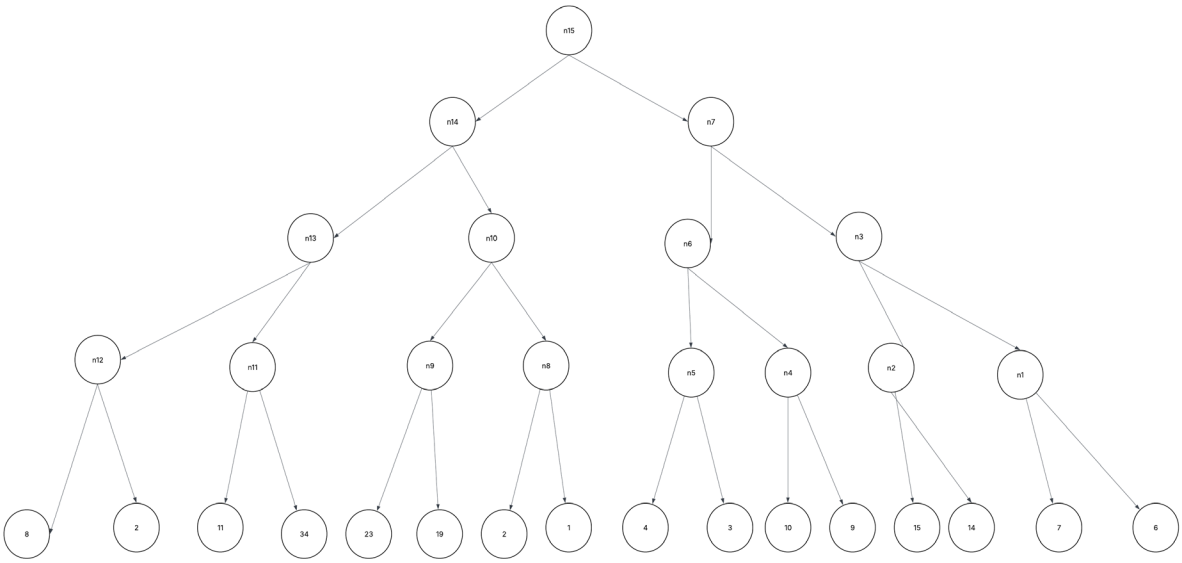

Figure 4 Full Tree

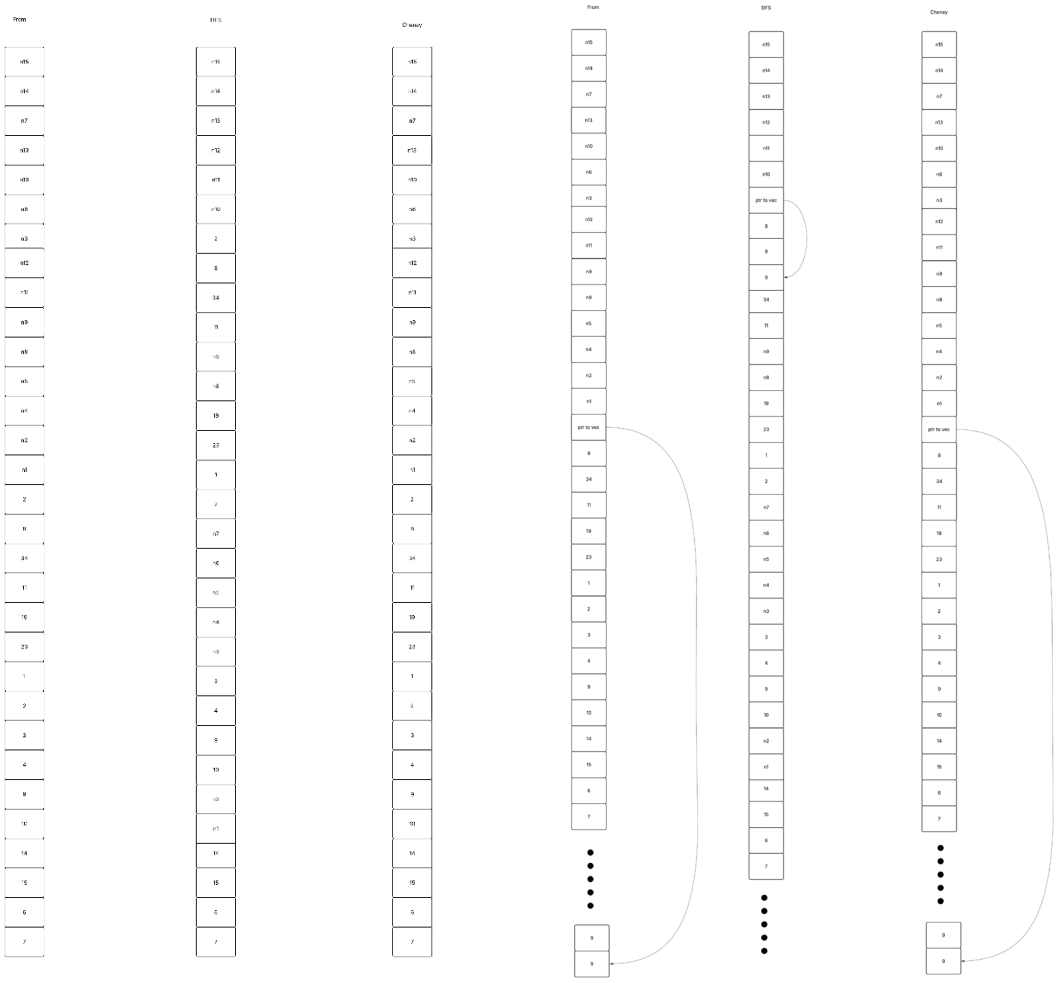

Figure 5 Results of Cheney's and DFS garbage collection. (Left 3) Full and complete tree. (Right 3) Gap between tree and an inserted node.

A cons is used to represent each node. There are four levels of this binary tree. Figure 5 gives the memory layout of the tree in the from space. Notice that because of the order in which the nodes are allocated, all the leaves nodes are close to each other, then all the nodes in the next layer are close to each other, following by the next layer, etc. When the Cheney's garbage collector is run on this binary tree, the order the nodes care copied to the to space follows the order of references on the stack, this means that the root is first copied to the to space, followed by all the nodes of layer 1, and then all the nodes of layer 2, and so on (shown on Figure 5). In comparison, our DFS based garbage collector will copy the objects in such a way that n1 and n2 are close to each other, n3 and n4 are close to each other, n5 and n6 are close to each other, etc. In other words, each "local region", i.e., elements of a subtree are close to each other in memory. Hence, suppose that we are at n13, and wish to access it children, n12, and n11, or its parent, n14, they are all located

close to each other in memory and can result in good cache locality. However, if we look at the memory resulting from Cheney's garbage collection, n13's children and parents are exactly one layer of nodes away and this result in poor cache locality for large trees.

```
; Code 2, Gap between tree and inserted node
