Using Page Residency to Balance Tradeoffs in Tracing Garbage Collection

Daniel Spoonhower*
spoons@cs.cmu.edu

Guy Blelloch
blelloch@cs.cmu.edu

Robert Harper
rwh@cs.cmu.edu

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

We introduce an extension of mostly copying collection that uses *page residency* to determine when to relocate objects. Our collector promotes in place pages with high residency, avoiding unnecessary work and wasted space. It predicts the residency of each page, but when its predictions prove to be inaccurate, our collector reclaims unoccupied space by using it to satisfy allocation requests.

Using residency allows our collector to dynamically balance the tradeoffs of copying and non-copying collection. Our technique requires less space than a pure copying collector and supports object pinning without otherwise sacrificing the ability to relocate objects.

Unlike other hybrids, our collector does not depend on application-specific configuration and can quickly respond to changing application behavior. Our measurements show that our hybrid performs well under a variety of conditions; it prefers copying collection when there is ample heap space but falls back on non-copying collection when space becomes limited.

General Terms

algorithms, languages, measurement, performance

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*C#*; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

Keywords

predicted residency, fragmentation, compaction, pinning

*This work was supported by a Microsoft Graduate Fellowship and a generous gift from Microsoft.

VEE'05, June 11-12, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

Garbage collectors in modern virtual machines must satisfy the demands of both a high-level language and a native runtime implementation. While the former takes an abstract view of memory, the latter requires a transparent interface where the position and layout of heap objects are made explicit. To interoperate with existing libraries, many languages also provide foreign function interfaces (*e.g.* Haskell [10], Java [27]). Each of these imposes constraints on how garbage can be collected, for example by limiting when objects can be relocated.

In this work, we explore the tradeoffs of tracing garbage collection. We show how the two simplest forms of tracing collection, mark-sweep and semi-space collection, complement each other not only in terms of their benefits and costs, but also in their approaches to reducing fragmentation. We observe that there is a smooth continuum of tracing collectors that spans both mark-sweep and semi-space collection and use this insight to design a garbage collector that combines these two techniques dynamically. We have implemented our collector as part of a virtual machine for C# [19] and measure the overhead of the collector for several benchmarks. Our experiments show that our collector mimics the behavior of either a copying or a non-copying collector depending on which would yield better performance.

Many collectors combine copying and non-copying collection by grafting several smaller heaps together and using a different technique to manage each heap. For example, large objects are often relegated to a separate heap that is managed using a mark-sweep collector [29, 9]. In addition to size, objects in these separate heaps may also be distinguished using profiling information [12, 5].

Some generational collectors [22, 28] also combine copying and non-copying collection by distinguishing objects based on their age and using different techniques for the nursery and the older generations. Generational collectors improve collector performance in two ways. First, they improve latency (on average) by performing partial traversals of the heap during minor collections. Second, they may improve throughput by focusing on portions of the heap that contain few reachable objects: if the (weak) generational hypothesis holds, most objects will die young and the nursery will be relatively sparse.

Bartlett's mostly copying collector [3] combines copying and non-copying collection to support compaction in an environment with ambiguous roots (*i.e.* where the types of

roots are unknown). It divides the heap into a set of pages and uses a different technique for each page depending on the presence of one or more ambiguous roots.

Mostly copying collection forms the foundation for our work. Unlike hybrid collectors where each portion of the heap is statically configured to use either copying or non-copying collection, our collector allows the technique used to manage each page to vary from one collection to the next.

Instead of "syntactic" properties such as size or allocation point, our collector relies on *page residency* to determine which pages should be managed using copying collection and which using non-copying collection. Since residency reflects the runtime behavior of the mutator, we claim that our collector can adapt more easily to shifting memory usage patterns.

Although age has been shown to be a good indicator of the expected lifetime of young objects [18], we count age as only one of many predictors of residency. To take advantage of the relationship between age and expected lifetime, generational collectors must group objects of similar age together. The generational hypothesis implies that the residency of the nursery will be low because the nursery contains only young objects. We present a set of heuristics for predicting residency in Section 4 and a description of our algorithm in Section 5.

The experiments described in Section 6 show that our collector performs well under a variety of conditions using a single configuration. For example, our collector makes effective use of space regardless of the overall residency of the heap.

In this work, we focus primarily on the tradeoffs of tracing collection and on improving collector throughput. We extended our hybrid collector with a form of replicating incremental collection [24] to improve its latency, but defer discussion of incremental collection to Section 8.

The main contributions of this work include:

- the identification of a continuous range of tracing collectors including mark-sweep and semi-space collectors as antipodal strategies,
- the recognition of *measured* and *predicted residency* as the driving properties behind the behavior of tracing collectors (including existing hybrid algorithms), and
- an implementation that measures and predicts residency to dynamically balance the tradeoffs of copying and non-copying collection.

To give a better sense of how we arrived at our current collector design, we begin with a historic development of our ideas and an explanation of the work on which we build.

2. MOTIVATION AND BACKGROUND

The original goal of the current work was to extend the real-time collector of Blelloch and Cheng [6] with support for pinned objects. Blelloch and Cheng showed that replicating collection [24] can support concurrent and parallel collection with low latency. However, no pure copying collector can support pinned objects: unlike most objects in the heap, each pinned object must remain at a fixed location for some extended, but unspecified period of time. Pinned objects are required by many virtual machines and are essential in a language with a foreign function interface (*e.g.* Haskell [10], Java [27]) or in any programming environment that offers a transparent view of memory (*e.g.* languages such as C# [19] that support address arithmetic). Though our implementation uses replicating collection to give small upper bounds on pause times, our ideas apply to other forms of incremental collection as well.

Our collector supports pinning on a "pay-as-you-go" basis: its consumption of time and space are bounded only so long as objects are not frequently pinned. We are willing to relax our constraints on resource consumption when many objects are pinned, as pervasive use of pinning can defeat any efforts of the collector to manage the heap effectively.

