FRAMEWORK FOR ANALYZING GARBAGE COLLECTION *

Matthew Hertz Neil Immerman J Eliot B Moss
Dept. of Computer Science University of Massachusetts Amherst, MA 01003
{hertz, immerman, moss}@cs.umass.edu

Abstract While the design of garbage collection algorithms has come of age, the analysis of these algorithms is still in its infancy. Current analyses are limited to merely documenting costs of individual collector executions; conclusive results, measuring across entire programs, require a theoretical foundation from which proofs can be offered. A theoretical foundation also allows abstract examination of garbage collection, enabling new designs without worrying about implementation details. We propose a theoretical framework for analyzing garbage collection algorithms and show how our framework could compute the efficiency (time cost) of garbage collectors. The central novelty of our proposed framework is its capacity to analyze costs of garbage collection over an entire program execution.

In work on garbage collection, one frequently uses heap traces, which require determining the exact point in program execution at which each heap allocated object "dies" (becomes unreachable). The framework inspired a new trace generation algorithm, Merlin, which runs more than 800 times faster than previous methods for generating accurate traces [11]. The central new result of this paper is using the framework to prove that Merlin's asymptotic running time is optimal for trace generation.

1. Introduction

Most modern computer languages use a heap to hold objects allocated dynamically during the running of a program and a garbage collector to remove no longer needed objects from the heap. As use of these modern computer languages is increasing dramatically, it is very important that garbage collection run quickly. Towards this goal, many different garbage collection algorithms, optimizations, and techniques have been proposed and studied, e.g., [1, 2, 3, 10, 12, 19]. The experimental results document running times, relative volume of objects examined and copied, and other dynamically generated metrics from a small set of benchmarks. While this information illustrates

*This work is supported by NSF ITR grant CCR-0085792, NSF CCR-9877078, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

and convinces, it is unable to *prove* the arguments being made. For conclusive results, arguments need a theoretical foundation from which proofs can be offered. A strong theoretical foundation also supports abstract analyses of garbage collection, enabling a more thoughtful look at where opportunities for improvement exist.

This paper presents a new theoretical framework for examining garbage collection. Our framework is robust enough to capture the behavior of a garbage collector over the execution of a program, but abstract enough not to require any specific fashion of implementation. This allows the framework to be used with most garbage collection algorithms, to prove that collectors have several important properties, and to be expanded easily to include other analyses. While this is an important feature of our framework, its key feature is computing a garbage collector's asymptotic running time (or other costs) over an entire program execution, and not just a single invocation. With this, our framework can be used to prove optimality for garbage collection and related algorithms.

As an example of our framework's usefulness, we analyze the Merlin trace generation algorithm. Prior research has described Merlin and discussed its running time [11]; using our framework, we formally prove its asymptotic running time and that this time is optimal for trace generation.

Sections 2 and 3 of this paper discuss the structures and graphs that make up our framework. Section 4 uses the framework in a series of proofs of algorithmic requirements, asymptotic running times, and optimal running times. Finally, Section 7 summarizes these results.

2. Structures Used

Many modern computer languages allow objects to be allocated dynamically into a heap during program execution. This makes writing the program easier, but requires that objects be freed during program execution to avoid running out of memory. Many languages (such as Lisp, Smalltalk, and Java) use garbage collection (GC) to reclaim memory automatically, because GC increases program safety and makes the programmer's job easier. Difficulty arises in limiting the amount of time used for automated memory reclamation.

It has long been understood that a program's heap memory can be envisioned as a graph. Thus, our framework presents the analyses as graph theoretic problems. We first explain how a program allocates objects dynamically and how garbage collectors determine which objects may be freed. We then propose a series of graphs that model the heap and capture the execution of the program and garbage collector.

2.1. Program Structures

Before explaining our framework, it is important to understand what "objects" are and how programs use objects allocated dynamically into the heap.

Objects and References— We model each object as a finite mapping of keys to reference values, i.e., locations in the heap; each key names a unique field of the object. The fields are typed; they may contain values of primitive data types (such as integers) or may reference other objects. A *referring field* may refer to an object within the

heap or may be null (i.e., does not refer to any object). Because garbage collection is concerned only with how objects refer to one another, our framework considers the content only of fields that may contain references to other objects. When primitive data types and referring types cannot be disambiguated (e.g., C++ integer fields may contain references), the framework must consider the contents of all fields.

The Heap and Program Roots— A program's *heap* exists within an address space and holds the objects dynamically allocated during the program's execution. When allocating an object into the heap, the memory manager reserves space in the heap for each field and sets the map from the object's keys. Typically, the memory for an object is contiguous and a field's mapping key is the offset from the start of the object to that field. Objects reside in the heap because the program needs them; unneeded objects may be removed from the heap. A *garbage collector* reclaims these obsolete objects.

Determining which objects are no longer needed (will not be used again) requires knowledge of the future. As this is not always possible (true liveness is an undecidable property), *reachability-based* garbage collectors reclaim only objects they can demonstrate that the program will not use.