(let ([n16 (make-vector 2 9)])
 (let ([v1 (make-vector 20 4)])
 (let ([n1 (cons 6 7)])
 (let ([n2 (cons 14 15)])
 (let ([n4 (cons 9 10)])
 (let ([n5 (cons 3 4)])
 (let ([n8 (cons 1 2)])
 (let ([n9 (cons 19 23)])
 (let ([n11 (cons 34 11)])
 (let ([n12 (make-vector 2 8)])
 (let ([n3 (cons n1 n2)])
 (let ([n6 (cons n4 n5)])
 (let ([n10 (cons n8 n9)])
 (let ([n13 (cons n11 n12)])
 (let ([n7 (cons n6 n3)])
 (let ([n14 (cons n10 n13)])
 (let ([n15 (cons n7 n14)])
  (begin

  (vector-set! n12 1 n16)
  (collect-garbage)))))))))))))))))))
)
```

In addition, binary trees are usually not static. Nodes can be removed from or added to a tree. A key advantage of a DFS based garbage collector is that it can compact related objects together even if they were not close in memory before the garbage collection. For example, consider the following Rachet program (Code 2) that allocates a tree. After it allocates the full tree, it also allocates a large vector and another node. Then it also appends that node to the tree. We use a vector for a node here because the loot compiler does not support a cons-set primitive. The representation, however, is not important. The high-level idea here is that the appended node can be quite far away from the tree and its parent node in memory. This may due to the nodes being allocated at different times and other objects being allocated in between. A DFS approach, as seen from the memory layout in Figure 5 (Right 3), will put the appended node next to the tree, resulting in much better cache locality than the Cheney's approach. The exact memory layout printed out by the running program is given in the appendix.txt file.