Balancing Tradeoffs.

Support for pinned objects and the ability to improve locality are examples of features that drive the design of hybrid tracing collectors. The remaining tradeoffs of pure copying and non-copying collectors are reviewed briefly below. We provide only a cursory description as a reminder of the numerous complementary aspects of these two forms of tracing collection. (Jones and Lins provide a more thorough survey and comparison of these techniques [20].)

Copying collectors [17] eagerly reduce fragmentation by reorganizing the heap during each collection. This reorganization affords several advantages, including improved locality of reference and faster allocation, but comes at a cost: copying objects requires additional time and space, and potentially dirties more pages in the cache. In contrast, non-copying collectors [23] only attempt to reduce fragmentation by using unoccupied space to satisfy allocation requests. They do not offer the benefits of reorganizing the heap, but neither do they pay the penalties. They perform well even if the heap is densely populated, and they support a broader set of language semantics, including "conservative" collection [7] and object pinning.

To build a collector that aggressively defragments the heap and provides good asymptotic bounds on running time, but naturally supports pinned objects, we look to type-inaccurate or "conservative" collection for inspiration.

Mostly Copying Collection.

Bartlett proposed *mostly copying collection* [3] as a mechanism to mediate some of these tradeoffs: his collector supports a limited form of compaction in the presence of ambiguous roots. Bartlett's collector divides the heap into a set of pages,[1] rather than two semi-spaces. The distinction between the *from-space* and *to-space* is made logically rather than by fixed ranges of addresses. Seen abstractly as sets of pages, the from- and to-spaces are not necessarily contiguous regions of memory, and pages may move fluidly from one set to the other.

During collection, reachable objects are "moved" into to-space. An individual object can be moved either by evacuating it to a to-space page and updating any references to it (as in semi-space collection), or by promoting the entire page on which it resides. In the latter case, a page is promoted by simply removing it from the set of from-space pages and adding it to the set of to-space pages. Roots whose types are unknown are promoted in place, and therefore, those that are misclassified (as pointers) will not be

[1] In both Bartlett's implementation and ours the size of a heap page may be chosen independently of size of pages managed by the operating system.

changed erroneously during collection.

Though our collector maintains accurate type information about the call stack, certain roots are designated by the programmer as *pinned roots*. This imposes a constraint similar to that addressed by Bartlett: the collector must preserve the address of any object that is the referent of one or more pinned roots.

Large Objects. Simultaneously supporting ambiguous roots and heap compaction is only one way that mostly copying collection can balance the tradeoffs of copying and non-copying collection: there are other opportunities to use in-place promotion. In particular, it is well-recognized [26] that there is little or no benefit in copying objects that consume all of the space associated with a page. These *large objects* are expensive to copy, both in terms of the time required to do so and the additional space required for the duplicate. Instead, large objects can be promoted in place just as if they are pinned. Thus our collector treats large objects using the same mechanisms already employed for pinned objects. Objects larger than a single page are allocated using a set of contiguous pages. Though it may be necessary to relocate existing large objects to satisfy these requests, we speculate that nearly all requests can be satisfied without moving large objects.

Support for pinned and large objects are both features usually associated with non-copying collectors, yet mostly copying collection allows us to integrate them cleanly into a copying collector. In both cases the "down payment" is low: there is little impact on performance if the mutator does not use pinning or allocate large objects.

3. PAGE RESIDENCY

Large objects provide an illustrative example of the differences between copying and non-copying collection. We claim that collectors should avoid copying large objects not only because they are expensive to copy, but because there is little or nothing to be gained in doing so. Pages containing large objects make exemplary candidates for in-place promotion because the collector can determine *a priori* that there is little risk of fragmentation. Ideally, the collector would promote only those pages where the cost of copying reachable objects outweighs the otherwise resulting fragmentation.

Previous implementations of mostly copying collection [3, 26] promoted pages only in the presence of ambiguous roots or large objects. We extend this work and use *page residency*, the density of reachable objects on a page, to determine when to promote or evacuate the objects on a given page. Using this concept, we say that if the residency of a page is sufficiently high, the page should be promoted in place. Seen in this light, large objects should remain in fixed positions in the heap, not merely because they are large, but because they occupy densely populated pages.

A Continuum of Tracing Collectors. Using residency to determine which pages should be evacuated, and which should be promoted, gives rise to a range of collector configurations determined by an *evacuation threshold*. In each configuration, a page will be evacuated if its residency is less than or equal to this threshold. At one extreme of this range is a semi-space collector, which evacuates all pages regardless of their residency.

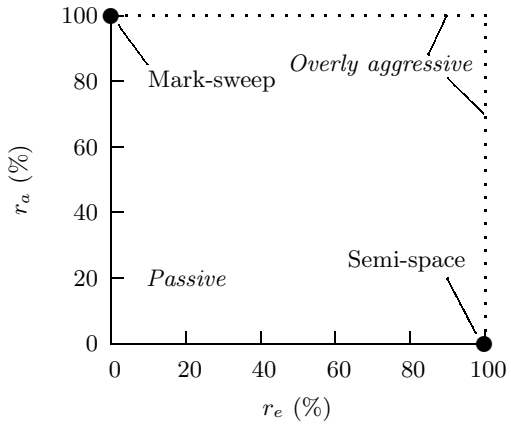

Figure 1: A Continuum of Tracing Collectors. Each point represents a different tracing collector. Both the x- and y-axes range over residency thresholds. The x-axis ranges over the maximum residency at which a page will be evacuated, and the y-axis over the maximum residency for a page that will be considered during allocation.

To allow our collector to reclaim space during the allocation phase, we identify a second, orthogonal range of configurations defined by an *allocation threshold*. The unoccupied space on pages whose residency is less than or equal to this threshold will be used to satisfy mutator allocation requests. At one extreme lies a mark-sweep collector, which uses all free space in the heap to satisfy allocation requests.

