Controlling Fragmentation and Space Consumption in the Metronome, a Real-time Garbage Collector for Java

David F. Bacon
dfb@watson.ibm.com

Perry Cheng
perryche@us.ibm.com

V.T. Rajan
vtrajan@us.ibm.com

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Now that the use of garbage collection in languages like Java is becoming widely accepted due to the safety and software engineering benefits it provides, there is significant interest in applying garbage collection to hard real-time systems. Past approaches have generally suffered from one of two major flaws: either they were not provably real-time, or they imposed large space overheads to meet the real-time bounds.

Our previous work [3] presented the Metronome, a mostly non-copying real-time collector. The Metronome achieves worst-case pause times of 6 milliseconds while maintaining consistent mutator CPU utilization rates of 50% with only 1.5–2.1 times the maximum heap space required by the application, which is comparable with space requirements for stop-the-world collectors.

However, that algorithm assumed a constant collection rate, ignored program-dependent characteristics, and lacked a precise specification for when to trigger collection or how much defragmentation to perform. This paper refines the model by taking into account program properties such as pointer density, average object size, and locality of object size. This allows us to bound both the time for collection and consequently the space overhead required much more tightly. We show experimentally that most parameters usually are not subject to large variation, indicating that a small number of parameters will be sufficient to predict the time and space requirements accurately.

Our previous work also did not present the details of our approach to avoiding and undoing fragmentation. In this paper we present a more detailed analysis of fragmentation than in previous work, and show how our collector is able to bound fragmentation to acceptable limits.

General Terms

Algorithms, Languages, Measurement, Performance

LCTES'03, June 11–13, 2003, San Diego, California, USA.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Java; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

Keywords

Fragmentation, compaction, space bounds, cost model

1. INTRODUCTION

Garbage collected languages like Java are making significant inroads into domains with hard real-time concerns, such as automotive command-and-control systems. However, the engineering and product life-cycle advantages consequent from the simplicity of programming with garbage collection remain unavailable for use in the core functionality of such systems, where hard real-time constraints must be met. As a result, real-time programming requires the use of multiple languages, or at least (in the case of the Real-Time Specification for Java [7]) two programming models within the same language. Therefore, there is a pressing practical need for a system that can provide real-time guarantees for Java without imposing major penalties in space or time.

In previous work [3], we presented the design and evaluation of a uniprocessor collector that is able to achieve high CPU utilization during collection with far less memory overhead than previous real-time garbage collectors, and that is able to guarantee time and space bounds provided that the application can be accurately characterized in terms of its maximum live memory and average allocation rate over a collection interval.

The contributions of this work are

- A more precise analysis of fragmentation than previous studies of garbage collected systems, with an experimental evaluation of the efficacy of our techniques for avoiding and undoing fragmentation;
- A precise cost model for the time required for performing real-time incremental garbage collection, which allows us to bound the time required for collection and the associated space overhead for real-time operation much more precisely than in previous work; and
- An evaluation of the relationship between mutator CPU utilization level and space consumption in a real-time garbage collected environment.

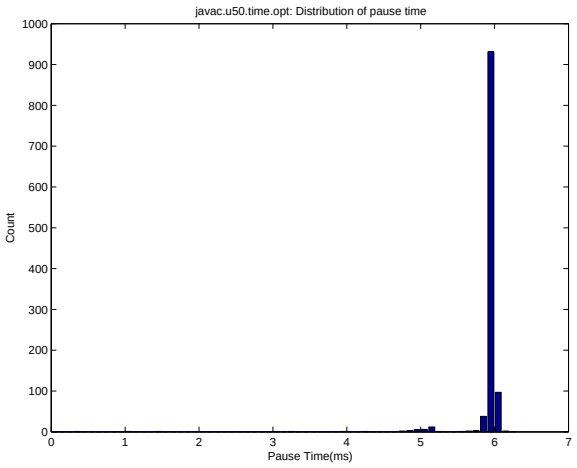

Figure 1: Pause time distributions for javac in the Metronome, with target maximum pause time of 6 ms. Note the absence of a "tail" after the target time.

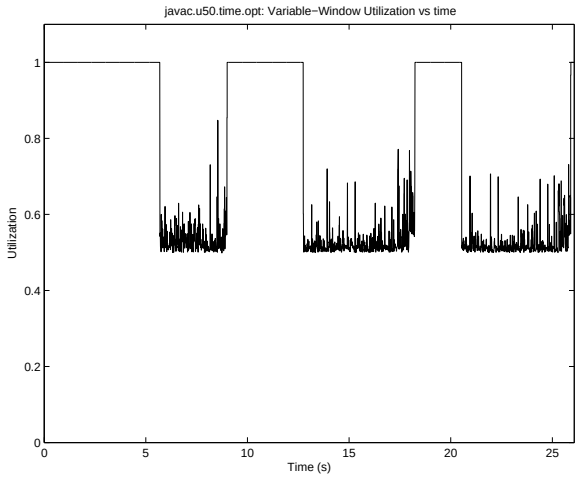

Figure 2: CPU utilization for javac under the Metronome. Mutator interval is 6 ms, collector interval is 6 ms, for an overall utilization target of 50%; the collector achieves this within 3% variation.