Objects within the heap are allocated during program execution; the program, not knowing where in memory the objects reside, cannot access them directly. To access the heap the program relies on its *root set*, locations the program accesses directly that may hold references into the heap. Like an object, the root set is a mapping of keys (each representing a unique *root* with a referring type) to reference values. Unlike an object, this mapping is neither bounded nor fixed. Keys may be added and removed, because *root references* are found in, for example, the program stack and the static and global variable table, which change size during execution. The program uses dynamically allocated objects only through the root set; objects not reachable from the root set cannot be used by the program. Garbage collectors determine which objects the program may use and which may be safely removed from the heap by analyzing reachability from the root set. For convenience we use "reachable" and "live" interchangeably, and likewise "unreachable" and "dead".

2.2. Heap State

The current state of the heap is defined by the objects and references within the heap and the program's root set. To allow for further analyses, we also include the set of objects that have been identified by the collector as unreachable. We represent the *heap state* (the current state of the heap) as a rooted, directed multi-graph. We call this multi-graph the *heap state multi-graph*, and express it as $H = (L, D, r, E)$.[1]

The set of vertices of H, $V = L \cup D$, includes a vertex for each object allocated in the heap, and a special vertex, called the root vertex and designated as r, representing the root set. The set D contains the vertices identified as dead, while L includes the remaining vertices ($L = V - D$, so $L \cap D = \emptyset$). Since roots are always reachable, r must be in L. Vertices represent only the existence of an object. This representation

[1]Other elements could exist within the heap state multi-graph; we include only those elements needed for this paper.

makes modeling garbage collectors easier by abstracting away implementation details of objects' actual locations in the address space.

Edges in the multi-graph represent references to objects in the heap. The edge multiset, E, contains an edge $(\langle v,n\rangle,o)$ if and only if object v at the field mapped to by key n contains a reference to object o. Notice that v may be r, in which case n is the key to a root that refers to o.

This structure is not a graph, but a multi-graph, because an object may have multiple fields that refer to the same object. Likewise, there may be multiple root locations that refer to the same object. Just as a garbage collector analyzes the heap using the root set and objects, this multi-graph can be analyzed for the relationships between the root vertex and other vertices.

2.3. Reachability

Given $H = (L,D,r,E)$, we say v refers to o (in H), written $refers_H(v,o)$, if and only if v has at least one field that refers to object o: $refers_H(v,o) = (\exists n)((\langle v,n\rangle,o) \in E)$.

Extending this definition, we say v *reaches* o (in H), written $reaches_H(v,o)$, if and only if v and o are equal or o is in the transitive closure of objects to which v refers: $reaches_H(v,o) = (v = o \vee (\exists p)(refers_H(v,p) \wedge reaches_H(p,o))$.

Given the importance of objects that the program can access from the root set, an object o is *reachable* (in H) if and only if r reaches o ($reachable_H(o) = reaches_H(r,o)$). Reachable objects may be used in the future by the program. Objects not reachable in the heap state (e.g., objects not in the transitive closure of the root set) cannot be accessed by the program, so garbage collectors remove only *unreachable* objects.

A heap state multi-graph $H = (L,D,r,E)$ is *well-formed* if and only if all reachable objects are in L: $L \supseteq \{o | reachable_H(o)\}$. From here on we are concerned only with well-formed heap state multi-graphs.

2.4. Program Actions

The heap state and its corresponding graph are useful for analyzing snapshots of a program. But programs and their heaps are dynamic entities: the program mutates its heap as it runs. These changes include objects being dynamically allocated, fields of objects being updated, and objects being passed to and from functions. These changes occur throughout program execution and a dynamic memory manager must be capable of handling all of them.

Interesting Program Actions— While programs perform many operations, our framework is interested only in those actions that affect the heap. Though the causes of these mutations are language-specific, we group actions by their effect on the heap: object allocation, root creation, root deletion, field reference creation, field reference deletion, and program termination.[2]

[2]Other actions could be included; these primitives are quite general and can be combined. We distinguish root and heap reference actions because many algorithms treat them differently. We specifically exclude GC behavior from program actions.

Action Name	*Effect*	*Precondition*
Object Allocation	$L_{t+1} = L_t \cup \{o\}$; $E_{t+1} = E_t \cup \{(\langle r,n\rangle, o)\}$	$o \notin L_t \cup D_t \wedge \neg(\exists o')((\langle r,n\rangle, o') \in E_t$
Root Creation	$E_{t+1} = E_t \cup \{(\langle r,n\rangle, o)\}$	$reachable_{H_t}(o) \wedge \neg(\exists o')((\langle r,n\rangle, o') \in E_t)$
Root Deletion	$E_{t+1} = E_t - \{(\langle r,n\rangle, o)\}$	$(\langle r,n\rangle, o) \in E_t$
Heap Reference Creation	$E_{t+1} = E_t \cup \{(\langle v,n\rangle, o)\}$	$reachable_{H_t}(v) \wedge reachable_{H_t}(o) \wedge \neg(\exists o')((\langle v,n\rangle, o') \in E_t)$
Heap Reference Deletion	$E_{t+1} = E_t - \{(\langle v,n\rangle, o)\}$	$reachable_{H_t}(v) \wedge (\langle v,n\rangle, o) \in E_t$
Program Termination	$E_{t+1} = E_t - \{(\langle r,n\rangle, o) \mid (\langle r,n\rangle, o) \in E_t\}$	None

Table 1. Definition of action a_t. Only changes from H_t to H_{t+1} are listed.