Using both thresholds yields a continuum of tracing collectors that includes both semi-space and mark-sweep collectors as depicted in Figure 1. The horizontal axis gives possible values for the evacuation threshold r_e, while the vertical axis gives possible values for the allocation threshold r_a. Both axes range from 0 to 100%. If the page residency r is less than or equal to r_e then objects on the page will evacuated during collection; if r is less than or equal to r_a then free gaps on the page will be considered by the allocator in satisfying requests for more memory.

An evacuation threshold of 100% indicates that objects will always be evacuated during collection (since page residency is always less than or equal to 100%). This corresponds to the behavior of a semi-space collector. On the other hand, an evacuation threshold of 0% says that (because no objects will be found on a page with 0% residency) all objects will be promoted in place.

An allocation threshold of 0% indicates that the free portions of any page containing reachable data will never be considered for reuse until all reachable data has been evacuated from the page (i.e. the page contains exactly one "gap"). Again, this corresponds to the behavior of a semi-space collector. At the opposite extreme, an allocation threshold of 100% says that any unused space can be used for allocating new objects.

Limiting Fragmentation. These two thresholds determine how and when the collector reduces fragmentation in the

heap. For example, semi-space collectors eliminate all fragmentation during the collection phase. Mark-sweep collectors discover fragmentation during collection, but only reduce it during allocation by using new objects to fill in the gaps between old ones. In this case, another collection is initiated only when the remaining gaps are too small to satisfy the current allocation request. Thus reducing fragmentation during collection is a property of copying collectors, while reducing fragmentation during allocation is a property of non-copying collectors.

Configurations in the lower-left part of the graph (where both r_e and r_a are small) permit significant fragmentation to persist through both the collection and allocation phases. These "passive" collectors make poor use of space by neither reorganizing the heap during collection nor filling in unused gaps during allocation. Large parts of the heap will remain unavailable to the mutator, and reachable objects will become strewn throughout the heap, resulting in poor cache performance.

The "overly aggressive" configurations along the upper and right-hand edges also suffer from a poor use of resources. Just as permitting fragmentation has an overhead (*e.g.* larger memory footprint, poor locality), reducing it also incurs costs. Reducing fragmentation requires either additional space to hold duplicates (as in a semi-space collector) or additional passes over the heap to rearrange live objects (as in a mark-compact collector).

We find that admitting a small amount of fragmentation improves the performance of our collector. Our hybrid collector uses page residency to quantify fragmentation and to reduce (but not eliminate) fragmentation during both the collection and allocation phases. To do so, it must determine the residency of each page precisely and efficiently.

4. PREDICTING RESIDENCY

Measuring the residency of even a single page requires a traversal of the entire heap. Our collector measures the residency of each page as part of the normal traversal that occurs during collection. Then it uses this information to determine the set of pages that will be used to satisfy allocation requests. However, the set of pages that will be promoted in place should reflect the residency of each page at the *beginning* of a collection, before the heap is traversed.

Since our goal is to use residency to improve the performance of our collector, we consider an additional traversal at the beginning of each collection to be prohibitively expensive. Instead of using the actual residency to determine which pages will be promoted in place, our collector uses the *predicted* residency of each page.

We describe several heuristics for predicting page residency. Our collector relies on past measurements of the residency to form its predictions, but we also identify two additional predictors of residency. In the first case, residency predictions are based on static configuration parameters: certain pages are permanently designated as densely populated and others as sparsely populated. In the second case, the age of objects residing on a page is used to predict the page residency.

4.1 Historical Measurements

Our collector uses residency measurements taken during one collection to predict the residency of each page at the beginning of the next collection. Residency is measured during the tracing phase; the residency of a page is updated each time an object on that page is shaded.

Our implementation assumes that the residency of each page will remain unchanged from one collection to the next. However, this assumption is not essential to our algorithm. More sophisticated heuristics for predicting residency might account for the rate at which objects become unreachable, for example, by using decay models to compute residency as a function of time. Furthermore, while our collector currently uses only the most recently measured residency, it could maintain a series of historical measurements for pages that are promoted in place during several consecutive collections.

For each page whose predicted residency is less than or equal to the evacuation threshold, space must be reserved to hold replicas in the next collection. Thus the end of one collection is a natural time to determine which pages will be promoted in the next collection, since that is when it must also determine how much space can be allotted to the mutator and how much must be reserved for evacuated objects.

Our collector has no historical measurements for those pages that have not yet survived a collection. One design choice is to assume a small, fixed residency for all young pages. Instead, our collector records the residencies of young pages during the collection phase and uses these measurements to predict the residency of young pages in the future. This is similar to a technique used in garbage-first collection [15].

This heuristic still assumes that the residency of young pages will be uniform. An alternative heuristic would measure age not in terms of the number of collections survived but in terms of bytes subsequently allocated. Following the intuition of older-first garbage collection [13] our collector could predict that the residency of pages allocated just before a collection will be higher than those allocated earlier, possibly promoting the former in place (depending, of course, on the evacuation threshold).

Note, however, that the predicted residency of young pages is used only to determine the promotion strategy and not the amount of space that must be reserved for replicas. In contrast, the measured residency of an older page forms an upper bound on the amount of data that may be evacuated in the next cycle (unless free space on the page is used to satisfy new allocation requests). Our collector conservatively estimates the amount of space that must be reserved for replicas by considering the maximum amount of data that may be evacuated from each page, regardless of the predicted residency.

4.2 Static Configuration

Instead of using historical measurements, the residency of each page can be predicted using configuration parameters set by the application programmer or the end user. If the behavior of the mutator is known in advance, the collector can be configured to anticipate that behavior.