The paper is organized as follows: Section 2 gives an overview of our previous work on the Metronome collector. Section 3 analyzes the causes of fragmentation and Section 4 describes our mechanisms for avoidance and removal of fragmentation. Section 5 describes the way in which we obtain time bounds for collection, provides a model for the cost of collection, and evaluates the relevant metrics. Section 6 presents tighter space bounds and describes the conditions for triggering collection. Section 7 discusses related work.

2. OVERVIEW OF THE METRONOME

We begin by summarizing the results of our previous work [3] and describing the algorithm and engineering of the collector in sufficient detail to serve as a basis for understanding the work described in this paper.

Our collector, the Metronome, is an incremental uni-processor collector targeted at embedded systems. It uses a hybrid approach of non-copying mark-sweep (in the common case) and copying collection (when fragmentation occurs).

The collector is a snapshot-at-the-beginning algorithm that allocates objects black (marked). While it has been argued that such a collector can increase floating garbage, the worst-case performance is no different from other approaches and the termination condition is easier to enforce. Other real-time collectors have used a similar approach.

Figures 1 and 2 show the real-time performance of our collector. Unlike previous real-time collectors, there is no "tail" in the distribution of pause times, CPU utilization remains very close to the target, and memory overhead is low — comparable to the requirements of stop-the-world collectors. In this section we explain how the Metronome achieves these goals.

2.1 Features of our Collector

Our collector is based on the following principles:

Segregated Free Lists. Allocation is performed using segregated free lists. Memory is divided into fixed-sized pages, and each page is divided into blocks of a particular size. Objects are allocated from the smallest size class that can contain the object.

Mostly Non-copying. Since fragmentation is rare, objects are usually not moved.

Defragmentation. If a page becomes fragmented due to garbage collection, its objects are moved to another (mostly full) page.

Read Barrier. Relocation of objects is achieved by using a forwarding pointer located in the header of each object [8]. A read barrier maintains a to-space invariant (mutators always see objects in the to-space).

Incremental Mark-Sweep. Collection is a standard incremental mark-sweep similar to Yuasa's snapshot-at-the-beginning algorithm [18] implemented with a weak tricolor invariant. We extend traversal during marking so that it redirects any pointers pointing at from-space so they point at to-space. Therefore, at the end of a marking phase, the relocated objects of the previous collection can be freed.

Arraylets. Large arrays are broken into fixed-size pieces (which we call arraylets) to bound the work of scanning or copying an array and to bound external fragmentation caused by large objects.

Since our collector is not concurrent, we explicitly control the interleaving of the mutator and the collector. We use the term *collection* to refer to a complete mark/sweep/defragment cycle and the term *collector quantum* to refer to a scheduler quantum in which the collector runs.

2.2 Read Barrier

We use a Brooks-style read barrier [8]: each object contains a forwarding pointer that normally points to itself, but when the object has been moved, points to the moved object.

Our collector thus maintains a to-space invariant: the mutator always sees the new version of an object. However, the sets comprising from-space and to-space have a large intersection, rather than being completely disjoint as in a pure copying collector.

While we use a read barrier and a to-space invariant, our collector does not suffer from variations in mutator utilization because

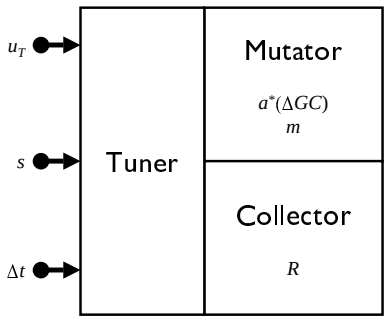

Figure 3: Tuning the performance of an application (mutator) with the collector. The mutator and collector each have certain intrinsic properties (for the mutator, the allocation rate over the time interval of a collection, and the maximum live memory usage; for the collector, the rate at which memory can be traced). In addition, the user can select, at a given time resolution, either the utilization or the space bound (the other parameter will be dependent).

all of the work of finding and moving objects is performed by the collector.

Read barriers, especially when implemented in software, are frequently avoided because they are considered to be too costly. We have shown that this is not the case when they are implemented carefully in an optimizing compiler and the compiler is able to optimize the barriers.