We describe the effect each action has on the heap state with respect to the heap state multi-graph at time t, $H_t = (L_t, D_t, r, E_t)$, and the multi-graph following the action, $H_{t+1} = (L_{t+1}, D_{t+1}, r, E_{t+1})$.[3] We additionally describe the preconditions necessary within H_t for the possible actions a_t. A precise mathematical definition of these effects and limitations can be found in Table 1. The following paragraphs provides a simple description of each of the actions:

Object Allocation actions occur when an object is allocated in the heap. An object allocation action defines a new vertex to be added to the set of live vertices and the key value of the root that references the newly allocated vertex.

Root Creation actions occur when a root reference to an object is created. The root creation action defines an new edge to be added to the edge set from the root vertex to a reachable vertex.

Root Deletion actions remove an existing edge from the root vertex to a vertex in the set of live vertices. This action is equivalent to deleting a root reference or making a root null.

Heap Reference Creation actions occur when the program updates a heap object's unused field updated to reference another object. These actions add an edge to the edge set from the source vertex to the target vertex.

Heap Reference Deletion actions specify an edge in the edge set between two vertices that is removed. Heap reference deletion actions occur whenever an object in the heap has a non-null field made null.

Program Termination actions occur when the program execution ends. Program termination may occur at any time and deletes the root set (removes any edge whose source is the root vertex).

In our framework, as in program execution, only existing references may be removed, each object field contains at most one reference, and only reachable objects may be involved in actions. Since the heap state multi-graph reflects the heap, the

[3] Actions may also be considered functions producing H_{+1} from H_t; we do this when it is convenient.

$$\underset{(L_0,D_0,r,E_0)}{H_0} \xrightarrow{a_1} \underset{(L_1,D_1,r,E_1)}{H_1} \xrightarrow{a_2} \ldots \xrightarrow{a_T} \underset{(L_T,D_T,r,E_T)}{H_T}$$

Figure 1. Example Program History

program actions mirror the changes in the heap. *By construction* programs cannot attempt actions whose preconditions are not met, so we need not consider the possibility. Further, it is easy to see that program actions preserve well-formedness of heap state multi-graphs (since they cannot remove vertices nor cause unreachable objects to become reachable).

2.5. Program History

Every program begins with the same heap. This *initial heap state* is represented by the multi-graph $H_0 = (L_0, D_0, r, E_0)$. This graph has only one vertex (the root vertex) and an empty edge multiset ($L_0 = \{r\}, D_O = \emptyset, E_0 = \emptyset$).

We also consider a program's heap state following program termination. *Final heap state* multi-graphs, designated $H_T = (L_T, D_T, r, E_T)$, have a vertex (in $L_T \cup D_T$) for each object allocated into the heap, plus the root vertex. Final heap state multi-graphs cannot contain edges from the root vertex, so only the root vertex is reachable in these multi-graphs.

The initial and final heap states do not contain much information about the program execution, but the entire run is necessary to analyze GC algorithms and optimizations. To record this information, our framework uses a *program history*. The program history begins with the initial heap state multi-graph, H_0, and the first program action, a_1. From this multi-graph and action, we build the successor heap state multi-graph, H_1, then add action, a_2, and so on. The program history continues up to the program termination action (a_T) and final heap state multi-graph, H_T. Figure 1 illustrates a program history.

Since all programs start from the initial heap state, and each program action is deterministic, the actions alone are sufficient to recreate the program history. By replaying the actions, GC can be simulated or, via profile feedback [4], tuned. Files called *heap traces* store the actions (and an additional piece of information, as we explain in Section 3) for these purposes. Since it works in a manner similar to heap traces, our program history is intuitive to use.

2.6. Null and Reachable Multi-Graphs

Using the program history, our framework can compute the *null heap state multi-graph* for each time step. The null heap state multi-graph at time t, $H_t^\emptyset = (L_t^\emptyset, D_t^\emptyset, r, E_t)$, is the heap state multi-graph where no objects have been determined to be dead (e.g., $D_t^\emptyset = \emptyset$).

Using the program history, the time when each object becomes unreachable can be determined. Given $H_t = (L_t, D_t, r, E_t)$, the *reachable heap state multi-graph* is $H_t^* = Live(H_t) = (L_t^*, D_t^*, r, E_t)$. The reachable multi-graph specially defines L and D: $L_t^* = \{v \in L_t | reachable_{H_t}(v)\}$; $D_t^* = V_t - L_t^*$, that is, L_t^* is exactly the set of reachable

vertices, and D_t^* the remainder (the $*$ superscript is intended to suggest the optimal, i.e., smallest possible, L_t set.)

Given a heap state multi-graph H_t, we define the *reduced heap state multi-graph*, $H_t^R = Reduce(H_t)$, as the heap state multi-graph (L_t^R, D_t^R, r, E_t^R), where: $L_t^R = L_t$, $D_t^R = \emptyset$, $E_t^R = \{(\langle v,n\rangle, o) \in E_t | v \in L_t\}$. This reduction removes those vertices identified as dead, and any edges from these vertices, from the multi-graph. By removing edges and vertices that are known to be unnecessary, the reduced heap state multi-graph resembles the physical heap following GC.