This configuration effectively determines the predicted residency of each page on a rough scale. Existing hybrid collectors then use a heuristic to determine whether a given object should be allocated on a high residency page or a low residency page. Since predictions about page residency remain fixed, the performance of the collector depends on the accuracy with which an appropriate page is chosen by the

Page status	Contents during allocation phase	Contents during collection phase	Role in semi-space collection	Colors during collection
EVACUATED	new and surviving objects	original copies and unreachable objects	from-space	white
PROMOTED	new and surviving objects	objects promoted in place, replica copies, and unreachable objects	from- and to-space	gray (if in queue), black (if marked), or white (otherwise)
REPLICA	no objects	replica copies	to-space	gray (if in queue) or black
FREE	no objects	no objects	-	-

Table 1: Description of Page Status.

allocator.

For example, a large object heap managed using a mark-sweep collector is a set of pages for which the residency is predicted to be high. Thus pages in the large object heap will always be promoted in place. Recall that rather than segregating large objects into a separate set of pages, we observe that pages containing large objects should be promoted in place wherever they reside. This avoids the need to set aside heap space for large objects. Integrating large and small objects on a single set of pages also allows the collector to use small objects to fill the gaps between large ones.

Other properties derived from the point of allocation can also be used in attempts to make static residency predictions come true. Unfortunately, none of these properties can account for changes in the behavior of the mutator over time. While these "syntactic" properties may be good predictors of residency for pages containing young objects, runtime information about residency offers better predictions in the long run.[2]

4.3 Age

Generational collectors assume that most objects will die young. To take advantage of this assumption, they group young objects together so that the residency of the nursery will be low. It follows as a corollary that objects in the nursery should be evacuated.

However, generational collectors often defer to other predictors of residency in cases where age is not strongly correlated with expected lifetime. For example, many generational collectors pretenure larger objects [29, 9] in part because allocating large reachable objects in the nursery would violate the assumption that the residency of the nursery is low. The same is true for other forms of pretenuring based on profiling data [12, 5]. Since they assume that the nursery is sparsely populated, these generational collectors take steps to ensure that no long-lived objects reside there. Thus young objects are not treated uniformly. These collectors use age as a guide only when it is assured to be a good predictor of residency.

While it may be reasonable to assume that many young objects are unreachable, it is not safe to assume that most old objects are still live. As objects become older, their age offers less information about their expected lifetimes [18]. In particular, age does not give any indication of when old objects should be "demoted."

[2]One exception to this claim may be the case of *immortal objects*, those objects that are statically known to survive forever.

4.4 Poor Predictions

If residency predictions are accurate then only densely populated pages will be promoted in place. If predictions are inaccurate, however, then promoted pages may contain few reachable objects, leading to potential fragmentation. Our collector gracefully recovers from this condition using the mechanisms described above.

Any collection that uses inaccurate predictions to promote pages will also measure the true residency of each page while scanning the heap. Sparsely populated pages will then be evacuated during the subsequent collection. In the extreme case, poor residency predictions will prevent the recovery of any usable space, and collector will be invoked again immediately. In this case, the first collection serves to correct the inaccurate predictions, while the second collection uses the new measurements to reorganize the heap. In this way, our collector only takes the time to measure page residency using a second traversal in cases where its predictions would be inaccurate.

In addition, since the true residency of each page is known at the end of each collection, unused space can immediately be reused during the allocation phase to satisfy mutator requests.

5. A DYNAMIC HYBRID ALGORITHM

Our hybrid algorithm implements the full range of configurations shown in Figure 1. It is independent of the heuristic used to predict residency and parameterized by the evacuation and allocation thresholds.

The behavior of a tracing collector is largely determined by the actions it takes when traversing objects and by the way it satisfies allocation requests. All tracing collectors must distinguish between white, gray, and black objects,[3] but each one may maintain this distinction using a different data structure. Our hybrid collector takes advantage of the common structures found in both copying and non-copying collectors when possible but occasionally maintains separate structures for the sake of efficiency. Despite this, our algorithm admits a simple description that is faithful to our implementation.

As in Bartlett's presentation of mostly copying collection, our collector first divides the heap into pages.[4] All pages are initially designated as *free pages*. We describe three other possibilities for the status of each page in Table 1. First, *evacuated pages* form part of the from-space, and any reachable objects on an evacuated page will be copied during col-

[3]We use the standard tri-color abstraction [16].

[4]Our implementation uses pages of 4 kB.

lection. All evacuated pages will be reclaimed at the end of collection. Second, *promoted pages* contain objects that will be promoted in place. These pages may contain white, gray, and black objects. Because there are no replicas associated with objects that are promoted in place, promoted pages are part of both the from- and to-spaces. Finally, *replica pages* contain replicas created during collection (with one exception described below). They form the remainder of the to-space.

The *page table* stores both the status and the residency of each page. Our collector maintains the page table separately from the data associated with each page. To allow the set of pages of any status to be enumerated and manipulated efficiently, they are stored in doubly-linked lists, one for each possible status. This list structure is also stored in the page table. Each page table entry consumes 16 bytes in our implementation.

5.1 Collection

Collection is initiated when the remaining amount of free space in the heap is equal to the total size of all objects that may be evacuated during collection. For example, a mark-sweep collector begins a new collection when no free space remains, since no objects will be evacuated. For a semi-space collector, collection begins when half of the heap is occupied, since all objects may be evacuated (if they are reachable).

Initial Phase. The collector begins by enumerating the root set. To avoid prematurely evacuating a pinned object, it must consider all pinned roots before performing any evacuation. Pages containing the referents of pinned roots are promoted in place, and these referents are shaded. Then the collector enumerates all remaining roots and shades their referents. The collection technique for each remaining page follows from the page status and was determined during the preceding collection or allocation phase. Because pinned roots never increase the number of evacuated objects, the collector is still guaranteed to have sufficient space to complete the current collection.

Tracing Phase. Once the collection technique for each page has been fixed, the collector traces the memory graph by advancing the gray frontier. When a shaded object resides on a promoted page, it is promoted in place; shaded objects on evacuated pages are (as the name suggests) evacuated.

To maintain the set of gray objects efficiently, our hybrid collector uses two different data structures. The *mark queue* holds references to gray objects that reside on promoted pages. A *mark bit* associated with each object on a promoted page is set to ensure that each object is added to the mark queue exactly once (and that cycles are handled correctly). Each time a mark bit is set, the page residency is updated to reflect the size of the shaded object.