Use Case 2: Linked List

```
; Code 3, Linked list where nodes are scattered throughout memory
(let ([b1 (box 0)])
 (let ([b2 (box b1)])
 (let ([c1 (cons 6 7)])
 (let ([b3 (box b2)])
 (let ([v1 (make-vector 15 8)])
 (let ([b4 (box b3)])
 (let ([b1 (make-string 23 #\a)])
 (let ([b5 (box b4)])
  (begin
    (let ((y (box 8)))
          2
    )

  (collect-garbage)))))))))
)
```

Linked list is a very commonly used data structure. For a linked list whose elements are allocated in contiguous memory, a Cheney's vs a DFS garbage collection makes no difference. However, that is rarely the case for a linked list. Nodes are added to and removed from the linked list at different points in a program. Other large objects and data structures may be allocated between. The nodes of the linked list may be scattered around in memory. As we mentioned before, a DFS approach can compact related objects together even if they were not close in memory previously. We give an example of a Rachet program (Code 2) that allocates nodes of a linked list while allocating other objects between the nodes. We show that the DFS approach group all the nodes of the linked list together (Figure 5) resulting in better cache locality.

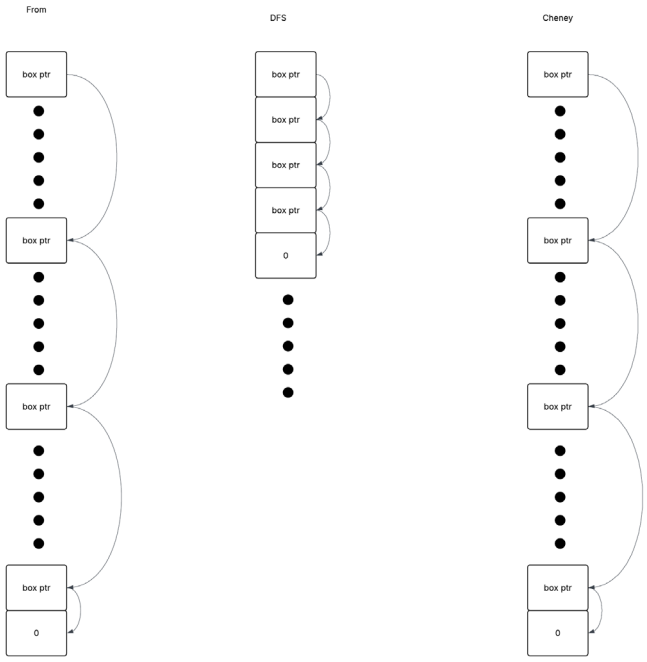

Figure 5 Results of Cheney's and DFS Garbage Collection

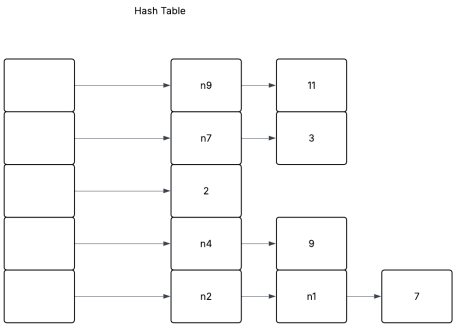

Figure 6 Hash Table

Use Case 3: Hash Table

```
; Code 4, Hash Table with small number of frequently accessed key value pairs
(let ([n1 (box 7)])
  (let ([n2 (box n1)])
  (let ([n3 (box n2)])
  (let ([n4 (box 9)])
  (let ([n5 (box n4)])
  (let ([n6 (box 2)])
  (let ([n7 (box 3)])
  (let ([n8 (box n7)])
```

```
 (let ([n9 (box 11)])
 (let ([n10 (box n9)])
 (let ([h1 (make-vector 5 0)])
  (begin
  (vector-set! h1 0 n3)
  (begin
  (vector-set! h1 1 n5)
  (begin
  (vector-set! h1 2 n6)
  (begin
  (vector-set! h1 3 n8)
  (begin
  (vector-set! h1 4 n10)
  (begin
  (mark-hot h1)
  (begin
  (mark-hot n1)
  (begin
  (mark-hot n2)
  (begin
  (mark-hot n3)
  (collect-garbage))))))))))))))))))))
)
```

We move on from discussing results of DFS to talk about the results of implementing hot cold object segregation on top of DFS garbage collection. Hash table is an important data structure in computer science. We believe that hash table can significantly benefit from hot cold object segregation. Consider for example, a fictional hash table that stores active sessions information for a web server. The key of the hash table is a string (session key) and the value of the hash table is a struct that stores user session information. There may be millions of users, but only a small percentage of them are frequently accessed. Furthermore, hash table often resolves collision using chaining. When resolving a collision, a linked list may be searched. The frequently accessed linked list will be in the hot region, and the infrequently accessed linked list will in the cold region. This can significantly improve cache locality.