Finally, given a heap state multi-graph H_t, we define the *reduced reachable heap state multi-graph*, H_t^-, as the heap state multi-graph $Reduce(Live(H_t))$.

The null, reachable, and reduced reachable heap state multi-graphs are similar: their reductions via $Reduce \circ Live$ are the same. Further, since program actions can manipulate only reachable objects, if we apply a_{t+1} to $H_t^\emptyset$, H_t^*, and H_t^-, we get analogous results, a fact we state precisely and prove in a moment. First we argue that if we have a well-formed heap state $H_t^\emptyset$ and corresponding action a_{t+1}, then a_{t+1} is legal for H_t^* and H_t^-.

Lemma 1 *If $H_t^\emptyset$ fulfills the preconditions of a_{t+1}, then so do H_t^* and H_t^-.*

Proof Assume H_t satisfies the preconditions of a_{t+1}. First, we note that all vertices reachable in $H_t^\emptyset$ are in L_t^* and thus in L_t^-, and that $L_t^- \cup D_t^- \subseteq L_t^* \cup D_t^* = L_t \cup D_t$. Thus if a_{t+1} is an object allocation, its precondition is satisfied in H_t^* and H_t^-. Reference creation and deletion preconditions are trivially satisfied because they mention only reachable objects and edges between reachable objects. ■

Now we state and prove a stronger relationship for program histories and their corresponding reachable and reduced reachable heap states:

Theorem 2 *If a_{t+1} takes H_t to H_{t+1}, then $a_{t+1} \circ Live$ takes H_t^* to H_{t+1}^* and $a_{t+1} \circ Live \circ Reduce$ takes H_t^- to H_{t+1}^-.*

Proof Assume $H_t = a_t(H_{t-1})$ (interpreting a_t as a function, and implying that H_{t-1} is well-formed and meets the preconditions of a_t). H_{t-1}^* differs from H_{t-1} only in the partitioning of the vertices into L and D sets, and both are well-formed. The same is true of $H_t = a_t(H_{t-1})$ and $a_t(H_{t-1}^*)$. Since the edges are the same between these two graphs, their sets of reachable objects are the same, so applying *Live* to each of them gives the same result. Hence $Live(H_t) = Live(a_t(H_{t-1}^*))$. But $Live(H_t) = H_t^*$ by definition, proving the first part of the theorem.

The second part of the theorem follows if

$$Reduce(Live(a_t(H_{t-1}^*))) = Reduce(Live(a_t(Reduce(H_{t-1}^*))))$$

Let L_x and D_x be the L and D sets of H_{t-1}^*. Since *Reduce* does not affect L sets, and program actions do not add to D sets, the L sets of $a_t(H_{t-1}^*)$ and $a_t(Reduce(H_{t-1}^*))$ are the same. Since a_t applies and preserves well-formedness, the L sets of $Live(a_t(H_{t-1}^*))$ and $Live(a_t(Reduce(H_{t-1}^*)))$ are also the same, and consist of exactly those objects reachable in H_t^*. Finally, applying *Reduce* gives heap states with the same L set and an empty D set. Thus the L and D components of the two heap states are the same.

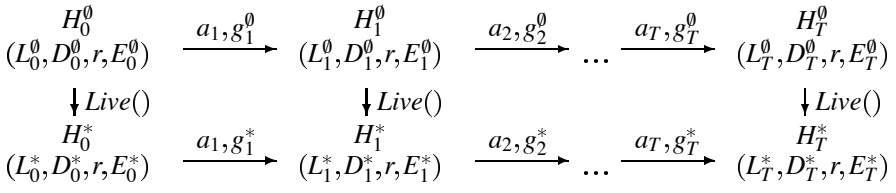

Figure 2. Expanded Program History, including the null ($H^\emptyset$) and reachable (H^*) heap states.

Their r component is trivially the same (none of our functions changes r). Their E component is the same for reachable nodes, but after applying *Live* then *Reduce*, those are the *only* nodes in the graph. Finally, their w component is the same for the reachable nodes (which is all the nodes). ■

2.7. Modeling Collector Behavior

We model garbage collector behavior by following each program action a_t with a garbage collector action g_t. Thus, we form H_t by first applying program action a_t to H_{t-1}, and then applying collector action g_t. A collector action potentially identifies some unreachable objects as dead. In fact, we will equate g_t with the set of objects it identifies as dead, and define its effect on the heap state as mapping heap state $H = (L, D, r, E)$ to $(L - g_t, D \cup g_t, r, E)$, with the precondition that $g_t \subseteq L - \{o \in L | reachable_H(o)\}$. When convenient, we will also use the notation g_t for the function that the collector action induces on heap states.

The simplest collector, which we call the *null collector*, never identifies any objects as unreachable. We write its actions as $g_t^\emptyset$; it induces the identity function on heap states.

The most "aggressive" collector, which we call the *comprehensive collector*, always identifies all unreachable objects. We write it as g_t^* and the function it induces is complementary to *Live* (i.e., it identifies the unreachable objects).

Figure 2 shows how the null ($H^\emptyset$) and reachable (H^*) heap state multi-graphs relate to null and comprehensive collector actions.