We apply a number of optimizations to reduce the cost of read barriers, including well-known optimizations like common subexpression elimination, as well as special-purpose optimizations like barrier-sinking, in which we sink the barrier down to its point of use, which allows the null-check required by the Java object dereference to be folded into the null-check required by the barrier (since the pointer can be null, the barrier can not perform the forwarding unconditionally).

This optimization works with whatever null-checking approach is used by the run-time system, whether via explicit comparisons or implicit traps on null dereferences. The important point is that we usually avoid introducing extra explicit checks for null, and we guarantee that any exception due to a null pointer occurs at the same place as it would have in the original program.

The result of our optimizations is that for the SPECjvm98 benchmarks, read barriers only have a mean cost of only 4%, or 9.6% in the worst case (in the `201.compress` benchmark).

2.3 Time-Based Scheduling

Our collector can use either time- or work-based scheduling. Most previous work on real-time garbage collection, starting with Baker's algorithm [4], has used work-based scheduling. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The reason for this is simple: work-based algorithms do a little bit of collection work each time the mutator allocates memory. The idea is that by keeping this interruption short, the work of collection will naturally be spread evenly throughout the application. Unfortunately, programs are not uniform in their allocation behavior over short time scales; rather, they are bursty. As a result, work-based strategies suffer from very poor mutator utilization during such bursts of allocation.

In fact, we showed both analytically and experimentally that work-based collectors are subject to these problems and that utilization often drops to 0 at real-time intervals.

Time-based scheduling simply interleaves the collector and the mutator on a fixed schedule. While there has been concern that time-based systems may be subject to space explosion, we have shown that in fact they are quite stable, and only require a small number of coarse parameters that describe the application's memory characteristics in order to function within well-controlled space bounds.

2.4 Provable Real-time Bounds

Our collector achieves guaranteed performance provided the application is correctly characterized by the user. In particular, the user must be able to specify the maximum amount of simultaneously live data m as well as the peak allocation rate over the time interval of a garbage collection $a^\star(\Delta GC)$. The collector is parameterized by its tracing rate R.

Given these characteristics of the mutator and the collector, the user then has the ability to tune the performance of the system using three inter-related parameters: total memory consumption s, minimum guaranteed CPU utilization u_T, and the resolution at which the utilization is calculated Δt.

The relationship between these parameters is shown graphically in Figure 3. The mutator is characterized by its allocation rate over the interval of a garbage collection $a^\star(\Delta GC)$ and by its maximum memory requirement m. The collector is characterized by its collection rate R. The tunable parameters are Δt, the frequency at which the collector is scheduled, and either the CPU utilization level of the application u_T (in which case a memory size s is determined), or a memory size s which determines the utilization level u_T.

3. CAUSES OF FRAGMENTATION

Memory consists of pages of size Π. Each page is divided into fixed-size blocks, with a maximum block size of Σ. Π and Σ are generally powers of two. There are n different block size classes $c_1, \ldots, c_n$, where $c_n = \Sigma$.

The total fragmentation in the system consists of three types:

- unused space at the end of a block (internal fragmentation);
- unused space at the end of a page (which we have named *page-internal fragmentation*); and
- unused blocks that could satisfy a request for a different size object (external fragmentation).

Our use of the term external fragmentation is not the standard one, but it is consistent with the existing definitions.

3.1 Internal Fragmentation

Internal fragmentation is fundamentally expressed as a ratio between the inherent space required by live objects and the actual space they consume. We thus define the absolute internal fragmentation as follows. The memory is considered to be divided into B blocks $b_1, \ldots, b_B$ and $Size(b_i)$ is the size of the block, $Data(b_i)$ is the size of the object residing in the block, and $Live(b_i)$ is 1 if the block contains a live object and 0 otherwise. Thus the absolute

internal fragmentation is

$$F_I = \sum_{i=1}^{B} (Size(b_i) - Data(b_i)) \cdot Live(b_i) \quad (1)$$

and the total live data is

$$m(t) = \sum_{i=1}^{B} Data(b_i) \cdot Live(b_i) \quad (2)$$

so the relative internal fragmentation is

$$f_I = \frac{F_I}{m(t)} \quad (3)$$

Note that this is the definition of fragmentation *at a particular point in time*, and furthermore, the use of the $Live(b_i)$ predicate implicitly assumes that information from the garbage collector marking phase is available and precise.

3.1.1 Controlling Internal Fragmentation

Berger et al. [6] discuss the problems due to internal fragmentation in a segregated list allocator, and use a ratio between adjacent block sizes to bound internal fragmentation in the Hoard collector, such that

$$c_i = \lceil c_{i-1}(1+\rho) \rceil \quad (4)$$

where ρ is the maximum acceptable internal fragmentation ratio. We adopt this strategy as well.