It is possible to use a mark queue to store the set of evacuated gray objects. However, this set can be managed more efficiently in a semi-space collector using a pair of addresses, since these evacuated objects will reside in a contiguous portion of the heap [11]. Unlike a semi-space collector, not all evacuated pages may be adjacent to each other in our hybrid collector. The *Cheney queue* holds pairs of addresses that delimit regions of gray replica objects. When a pair of addresses is removed from the queue, every object residing between the pair of addresses is traced. Evacuated objects are assigned forwarding pointers to ensure that each object is copied at most once. In these aspects of our implementation, we follow the work Smith and Morrisett [26].

Replica copies may reside on either replica pages or the unoccupied parts of promoted pages. In the latter case, replicas should be marked, regardless of whether they are added to the mark queue or the Cheney queue, so that they will be preserved through the sweep phase.

In summary, an object is gray if it appears in either the mark queue or between a pair of addresses in the Cheney queue. Black objects are those that do not appear in either queue but reside on promoted pages and are marked, or simply reside on replica pages. All remaining objects are white.

Sweep Phase. When no gray objects remain, all objects on replica pages are black by definition. Evacuated pages contain only white objects. Promoted pages, however, will contain both black and white objects. Each of these pages must be swept, clearing the mark bit from each black object. The collector also performs an additional task during the sweep phase: for pages whose residency is known to be low, it coalesces adjacent white objects to form free gaps. These gaps are threaded together to form the *gap list* and will be reused during allocation or the next collection.

Final Phase. At the end of each collection, the collector must flip the roles of the from- and to-spaces. All evacuated pages may be recycled in bulk and are now considered free pages. For each promoted and replica page, the collector considers the residency measured during collection and uses this value to predict the residency at the beginning of the next collection. Pages whose predicted residency is less than or equal to the evacuation threshold are designated as evacuated pages. The rest will be promoted.

One final point must be considered in determining the collection technique of each page. An evacuated page cannot be used to hold replicas while it is being evacuated. Thus we have one final criterion for in-place promotion: pages with free gaps that will be used to hold replicas in the next collection should also be promoted in place.

For each page that will be evacuated in the next collection, the collector must reserve adequate space to hold replicas created during evacuation. For this purpose, the collector computes the total of size of all objects to be evacuated by computing the sum of the measured residencies of evacuated pages.

The free space available to hold future allocations and replicas is given by the total size of all free pages, plus any space available in the gap list. From this, the collector subtracts the total size of objects to be evacuated. The remaining quantity is the total *unreserved space*. This quantity will be used during allocation to trigger the next collection.

5.2 Allocation

Allocation requests may be satisfied in two ways. Free pages can be used to satisfy requests using a pointer-bumping technique (as in a semi-space collector). The allocator maintains a pair of addresses, the *allocation pointer* and the *allocation limit*, to delimit the block of memory that will be considered in fulfilling the next request.