Real collectors are bounded (in what they reclaim) by the null and comprehensive collectors. Further, many collectors identify unreachable objects only occasionally. For example, they may allow a portion of the heap to fill, and identify unreachable objects in a batch only after the space is full. While we model only the end result (e.g., the set of objects identified as dead), in applying the framework it is easy to associate costs with collector action g_t and derive the effort taken by the collector at each time step.

2.8. Some Possible Extensions

While the framework as presented forms a base adequate for the purposes of this paper, and perhaps for considerable analysis of garbage collection, we have envisioned some possible extensions that make it even more broadly applicable.

At present we do not model program and collector behavior; that is, we take the a_t and g_t as given, and model their effects on the heap state, but we do not model generating the a_t and g_t themselves. If we add to the heap state multi-graph a *system state* component, which holds all other relevant aspects of program and collector state, then we can model the overall system as a generalized state machine, even allowing it to interact with an environment that supplies input, etc. Formally, we would extend H_t with an additional component, σ_t, giving

$$H_t = (L_t, D_t, r, E_t, \sigma_t)$$

The extended H_t, along with input i_t, would determine the action a_t and the collector action g_t. A desirable property of such a system is that the g_t do not affect the sequence of a_t. To that end, one would want to demonstrate that g_t does not affect any aspect of the state σ that can determine future actions a. Hence, one might want to partition the state into a program state component and a collector state component, etc. We leave the details to future work.

The state machine approach could bridge between algorithmic descriptions of programs, and collectors, and their representation in the model. The state machine approach also has the virtue that the cost of collector actions might be easier to model, in that we would be modeling *how* a collector determines which objects to add to D, not just the end result of that process.

3. Heap Traces

When exploring the performance of a new garbage collector, one can often work faster by using a simulator. For this, one runs a program in a system instrumented to produce a collector-neutral *heap trace*. The simulator accepts the trace as an input and, given the GC algorithm, system parameters, and algorithm tuning parameters (such as the maximum heap size allowed), estimates the work needed by the algorithm for the traced instance of the program.

A heap trace is a time-ordered sequence of records. The records are of these kinds: *object allocation*, giving the new object's size and a unique identifier; *object death*, giving the dying object's unique identifier; *heap reference update*, giving the source object, field key, and target object or null value; and *root reference update*, giving a location of the root, and the target object or null value. (The update records also implicitly define a reference deletion if the field/root previously contained a reference). With a perfectly accurate trace the simulator *could* determine when each object dies, but it is easier to write the simulator, and the simulator runs much faster, if the death times are provided in the trace. This is since a single trace file is used in many different simulations, it is cheaper to compute the death times once in advance.

One way to obtain death times for traces is to perform a comprehensive collection whenever a collection could occur in practice. Since most collectors attempt collection only in response to an allocation (i.e., when they require additional space in the heap), this requires doing a collection just prior to each allocation. This is the *brute force* approach to trace generation. As these constant collections take a substantial amount of time, researchers often use traces generated with less frequent comprehensive collections, resulting in traces that may distort simulator results significantly [11]. We

now extend our framework to model object death times, in preparation for presenting and analyzing the Merlin trace generation algorithm.

3.1. Object Death Time Multi-Graph

We add to our framework the *object death time multi-graph*. This multi-graph differs from the others because it concerns only the efficiency of a collector. It is not related to the heap at any moment of the program history; rather it exists to prove the minimum information and work needed to determine the earliest time each object could be reclaimed. This multi-graph can also compare the efficiency of comprehensive collectors by analyzing the relative work needed to populate and analyze this multi-graph. Before describing the new graph, we discuss a concept upon which it relies: *final reference deletion time*.

An object's final reference deletion time is the last time at which the object has an incoming reference deleted (by a root or heap reference deletion action or at program termination). Because each object is allocated with a reference from the root set, and the program termination action removes any root references that exist, each object has a final reference deletion time. This time occurs between the object's allocation and program termination. We define the function f to map each vertex to its final reference deletion time. Given a vertex v, f is defined as: $f(v) = \max_{i<T}((\exists o,n)((\langle o,n\rangle,v) \in E_i \wedge (\langle o,n\rangle,v) \notin E_{i+1}))$. An object may have incoming references at its final reference deletion time, provided that the remaining incoming references are not deleted by a program action; in the name, "final" modifies "deletion", not "reference".

With this definition, we present the last graph structure of the paper. The object death time multi-graph is also a directed, rooted multi-graph, $F = (V, E_T, f)$, where f is the final reference deletion time function from above. The multi-graph's set of vertices V contains a vertex for each object allocated in the heap, i.e., $V = V_T - \{r\}$. The multi-graph's edge multiset, E_T, is the edge multiset of the final heap state.

As the name implies, the multi-graph determines the *death time* for each object. An object's death time is the time at which it becomes unreachable—the time the corresponding vertex would be included in a g^* action. As the following theorem shows, this time can be computed in the object death time multi-graph as each vertex's latest *reaching final reference deletion time*, the latest final reference deletion time among the vertices that reach each vertex.

Theorem 3 *The latest reaching final reference deletion time to a vertex in F is the time the corresponding object in the heap became unreachable.*

Proof Objects become unreachable only when references are removed; thus object death times occur only at actions that remove references. Since unreachable vertices cannot be involved in program actions, only final reference deletions cause objects to become unreachable: if an earlier reference deletion left an object unreachable, the object could not be involved in the later action! Therefore, object deaths occur only at final reference deletions.