However, all non-defragmenting collectors face a fundamental tradeoff: a small ratio between sizes minimizes internal fragmentation but increases external fragmentation because free blocks are scattered among a larger number of free lists. Johnstone [14] observed that the coarse size classes that are common generally lose more to internal fragmentation than they gain from avoiding external fragmentation.

A major advantage of our collector is that we can allow a small ratio between block sizes because we know that if fragmentation occurs, the blocks will be compacted, allowing us to redistribute unused pages to other sizes. In this sense, our collector is adaptive, responding to the changing object size needs of the application. While it is in a steady state, the distribution of object sizes is relatively constant, and so we expect to perform little defragmentation.

3.2 Page-Internal Fragmentation

Page-internal fragmentation has not received much attention, but can occasionally lead to significant loss of space. In particular, the ratio between the page size Π and the maximum block size Σ must be chosen carefully, or in pathological cases 50% of memory could be wasted.

For instance, a common practice is for objects up to half the page size ($\Pi/2$) to be allocated in blocks, and for larger objects to be allocated as a set of contiguous pages. This means that an application that allocates many objects of size $\Pi/2+1$ will experience approximately 50% fragmentation.

More generally, page-internal fragmentation is bounded by Σ/Π.

Formally, memory is considered to consist of P pages denoted $p_1, \ldots, p_P$. $Size(p_i)$ is the size of the blocks in p_i, and $Live(b_i)$ is 1 if the page contains live blocks and 0 otherwise. Thus the absolute page-internal fragmentation is

$$F_P = \sum_{i=1}^{P} \left(\Pi - Size(p_i) \cdot \left\lfloor \frac{\Pi}{Size(p_i)} \right\rfloor \right) \cdot Live(p_i) \quad (5)$$

and since page-internal fragmentation always occurs on pages that have at least one live block, which must be larger than the fragment at the end of the page, the relative page-internal fragmentation can be expressed as

$$f_P = \frac{F_P}{m(t)} \quad (6)$$

3.3 External Fragmentation

In order to perform defragmentation, we must first be able to define and measure the fragmentation level. On the most naïve level, we say that a page that is discovered to be empty and is therefore usable by any size is not fragmented; similarly, a page that is fully occupied with live objects is not fragmented. Thus fragmentation occurs only in pages that contain a mixture of live and dead objects.

Formally, we define $Live(p_i)$ to be 1 if the page contains live blocks, $Blocks(p_i)$ to be the set of block indices for blocks in p_i, and $Dead(b_j) = 1 - Live(b_j)$. Then in the simplest terms the absolute external fragmentation can be defined as

$$F_{X_S} = \sum_{i=1}^{P} Live(p_i) \cdot \left(\sum_{j \in Blocks(p_i)} Size(b_j) \cdot Dead(b_j) \right) \quad (7)$$

However, this measure is far too conservative. Object allocation, like many other things, generally obeys a locality property: locality of size. That is, if the program allocates a lot of objects of size c_i in the recent past, it is likely to continue to do so in the near future. Thus, a program with locality of size will likely re-use many of the "fragmented" blocks before the next collection, and therefore any effort spent in defragmenting those blocks would be wasted.

If we could perfectly predict the mutator's future behavior, we could define $Useless(b_j)$ to be 1 if block b_j is dead and will not be re-used by the mutator before the next collection, and 0 otherwise. Then a perfect definition of external fragmentation is

$$F_{X_P} = \sum_{i=1}^{P} Live(p_i) \cdot \left(\sum_{j \in Blocks(p_i)} Size(b_j) \cdot Useless(b_j) \right) \quad (8)$$

Of course, we can not make use of such an oracular function. However, we can approximate it fairly accurately in practice by using the locality of size property: the number of objects in size class c_i that were re-used since the last collection will often be a good predictor of the number of objects that will be re-used until the next collection. We call an object that has been dead for more than one collection cycle a *mummy*. We can easily define (and compute) the predicate $Mummy(b_j)$ from the $Dead(b_j)$ predicate applied to the current and previous collection cycles. Then a practical approximation for external fragmentation is

$$F_X = \sum_{i=1}^{P} Live(p_i) \cdot \left(\sum_{j \in Blocks(p_i)} Size(b_j) \cdot Mummy(b_j) \right) \quad (9)$$

3.3.1 Relative External Fragmentation

However, how to define the relative fragmentation is more problematic. Previous work has been almost entirely in the context of explicit memory allocation, where the programmer explicitly calls `free()` to release a block of memory. Even in this environment, the question of how to quantify fragmentation is problematic.