Allocation requests can also be satisfied using the free

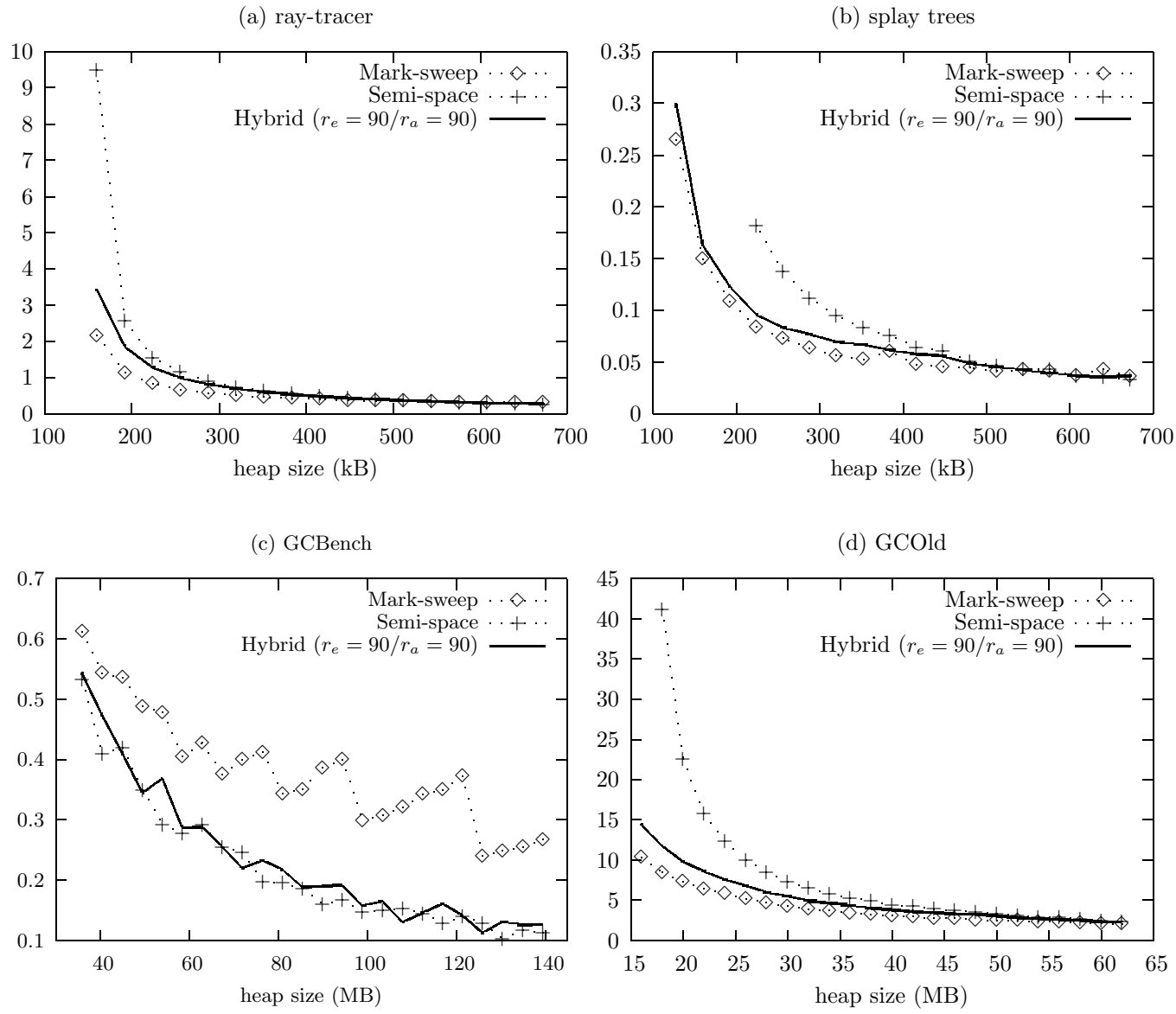

Figure 2: Collection Overhead vs. Heap Size. The vertical axis shows the total collection time measured in seconds. The same hybrid configuration is used in all cases ($r_e = 90\%$ and $r_a = 90\%$), but the performance of the hybrid collector always approximates the better of other two collectors. The jagged curve of mark-sweep collector in the GCBench application is caused by the significant amount of garbage and strong correlation among object lifetimes in that application.

gaps identified during collection. Our implementation uses a first-fit approach to allocate space from the gap list. Once the first sufficiently large gap is found and removed from the list, the allocation pointer and allocation limit are set to delimit that gap. The request is fulfilled and allocation continues until the gap is exhausted. While other allocation techniques are compatible with our algorithm, this technique allows us to use the same inlined allocation code for both free pages and the gap list. Our collector prefers to use the gap list to satisfy requests and uses free pages only if the gap list is empty.

As each block of free space is designated for allocation, either from a free page or a free gap, the collector considers whether or not that block will be evacuated in the next collection (using the predicted residency of the page). If it will be promoted in place then the remaining unreserved space is decreased by the size of the block; if the page will be evacuated, then it must be decreased by twice the size of the block. When the remaining amount of unreserved space reaches zero, a new collection is initiated. In either case, if the block resides on a page that was removed from the set of free pages, then the page is added to either the set of promoted pages or the set of evacuated pages, as appropriate.

Finally, large objects are always allocated by choosing a set of adjacent free pages with enough space to satisfy the request, and pages containing large objects will always be promoted during collection.

6. EMPIRICAL RESULTS

We now describe a series of experiments to demonstrate that our hybrid collector provides consistent performance across a range of applications and configurations. In practice, we expect that predicted residency would be used in combination with a form of generational or incremental collection (as we will discuss along with other related and future work below). However, to better understand our technique, we consider it in isolation. Because our collector makes decisions based on the layout of objects in the heap, a single configuration performs well for several different applications and heap sizes. We also show that the performance of the collector is relatively insensitive to changes in the allocation and evacuation thresholds.

We have implemented our collector as part of the Shared Source Common Language Infrastructure (Rotor), a compiler and runtime implementation for C# [19] that requires support for object pinning. We perform our experiments on an Intel 1.8 GHz Pentium 4 with one GB of physical memory running Microsoft Windows XP Professional. For time measurements, we report the average of two runs.

We use four applications in our experiments; all are memory intensive. The first benchmark, ported from the Java Grande benchmark suite, renders a three-dimensional scene containing 64 spheres. While the original benchmark supports multi-threaded execution, we use a single-threaded version for our tests. Our measurements show the results running with size "A" ($N = 150$).

The second application is an artificial benchmark based on truncated splay trees. This application provides a realistic, yet difficult, case for our collector. Random nodes are inserted repeatedly into a tree, and after each insertion, the tree is rebalanced using splay operations and truncated at a fixed depth. The resulting distribution of object lifetimes follows a log-normal curve: most objects die young, but some live to an intermediate age, and a few live for a long period of time. More importantly for our collector, long-lived objects are distributed throughout the heap and potentially introduce fragmentation. Nodes also vary in size. Trees are truncated at a depth of 30 in our experiments.

The third and fourth applications, GCBench and GCOld, are both originally Java benchmarks and have been ported to C# by the authors. Hans Böhm developed the Java version of GCBench, and David Detlefs wrote the original version of GCOld. Both of these benchmarks build large tree structures, though each has a distinct object lifetime distribution. For GCOld, we use the following parameter values: live data size is 8, work is 1, short-to-long ratio is 32, pointer mutation rate is 2, and steps is 1000. For an explanation of these parameters, we refer the reader to the original author's web site [14]. The C# versions of all four benchmarks are available on the first author's web site.[5]

6.1 Heap Size

Figure 2 shows the total time consumed during collection as a function of the heap size. Heap size is the total space available to the collector, regardless of how that space is presented to the mutator.

As expected, the mark-sweep collector yields lower overheads for small heaps. The semi-space copying collector requires more space and cannot satisfy allocation requests as the size of the heap approaches twice the amount of live data. However, when there is sufficient space and a significant amount of garbage as in the GCBench application, the copying collector yields better performance. (We also expect the mutator to achieve better locality of reference in the presence of the copying collector, though we do not measure it here.) While the semi-space collector performs more collections, each collection is much shorter. The jagged shape of the mark-sweep plot for GCBench is caused by the significant amount of garbage and the fact that objects become unreachable in large groups. As the heap becomes larger, more time is required for the sweep phase; when there is enough space that fewer collections are required, there is a sudden drop in collector overhead.

In these measurements, the hybrid collector evacuates data from pages whose predicted residency is less than or equal to 90% ($r_e = 90\%$). Pages with a measured residency less than or equal to than 90% are used for allocation ($r_a = 90\%$). For brevity, we refer to this configuration as the "90/90 hybrid."