Not all final reference deletions leave a vertex unreachable, however. From the definition of *reachable*, *reachable*(v) is true when v is in the transitive closure of the root vertex, regardless of final reference deletion times. If v is the target of an edge

from a reachable vertex, *reachable*(v) is true. If v's final reference deletion time has passed, v becomes unreachable at the latest time that a referring vertex becomes unreachable. By this logic, each vertex becomes unreachable at the latest final reference deletion time for it *or any vertex that reaches it* (since the vertices reaching it may be reachable only because they are referred to by still other vertices). Thus, any object in the transitive closure set of an object at its final reference deletion time may become unreachable at that time. These transitive closure sets are defined by the final pointers—the object death time multi-graph's edge multiset.

Therefore, vertices become unreachable at the latest reaching final reference deletion time. As reachability in the heap state reflects reachability in the heap, this time is the corresponding object's death time. ■

We note that this multi-graph is similar to H_T^* ($V = D_T^*$ and $E_T = E_T^*$) as one would expect, given its function. Using this multi-graph, we can compute object death times in asymptotically optimal time, as we show in the next section.

4. Merlin Algorithm Analysis

The Merlin trace generation algorithm, proposed by Hertz *et al*., generates traces over 800 times faster than the previous method of trace generation [11]. The Merlin algorithm, shown in Figure 3, achieves its speedup by performing a small amount of work with (some) program actions and can thus delay performing more costly analyses until necessary. Designed using insights gained from this research, the Merlin algorithm computes each object's final reference deletion time in conjunction with program actions and analyzes the object death time multi-graph that it constructs with this information. From this analysis, the Merlin algorithm can easily and quickly determine each object's death time for trace generation.

This section shows that our framework can help one to discover new insights about garbage collection and also prove asymptotic running times for garbage collection algorithms. We begin by proving the asymptotic running time for the Merlin algorithm. We then prove that this running time is optimal for heap trace generation.

4.1. Merlin's Running Time

Brute force trace generation, discussed in Section 3, computes object death times by performing a reachability analysis whenever it wants object death times. Computing which objects are reachable in the heap state is equivalent to solving the single-source/multi-sink reachability problem from the root vertex, which requires $O(L_t^R + E_t^R)$ time. Since this reachability analysis must be repeated throughout the running of the program, brute force trace generation can require up to $\sum_{t=1}^{T} O(L_t^R + E_t^R)$ time!

To generate traces, Merlin must compute when each object becomes unreachable. For this, Merlin builds and analyzes the object death time multi-graph as the program runs. Using the object death time multi-graph, finding when objects become unreachable requires comparing the reaching final reference deletion times for each vertex; this processing seems analogous to computing all of the transitive closures sets, requiring $O(V_T \cdot E_T)$ time [5, p. 766].

```
if (action = create root reference to o) then
   rootReferences[o]++
else if (action = create heap reference) then
   No action needed
else if (action = delete root reference to o) then
   rootReferences[o]--
   finalReferenceDeletionTime[o] ← current time
else if (action = delete heap reference to o) then
   finalReferenceDeletionTime[o] ← current time
else if (action = allocate object o) then
   rootReferences[o] ← 1
   finalReferenceDeletionTime[o] ← 0
else if (action = program termination) then
   for each vertex v do
      if rootReferences[v] ≠ 0 then
         finalReferenceDeletionTime[v] ← current time
   for each vertex v by decreasing final reference deletion time do
      push v onto the stack
   while (the stack is not empty) do
      pop v from the stack
      workingTime ← finalReferenceDeletionTime[v]
      for each non-null w to which v refers do
         if (finalReferenceDeletionTime[w] < workingTime) then
            finalReferenceDeletionTime[w] ← workingTime
            push w onto the stack
```

Figure 3. The Merlin Trace Generation Algorithm

Solving the transitive closures determines when each object can be reclaimed, but requires more work than is needed. Object death times are the *latest* reaching final reference deletion times; to find death times, Merlin requires a single depth-first search from each vertex in reverse order of vertex final reference deletion times.

Lemma 4 *When computing object death times, each vertex and each edge need be processed only once.*

Proof With the depth-first search, Merlin needs to process each vertex only once, because repeat visits to a vertex are computing equal or earlier reaching final reference deletion times. As only the latest time matters, processing these repeat visits will not change any object death times. Assume that by not processing subsequent visits an incorrect death time is computed for a vertex. For this to hold, a repeat visit must compute a later reaching final reference deletion time. But the depth-first search from the vertex with the later final reference deletion time must begin before, not after, the initial object death time was found. This contradiction invalidates our assumption; processing each vertex once correctly computes object death times. ■

With the framework, we can now prove Merlin's asymptotic running time.

Theorem 5 *Merlin requires $O(T)$ time to create the Object Death Time Multi-Graph and then $O(V_T + E_T)$ time to compute the object death times.*

Proof From the algorithm in Figure 3, the Merlin algorithm takes constant time at each program action to build the object death time multi-graph. For T actions, this clearly takes $O(T)$ total time.