Johnstone and Wilson [15] consider four different ways of measuring fragmentation, three of which are relative to the maximum memory allocated by the program. Note that their model for allocation is that the allocator attempts to satisfy the request from its available pool, and on failure gets more memory from the underlying system using `sbrk()`. As a result, the heap size is a

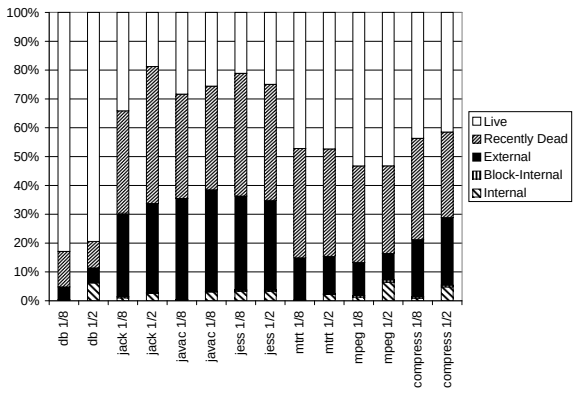

Figure 4: Fragmentation at $\rho = 1/8$ and $\rho = 1/2$.

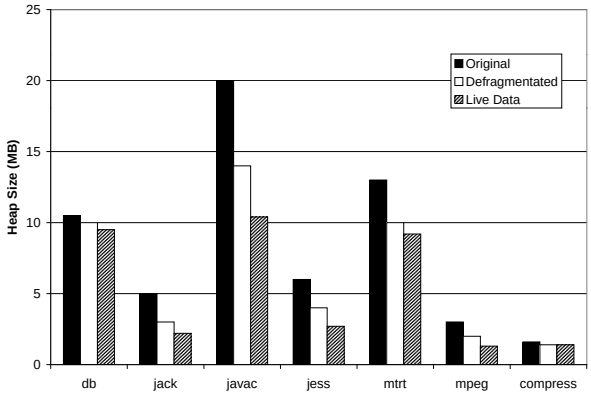

Figure 5: Reduction in maximum heap requirement due to defragmentation

monotonically increasing function, and they are always measuring fragmentation relative to the amount of memory in use at the end of the program.

On the other hand, in a garbage collected system, the application typically runs until some pre-defined heap size is exhausted, at which point collection and possibly compaction (either by semi-space copying, sliding compaction, or other techniques) is performed. Thus if the heap size is s, we could define the relative external fragmentation as $f_X = F_X/s$, but this has the obvious disadvantage that it is entirely dependent upon the heap size with which we run the program, and can therefore misrepresent the quality of the system.

Some collectors heuristically allocate more heap space under certain conditions, and this is often important for achieving good real-world performance, but it further complicates the already confusing question of how to define relative external fragmentation.

Johnstone and Wilson suggest that fragmentation is primarily a problem at allocation spikes, and this is certainly true. However, their primary method for measuring fragmentation is to compare the peak memory utilization (at the end of the run) to the maximum utilization (at any point in the run). Such a measure seems problematic, since the two points are disjoint in time.

Fundamentally, the relative external fragmentation should measure what proportion of the space is being wasted. Therefore, free blocks which are available for any size class should not be considered. Instead, we define relative external fragmentation in terms of the amount of memory which is *not* available for arbitrary allocation, relative to the amount of memory in use:

$$f_X = \frac{F_X}{m(t)} \tag{10}$$

3.4 Size-External Fragmentation

It is further useful to separately consider a particular kind of external fragmentation, namely that which occurs when for example there is a single object of a given size class. In that event, no matter what the size of the object, and no matter whether we are able to perform defragmentation, the object will consume a full page. We call this *size class fragmentation*, since it is a type of external fragmentation that occurs within a size class.

Fortunately, size class fragmentation is limited to one page per size class, so if there are n size classes the size-external fragmentation is bounded by $n \cdot \Pi$. However, since we are aggressive in our use of significant numbers of size classes to avoid internal fragmentation, this overhead may be significant on certain types of memory-constrained devices. In our implementation $n = 44$ and $\Pi = 16$ KB, so $F_S \leq 720$ KB.

Size class fragmentation is the portion of the fragmented data that can not be defragmented. It is computed as follows:

$$A(c_k) = \sum_{\{i \mid Size(p_i) = c_k\}} \left(\sum_{j \in Blocks(p_i)} Size(b_j) \cdot Live(b_j) \right) \tag{11}$$

$$F_S = \sum_{k=1}^{n} \Pi - (A(c_k) \text{ rem } \Pi) \tag{12}$$

where $A(c_k)$ is the total size of allocated blocks for size class c_k, and $F_S(c_k)$ computes the number of residual blocks that could not be compacted onto completely full pages, which is subtracted from the page size to get the number of bytes lost in this "last page". Finally, F_S simply sums the terms for all size classes.