By optimistically promoting dense pages in place, the hybrid collector achieves lower overheads than the semi-space collector for smaller heaps and satisfies allocation requests that the pure copying collector cannot. In applications such as GCBench where there are few dense pages, however, it more closely follows the performance of the semi-space collector.

6.2 Configuration Thresholds

Our algorithm requires two new configuration parameters: the evacuation and allocation thresholds. One might expect this to place a greater burden on the application programmer to tune the collector to match the behavior of the mutator. However, our hybrid collector is relatively insensitive to changes in the configuration shown in the previous experiment.

[5] `http://www.cs.cmu.edu/~spoons/gc/benchmarks.html`

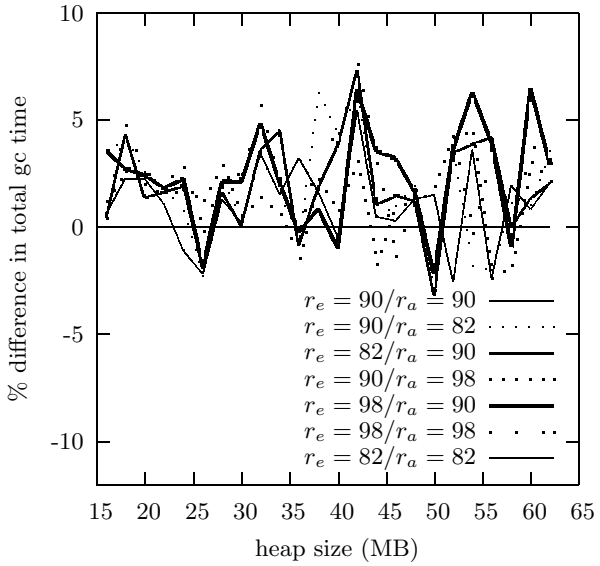

Figure 3: Overhead for Different Thresholds. The difference in overhead of each configuration is given as a percentage of the overhead of the $r_e = 90/r_a = 90$ hybrid for the GCOld benchmark. In each case, the margin is small compared to the performance differences shown in Figure 2.

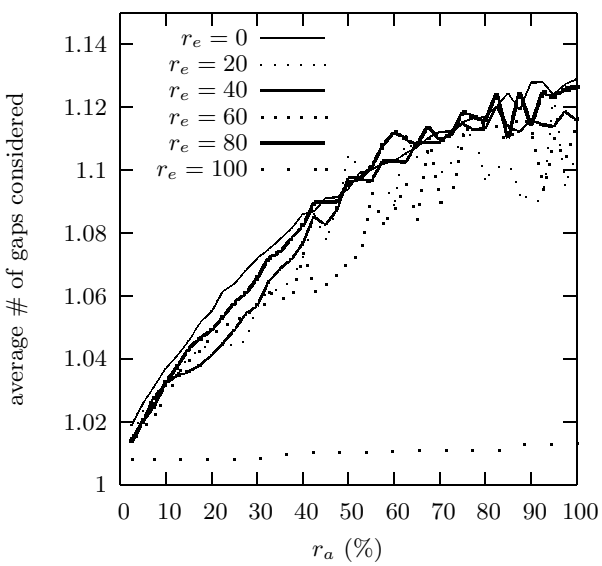

Figure 4: Cost of Allocation. The vertical axis shows the average number of gap list entries that must be considered to fulfill a single request as a function of the allocation threshold r_a. The number of gaps considered is largely independent of the evacuation threshold r_e (except when $r_e = 0\%$). This experiment uses the splay tree benchmark.

Figure 3 shows the difference in overhead for six other configurations running the GCOld benchmark as a percentage of the overhead of the 90/90 hybrid that appeared in Figure 2. We use the same heap sizes as in the previous experiment. Though there is some variation in performance among the different configurations, the margin is less than 8% of the overhead. This is relatively small compared to the differences in performance between the 90/90 hybrid and the semi-space collector (almost 250% for small heaps).

6.3 Cost of Allocation

While our collector does not need to be tuned precisely to the behavior of each application, the evacuation and allocation thresholds can be used to meet specific performance requirements expressed by the application. For example, the allocation threshold determines the speed with which allocation requests can be satisfied. If the allocation threshold is zero, then only empty pages will be used to fulfill allocation requests, and the cost of allocation will be low. If the allocation threshold is high, then the collector will add unused gaps on densely populated pages to the gap list, and the allocator may need to consider many gaps before it finds one large enough to satisfy the current request. In some cases, no gap will be large enough, and a collection must be performed.

Figure 4 shows the average number of entries in the gap list that must be considered to fulfill a single request while running the splay tree benchmark with a heap of 256 kB. As the allocation threshold increases, so does the number of gaps that must be considered by the allocator. For this application, at most 1.13 gaps must be considered, on average, to satisfy a given allocation request. In this benchmark, the evacuation threshold has no significant impact on the time required to satisfy an allocation request, except when it is 100% and all fragmentation is eliminated during collection.

7. RELATED WORK

Bartlett's mostly copying collector [3, 4] supports ambiguous roots and reduces fragmentation by evacuating objects that are not referenced by registers or the cells in call stack. Rather than marking objects on promoted pages, Bartlett's collector preserves all objects on a promoted page whether or not they are reachable. This avoids the extra space required to store mark bits, but results in the retention of unreachable objects.

Smith and Morrisett [26] extend Bartlett's algorithm to support arbitrary "conservative" objects, not just roots. They improve performance by marking reachable objects on promoted pages and thus reducing false retention. Despite this, Smith and Morrisett's collector does not allocate new objects using the space previously occupied by unmarked objects, nor does it measure residency to improve performance.

Mark-copy collection [25] addresses some of the performance tradeoffs of copying and non-copying collection. Sachindran and Moss extend copying generational collection by dividing the older generation into a set of ordered windows and introducing a sweep phase to record each pointer that crosses from a "greater" window to a "lesser" window. All objects are evacuated in mark-copy collection, but since the lowest occupied window may be collected independently of the remaining windows, the space in this window may be reclaimed during collection and used while evacuating later windows. Their collector also measures the residency of each

window during the sweep phase to determine the largest set of windows that may be evacuated simultaneously.