Given the object death time multi-graph, the time required to find all object death times is $O(V_T + E_T)$. The Merlin algorithm begins by sorting the vertices by their final reference deletion time. As these times must lie between 1 and T, a radix sort can order the vertices; this sorting taking $O(V_T)$ time. Since each vertex and edge needs to be processed only once, the depth-first searches also take $O(V_T + E_T)$ total time. Therefore, the Merlin algorithm computes object death times in total time $O(V_T + E_T) + O(T) = O(T)$, since $O(V_T + E_T) \leq O(T)$ (there were only T actions, so one cannot have created more than $O(T)$ vertices or edges). ■

This leads to an additional theorem.

Theorem 6 *The Merlin algorithm computes object death times in asymptotically optimal time for trace generation.*

Proof Each program action during a program's execution may define an action that is necessary in order to compute object death times (e.g., object allocations and program termination). Therefore any on-line algorithm to compute object death times requires at least $\Omega(T)$ time. Since the Merlin algorithm completes in this time, its $\Theta(V_T + E_T + T) = \Theta(T)$ time is optimal. To build the object death time multi-graph, the vertices, final pointers and final reference deletion times are needed. To find the vertices, each object allocation action must be handled. Since knowing which references are final pointers and which reference deletions are final is not computable until termination, all reference creations and deletions must be processed. This takes $\Omega(T)$ time.

Consider an object death time multi-graph where no two vertices share a final reference deletion time and an edge exists between two vertices if and only if the source vertex has an earlier final reference deletion time than the target vertex. As the reaching final reference deletion times to the vertices are the consecutive subsequences of final reference deletion times that begin with the earliest time, computing the object death times within this multi-graph requires completely ordering the object death times. This ordering requires as much time as sorting the vertices, which a radix sort accomplishes in $\Omega(V_T)$ time.

Without knowing the target of an edge, we cannot determine if the edge is on the path of the target's latest reaching final reference deletion time. If the edge is on this path, it must be processed; this cannot be determined, however, without examining the edge source and target. Therefore, every algorithm also requires $\Omega(E_T)$ time to compute object death times.

Combining the arguments of the previous three paragraphs, we conclude that $\Omega(T)$ time is required to build the object death multi-graph and $\Omega(V_T + E_T)$ time is needed to compute object death times. As the Merlin algorithm completes in this time, its $\Theta(V_T + E_T) + \Theta(T)$ time is optimal. ■

5. Related Work

This section discusses the two lines of research from which our framework grows. One line of research investigated the space needed to allocate objects dynamically, while the other developed systems and models to analyze garbage collection implementations.

Dynamic Storage Allocation

Trying to arrange objects allocated dynamically into a fixed-size heap is known as the *Dynamic Storage Allocation (DSA)* problem. For an in-depth review of over four decades of research into this problem, see [20]. Typically, the problem is stated as: given the allocation time and size of each object and object death times (g^*), compute the size of the smallest heap that could hold the objects. Requiring the fields of an object to be contiguous and objects to remain in place once allocated makes this simple problem NP-complete [17]. If algorithms must process object allocation and g^* actions on-line, computing the optimal heap size is EXPTIME-complete [18].

For the well-known First-Fit DSA algorithm [13], researchers have used graph theory to prove its worst-case space bound and that First-Fit's space bound is optimal [7, 8, 14]. Studies have also used graph theory to present other DSA algorithms and prove these other algorithms can approximate DSA solutions to within a small constant factor [7, 8].

This prior research was among the first to consider the total asymptotic running time of memory managers and to solve program memory usage problems via graph theory. While the aims and results of our research are quite different, our framework's approach shows the influence of this earlier research.

Analysis of Garbage Collectors

Given the difficulty of allocating space, other studies investigated how to ensure garbage collectors work properly. Using Larch [9], Nettles implemented a copying garbage collector and proved, albeit informally, that the garbage collector was both correct (i.e., wouldn't crash) and complete (i.e., could eventually collect all unreachable objects) [16]. For region-based memory allocators (which are similar to garbage collectors), Crary *et al*. developed the Capability Calculus. This strongly-typed language can prove if region-based memory managers programmed within it can guarantee correctness and completeness [6]. In both studies, the systems could draw conclusions only about algorithms implemented within them. Also unlike our framework, neither system could compare different collectors nor could they compute the asymptotic running time of an algorithm over a program's execution. But because they enable researchers to offer (limited) proofs about collectors, these systems are an important precursor to our work.

Closest to the work presented here, is the development of the lambda calculus λ_{gc} by Morrisett, *et al*. [15]. This research developed a series of rules describing how a garbage collected heap can change and, from these rules, created models of garbage collection that are independent of any specific system implementation. Like the other research, however, λ_{gc} focused on proving algorithms were correct and complete. While both λ_{gc} and our framework create a set of generic rules describing the behavior of a program, the former is ill-suited for determining a collector's total asymptotic running time or comparing different collectors. But, describing garbage collection

and memory management in purely theoretical terms laid the foundation upon which our framework rests.

6. Future Work

Our framework has been designed to be abstract and yet robust. We demonstrate how it can be used to easily prove asymptotic running times, optimal running times, and space bounds for garbage collection. It is our hope that we, and other researchers, use and extend this framework to fully develop a theoretical underpinning for garbage collection. There are, in particular, several areas in which we hope to expand upon the work in this paper.