We provide a Racket program that allocates a hash table (See Figure 6) with chaining collision resolution in Code 4. The program uses a vector of size 5. Each element of the vector has a linked list of various small sizes. The memory layout is shown on Figure 7. The program also marks the linked list associated with the first element, n1, n2, n3 as hot. This represents the frequently accessed key value pairs of the hash table. Of course, the hash

table is also marked as hot as it is accessed. We show the memory layout after cold hot object segregation garbage collection. The hash table (vector) and the linked list nodes, n1, n2, and n3 are copied to the hot region, where as the infrequently accessed key values pairs are now moved to the cold region. This is obviously a small example, but it demonstrates that with a large hash tables containing millions of elements, frequently accessed key value pairs will be located close to each other in memory, leading to better cache locality and improved performance.

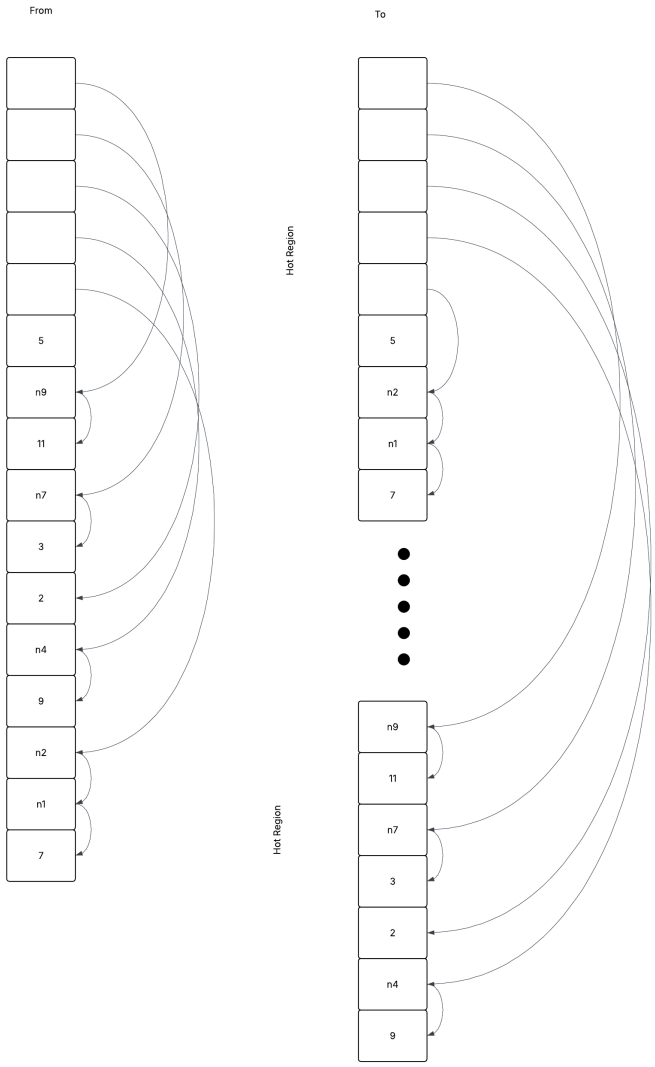

Figure 7 Hot Cold Object Segregation Garbage Collection on Hash Table

4. Conclusion

Cheney's algorithm traverses links in the order of where the references are on the stack. In many cases, this leads to related objects not being located close to each other, resulting in poor cache locality and performance of the program. In this final project, I implemented a garbage collector that traverses the links in a depth first manner, and additionally I added hot cold object segregation. I evaluated my garbage collector through example use cases such as trees, linked lists with nodes scattered throughout memory, hash table with small number of frequently accessed key value pairs. These use cases showed that DFS approach and hot cold object segregation can significantly improve program cache locality.