3.5 Measurements

Figure 4 shows the contributions of the different types of fragmentation for the SPECjvm98 benchmarks using two different geometric size class progressions: $\rho = 1/8$, which is what we use in our system, and $\rho = 1/2$, which is a common value in systems that are not able to perform dynamic defragmentation.

The bottom-most bar shows the contribution of internal fragmentation which varies with ρ, which is most apparent in `db`, `javac`, `mpeg`, and `compress`.

Not surprisingly, page-internal fragmentation is negligible, although it is visible for `mpeg`.

The black bars show true external fragmentation: blocks that have mummified, or not been allocated for more than a full collection cycle. These blocks have reduced the effective heap size and forced collection to begin sooner, reducing the overall throughput of the system.

The "Recently Dead" bar indicates blocks that are free but were in use during the previous collection cycle; therefore, due to local-

ity of size we expect to re-use them during the coming cycle and do not consider them as contributing to external fragmentation.

Figure 5 shows the effectiveness of defragmentation by measuring the minimum heap size in which the application can run when the collector operates in stop-the-world mode. Obviously, the real-time collector does not stop the mutators; this is more of a limit study of the capability of defragmentation to reduce heap usage.

The graph shows the original heap requirement, the heap requirement with defragmentation turned on, and the maximum live data requirement of the application (which is the lower bound). With the exception of `javac`, defragmentation allows us to run the application in a heap that is very close to the minimum size.

4. DEFRAGMENTATION

Now that we have defined the various kinds of fragmentation, we can summarize our strategy for preventing fragmentation overall:

- internal fragmentation is limited to ρ by setting neighboring size classes to $c_i = \lceil c_{i-1}(1+\rho) \rceil$;
- page-internal fragmentation is limited to Σ/Π; and
- external fragmentation is eliminated by the use of arraylets and explicit defragmentation as required to maintain real-time bounds.

In our implementation, we use $\Pi = 2^{14}$, or 16 KB pages and a maximum block size $\Sigma = 2^{11}$ or 2 KB. We use $\rho = 1/8$ to limit internal fragmentation, which gives $n = 44$ size classes.

Since we never subdivide pages into objects larger than $\Sigma = 2^{11}$, the loss of space in our collector due to page-internal fragmentation is bounded by $\Sigma/\Pi = 1/8$.

As we will describe in Section 6.2, our system automatically determines how many pages must be defragmented in order for the system to be able to guarantee that it will continue to meet real-time bounds.

Given this target number of pages, we divide the defragmentation work as evenly as possible across the individual size classes. Each size class consists of a linked list of pages.

The algorithm we use for defragmenting a size class is:

1. sort the pages by the number of unused objects per page.
2. set the allocation pointer to the first non-full page in the resulting list.
3. set the page to evacuate to the last page in the list.
4. while the target number of pages to evacuate in this size class has not been met, and the the page to evacuate does not equal the page in which to allocate, move the live objects from the page to evacuate into the next available object in the page at the head of the list.

This strategy performs defragmentation by moving objects from the most lightly used pages onto pages that are almost full. The result is that the minimal number of object moves is performed, and the maximal number of completely full pages is created. The latter part of the strategy may result in poor cache locality, so it might be more desirable to move objects onto pages that have a larger number of free blocks in order to preserve some of the locality among the moved objects.

5. TIME BOUNDS

In our previous work [3], the time required to collect m MB of live memory was described by a single parameter, R, the collection rate in MB/s [1]. However, our measurements showed that the actual collection rate varied from application to application by as much as 50% (36.7–57.4 MB/s). Furthermore, this constant assumes that collection time is entirely dependent on the mark phase of collection. While this is often true, it is somewhat imprecise and may be false for certain unusual applications.

In a time-scheduled collector, the parameter R determines the amount of excess memory e required for the mutator to be able to continue to run while a full collection is performed.

In this section we refine the notion of how we estimate the time required to perform a collection. In the following section we show how this can be used to tighten the bound on the heap size necessary in order to meet real-time requirements.

5.1 Our Previous Work

For clarity of exposition we briefly recapitulate the derivation of time and space bounds from our previous work [3]. In Sections 5.2 and later we will show how these derivations can be refined to produce tighter space bounds

We can define the real-time behavior of the combined system comprising the user program and our garbage collector with the following parameters:

- $\alpha^\star(\tau_1, \tau_2)$ is the amount of memory allocated in the time interval (τ_1, τ_2) in MB/s.
- $m^\star(\tau)$ is the total amount of live memory at time τ in MB (excluding garbage, overhead, and fragmentation).
- R is the garbage collector processing rate (MB/s). Since ours is a tracing collector, this is measured over live data.