Collector Efficiency Analysis
In this research we analyzed properties of comprehensive garbage collection. Using this framework, we hope to perform a similar analysis for non-comprehensive collectors and help focus research seeking faster algorithms. One approach for this future work is to find and recast the important features for the collector state multi-graph much as the object death time multi-graph distills the important features of the reachable multi-graph.

Collection Optimization Analysis
Another direction in which we wish to expand this work is determining the value of different optimizations. By using our framework to measure both the asymptotic cost and potential savings for a variety of optimizations, we could better understand what limits garbage collector speed and how best to overcome it.

7. Summary

This paper introduces several structures and analyses for garbage collectors. With the new structures, we prove the running time of the Merlin trace generation algorithm and show that this time is optimal for computing object death times.

Acknowledgments

We would like to thank Steve Blackburn and Darko Stefanovic for providing valuable insights. We would also like to thank J. M. Robson for his initial encouragement into this research.

References

[1] BARRETT, D. A., AND ZORN, B. G. Using lifetime predictors to improve memory allocation performance. In *Proceedings of SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, NM, June 1993), vol. 28(6) of *ACM SIGPLAN Notices*, ACM Press, pp. 187–196.

[2] BLANCHET, B. Escape analysis for object oriented languages. applications to Java. In *Proceedings of SIGPLAN 1999 Conference on Object-Oriented Programming, Languages, & Applications* (Denver, CO, Oct. 1999), vol. 34(10) of *ACM SIGPLAN Notices*, ACM Press, pp. 20–34.

[3] CANNAROZZI, D. J., PLEZBERT, M. P., AND CYTRON, R. K. Contaminated garbage collection. In *Proceedings of SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, Canada, June 2000), vol. 35(5) of *ACM SIGPLAN Notices*, ACM Press, pp. 264–273.

[4] CHILIMBI, T., JONES, R. E., AND ZORN, B. Designing a trace format for heap allocation events. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), vol. 36(1) of *ACM SIGPLAN Notices*, ACM Press, pp. 35–49.

[5] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to algorithms*. MIT Press, Cambridge, MA, 1990.

[6] CRARY, K., WALKER, D., AND MORRISETT, G. Typed memory management in a calculus of capabilities. In *Proceedings of SIGPLAN 1999 Symposium on Principles of Programming Languages* (San Antonio, TX, Jan. 1999), vol. 34(1) of *ACM SIGPLAN Notices*, ACM Press, pp. 262–275.

[7] GERGOV, J. Approximation algorithms for dynamic storage allocations. In *Proceedings of the Fourth European Symposium on Algorithms ESA 1996* (Barcelona, Spain, Sept. 1996), vol. 1136 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 52–61.

[8] GERGOV, J. Algorithms for compile-time memory optimization. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* (New York, NY, Jan. 1999), ACM Press, pp. 907–908.

[9] GUTTAG, J. V., HORNING, J. J., AND WING, J. M. Larch in five easy pieces. Tech. Rep. 5, DEC Systems Research Center, Palo Alto, CA, July 1995.

[10] HAYES, B. Using key object opportunism to collect old objects. In *Proceedings of SIGPLAN 1991 Conference on Object-Oriented Programming, Languages, & Applications* (Phoenix, AZ, Oct. 1991), vol. 26(11) of *ACM SIGPLAN Notices*, ACM Press, pp. 33–40.

[11] HERTZ, M., BLACKBURN, S. M., MOSS, J. E. B., MCKINLEY, K. S., AND STEFANOVIĆ, D. Error-free garbage collection traces: How to cheat and not get caught. To Appear at Sigmetrics 2002. Available at ftp://ftp.cs.umass.edu/pub/osl/papers/sigmetrics-2002-merlin.ps.gz.

[12] HUDSON, R. L., AND MOSS, J. E. B. Incremental garbage collection for mature objects. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France, Sept. 1992), vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 388–403.

[13] KNUTH, D. E. *Fundamental Algorithms*, 2*nd* ed., vol. 1 of *The Art of Computer Programming*. Addison Wesley, Reading, MA, 1973.

[14] LUBY, M. G., NAOR, J. S., AND ORDA, A. Tight bounds for dynamic storage allocation. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms* (Arlington, VA, Jan. 1994), ACM Press, pp. 724–732.

[15] MORRISETT, G., FELLEISEN, M., AND HARPER, R. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture* (La Jolla, CA, June 1995), ACM Press, pp. 66–77.

[16] NETTLES, S. A Larch specification of copying garbage collection. Tech. Rep. CMU-CS-92-219, Carnegie Mellon University, Pittsburgh, PA, Dec. 1992.

[17] ROBSON, J. M. Storage allocation is NP-hard. *Information Processing Letters 11(3)* (Nov. 1980), 119–125.

[18] ROBSON, J. M. Optimal storage allocation decisions require exponential time. Tech. Rep. TR-CS-81-03, Australian National University, 1981.

[19] SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. On the effectiveness of GC in Java. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), vol. 36(1) of *ACM SIGPLAN Notices*, ACM Press, pp. 12–17.

[20] WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management* (Kinross, Scotland, Sept. 1995), vol. 986 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–116.