Lang and Dupont [21] describe several related hybrid collectors. In one case, the heap is divided into pages. During each collection, one page is reserved as the to-space, one is designated as the from-space, and reachable objects residing on the from-space page are evacuated to the to-space. The remaining pages are promoted in place. The choice of from-space rotates with each collection, ensuring that the entire heap will be compacted evenutally. In another case, the size of the from- and to-spaces may vary, but each must still occupy a contiguous portion of memory. They speculate that the amount of fragmentation may also be used to guide the choice of the from-space, but do not elaborate on how fragmentation is to be determined. Allocation is performed using a free list, and all unused gaps (outside of the to-space) are used to satisfy mutator requests, regardless of the residency of portion containing each gap.

The garbage-first collector, presented by Detlefs *et al.* [15], is a soft-real time collector that uses residency (among other factors) to estimate the cost of evacuating parts of the heap, called regions. Their collector chooses a set of regions to be evacuated based on these estimates with the goal of achieving predictable pause times. Instead of promoting some regions in place, it simply avoids evacuating objects from them. Its predictions of residency are based on the amount of reachable data discovered during a separate marking process as well as the survival rates of objects in other regions. Their collector uses only empty regions for allocation.

Our work is most closely related to recent work by Bacon, Cheng, and Rajan [2, 1] in which they describe a real-time mostly *non*-copying collector and provide a detailed analysis of its usage of time and space. Because we initially focused on keeping allocation costs low, copying collection was a natural starting point. On the other hand, Bacon *et al.* were concerned with space overhead and thus began with a non-copying collector. This leads to a different set of design choices; for example, they use segregated free lists instead of a pointer-bumping allocator. However, their collector also uses residency to determine which objects should be relocated and to balance the tradeoff between fragmentation and space overhead. Instead of predicting page residency, their collector measures it during an additional sweep phase.

8. EXTENSIONS AND FUTURE WORK

In this presentation, we have focused on collector throughput. Though we do not report on it here, we use replicating collection [24] to support small upper bounds on pause times and consistent mutator utilization.

For the most part, we find that mostly copying collection does not impinge on the implementation of a replicating collector. However, an unbounded number of promoted pages could result in an unbounded stall of mutator execution. Our collector uses a from-space invariant, restricting the mutator to manipulate only those objects that reside in from-space. Because promoted pages are part of both the to- and from-spaces, pointers from objects on promoted pages to objects on evacuated pages cannot be updated incrementally. Instead, they must be flipped atomically at the end of a complete collection.

We accept larger latencies when many objects are pinned, but limit the number of pages that are promoted because of high predicted residency, achieving better latency at the cost of throughput. Another possible solution is to use a more flexible invariant, for example, a Brooks-style invariant [8] as is used by Bacon *et al.* [1]. We have not yet investigated this alternative and continue to study the interactions of mostly copying and incremental collection. We also plan to extend our algorithm to support concurrent and parallel collection.

9. CONCLUSIONS

We have presented a tracing garbage collector that uniformly integrates the features of both copying and non-copying collection. Our collector uses residency to dynamically balance the tradeoffs between these two types of tracing collection. Instead of consuming additional resources to measure residency, it uses predicted residency to make decisions about whether to promote pages or to evacuate them. To recover from inaccurate predictions, it also uses sparsely populated pages to fulfill allocation requests.

We have shown that many existing hybrid algorithms can be viewed in the light of their assumptions about residency. We hope that some insight can be drawn from these hybrids to improve the residency predictions of our collector.

Acknowledgments

We thank the anonymous referees for their helpful and detailed comments.

10. REFERENCES

[1] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 81–92. ACM Press, 2003.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–298. ACM Press, 2003.

[3] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, February 1988.

[4] J. F. Bartlett. A generational compacting garbage collector for C++. In *ECOOP/OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, Canada, 1990.

[5] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 342–352. ACM Press, 2001.

[6] G. E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 104–117. ACM Press, 1999.

[7] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[8] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262. ACM Press, 1984.

[9] P. J. Caudill and A. Wirfs-Brock. A third generation smalltalk-80 implementation. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 119–130. ACM Press, 1986.

[10] M. Chakravarty et al. The Haskell 98 Foreign Function Interface 1.0. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`.

[11] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[12] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 162–173. ACM Press, 1998.

[13] W. D. Clinger and L. T. Hansen. Generational garbage collection and the radioactive decay model. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 97–108. ACM Press, 1997.

[14] D. Detlefs. Gcold: a benchmark to stress old-generation collection. `http://www.experimentalstuff.com/Technologies/GCold/`. Last viewed April 13, 2005.

[15] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM Press, 2004.

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

[17] R. R. Fenichel and J. C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, 1969.

[18] B. Hayes. Using key object opportunism to collect old objects. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 33–46. ACM Press, 1991.

[19] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley, 2003.

[20] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[21] B. Lang and F. Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*, pages 253–263, New York, NY, USA, 1987. ACM Press.

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

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

[24] S. Nettles, J. O'Toole, and D. Pierce. Replication-based incremental copying collection. In *Proceedings of the International Workshop on Memory Management*, pages 357–364. Springer-Verlag, 1992.

[25] N. Sachindran and J. E. B. Moss. Mark-copy: fast copying gc with less space overhead. In *Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 326–343. ACM Press, 2003.

[26] F. Smith and G. Morrisett. Comparing mostly-copying and mark-sweep conservative collection. *ACM SIGPLAN Notices*, 34(3):68–78, 1999.

[27] Sun Microsystems, Inc. JNI - Java Native Interface. `http://java.sun.com/j2se/1.4.1/docs/guide/jni/index.html`

[28] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 157–167. ACM Press, 1984.

[29] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–17. ACM Press, 1988.