A time τ is on an idealized axis in which the collector runs infinitely fast — we call this *mutator time*. As a practical matter this can be thought of as time measured when the program has sufficient memory to run without garbage collecting.

5.1.1 Mapping Between Mutator and Real Time

Now consider a realistic execution in which the collector is not infinitely fast. Execution will consist of alternate executions of mutator and collector. Time along real time axis will be denoted with the variable t.

The function $\Phi(t) \rightarrow \tau$ maps from real to mutator time, where $\tau \leq t$. Functions that operate in mutator time are written $f^\star(\tau)$ while functions that operate in real time are written $f(t)$.

The live memory of the program at time t is thus

$$m(t) = m^\star(\Phi(t)) \tag{13}$$

and the maximum memory requirement over the entire program execution is

$$m = \max_t m(t) = \max_\tau m^\star(\tau) \tag{14}$$

5.1.2 Time-Based Scheduling

Time-based scheduling interleaves the collector and mutator using fixed time quanta. It thus results in even CPU utilization but is subject to variations in memory requirements if the memory allocation rate is uneven. A time-based real-time collector has two additional fundamental parameters:

[1] In [3] we called this parameter P, but we use R in this paper because we have used P to denote the number of pages.

- Q_T is the mutator quantum: the amount of time (in seconds) that the mutator is allowed to run before the collector is allowed to operate.

- C_T is the time-based collector quantum (in seconds of collection time).

For the time being, we assume that the scheduler is perfect, in the sense that it always schedules the mutator for precisely Q_T seconds. A typical value for Q_T might be 10 ms.

Cheng and Blelloch [10] have defined the *minimum mutator utilization* or MMU for a given time interval Δt as the minimum CPU utilization by the mutator over all intervals of width Δt. From the parameters Q_T and C_T we can derive the MMU as

$$u_T(\Delta t) = \frac{Q_T \cdot \left\lfloor \frac{\Delta t}{Q_T + C_T} \right\rfloor + x}{\Delta t} \tag{15}$$

where the first term in the numerator corresponds to the number of whole mutator quanta in the interval, and the x term corresponds to the size of the remaining partial mutator quantum, which is defined as

$$x = \max\left(0, \Delta t - (Q_T + C_T) \cdot \left\lfloor \frac{\Delta t}{Q_T + C_T} \right\rfloor - C_T\right) \tag{16}$$

While this expression is fairly awkward, as the number of intervals becomes large, it reduces to the straightforward utilization expression

$$\lim_{\Delta t \to \infty} u_T(\Delta t) = \frac{Q_T}{Q_T + C_T} \tag{17}$$

Now consider the space utilization of a time-scheduled collector. Since we are assuming the collection rate is constant, at time t the collector will run for $m(t)/R$ seconds to process the $m(t)$ live data (since our collector is trace-based, work is essentially proportional to live data and not garbage). In that time, the mutator will run for Q_T seconds per C_T seconds executed by the collector. Therefore, in order to run a collection at time t, we require excess space of

$$e_T(t) = \alpha^\star\left(\Phi(t), \Phi(t) + \frac{m(t)}{R} \cdot \frac{Q_T}{C_T}\right) \tag{18}$$

We further define the maximum excess space required as

$$e_T = \max_t e_T(t) \tag{19}$$

Freeing an object in our collector may take as many as three collections: the first is to collect the object; the second is because the object may have become garbage immediately after a collection began, and will therefore not be discovered until the following collection cycle; and the third is because we may need to relocate the object in order to make use of its space. The first two properties are universal to incremental collection; the third is specific to our approach.

As a result, the space requirement of our collector paired with a given application (including unreclaimed garbage, but not including internal fragmentation) at time t is

$$s_T(t) \leq m(t) + 3e_T \tag{20}$$

and the overall space requirement is

$$s_T \leq m + 3e_T \tag{21}$$

However, the expected space utilization is only $m + 2e_T$, and the worst-case utilization is highly unlikely; this is discussed in more detail below.

5.2 Refining the Collection Cost Model

The above analysis assumed that the cost of collection could be modelled with a single gross parameter R, which measures the collection rate in MB/s of live data. However, this is inaccurate in two respects: first, it assumes that collection cost is totally dominated by the mark phase, and while this is generally true it is not always so and other phases may take non-trivial amounts of time. Second, it assumes that the "shape" of memory is uniform, and that a single trace rate can apply to all applications or even to all points in the execution of a single application.

We now examine the actual cost of collection, and investigate how the cost model for collection can be refined. This is particularly important because the parameter R directly controls e_T, the excess space required to run the garbage collector, which is multiplied by 3 in the equation for the total space requirement (21). Therefore if we can more tightly bound the collection rate, we can reap a threefold reduction in space overhead.

In order to evaluate the time required for a garbage collection, it is necessary to examine the phases that comprise a collection in detail. The phases and the associated functions that compute the time for each phase are:

Initialize and Terminate. This is the constant-time overhead of beginning and ending a collection (T_I).

Root Scan. Scan the thread stacks for root pointers (T_R).

Mark. Traverse the object graph starting at the roots and mark all objects encountered as "live" (T_M).

Sweep. Move pages with no live objects onto the list of free pages; mark remaining "unmarked" objects as "free" (T_S).

Defragment. If the number of free pages falls below a threshold, move objects from mostly empty to mostly full pages (T_D).

We can therefore define the time required to perform a collection at time t in terms of the quantities defined above as

$$T_{GC}(t) = T_I + T_R(t) + T_M(t) + T_S(t) + T_D(t) \tag{22}$$

This allows us to rewrite equation (18) more precisely as

$$e_T(t) = \alpha^\star\left(\Phi(t), \Phi(t) + T_{GC}(t) \cdot \frac{Q_T}{C_T}\right) \tag{23}$$

We now precisely define each of these component functions in terms of the live memory $m(t)$, a set of parameters which characterize the application (denoted by lower-case Greek letters), and a set of cost coefficients which are obtained by measuring the collector (denoted Ψ_x).

We characterize the mutator in terms of five parameters. In general, this is too many parameters to expect the user to supply. However, in the quantitative analysis we will show that the collection time is insensitive to some parameters, and that other parameters vary very little across applications. Therefore, we believe it will be practical to specify the behavior of an application very precisely with only one or two parameters.

5.2.1 Pointer Density and Mark Phase Cost

The mark phase is by far the most expensive in our collector. It is therefore of paramount importance to estimate its cost accurately.

We already know that the speed of marking, measured in MB/s, varies by about 50% across applications with our collector. In fact, it is surprising that it varies so little. We now consider the cost of marking in more detail.

When we process a pointer, there are three cases that may be handled: (1) if the pointer is null, we simply ignore it; (2) we check if the referent is marked already; if so, we are done; (3) otherwise, we mark the referent and process all the pointers it contains.

Now consider the processing of a pointer to an array. If the pointer is null, the check has taken Ψ_{M_2} computation steps, and 0 bytes of memory have been marked. Otherwise, assume the pointer is non-null and the referent is a 1 MB array. If the array is already marked, the check has taken Ψ_{M_3} computation steps and 0 bytes have been marked. Finally, if the array has not previously been marked, it is marked, which takes Ψ_{M_1} computation steps, and 1 MB of memory has been marked.

Furthermore, if the array is an array of pointers, each of those pointers must now be processed. However, for accounting purposes we consider this cost to be part of Ψ_{M_2}.

In order to characterize the marking cost more precisely, we define the following quantities:

- $r(t)$ is number of references in live memory at time t, including root pointers. $r(t) \leq m(t)/W$, where W is the word size of the machine;
- $r'(t)$ is the number of non-null references in live memory at time t, $r'(t) \leq r(t)$; and
- $l(t)$ is the number of live objects in memory at time t, $l(t) \leq r'(t)$.

In theoretical terms, since Ψ_{M_1}, Ψ_{M_2}, and Ψ_{M_3}, are constants, the complexity of the marking phase is $O(r(t))$.

Of course, in a real-time system we care very much about the constant factors. We can characterize the cost of marking much more accurately as

$$T_M(t) = \Psi_{M_1} \cdot l(t) + \Psi_{M_2} \cdot r(t) + \Psi_{M_3} \cdot r'(t) \tag{24}$$

With the proper values for the coefficients, this formula gives us an accurate accounting of the mark time of the collector. However, forcing the user to supply the quantities $r(t)$, $r'(t)$, and $l(t)$ directly is inconvenient and error-prone.

Instead, we use scaled parameters that are independent of the heap size:

- μ is the proportion of memory devoted to the heap (as opposed to thread stacks),
- ω is a lower bound on the average object size (in bytes),
- π is a lower bound on the number of pointers per word, and
- ν is an upper bound on the fraction of non-null references.

Given these parameters we can bound the quantities necessary to compute T_M from the size of live memory $m(t)$:

$$l(t) \leq \frac{\mu \cdot m(t)}{\omega} \tag{25}$$

$$r(t) \leq \frac{\mu \cdot m(t)}{\mathcal{W}} \cdot \pi \tag{26}$$

$$r'(t) \leq r(t) \cdot \nu \tag{27}$$

where $\mathcal{W}$ is the word size of the machine in bytes.

The use of the parameters μ, ω, π, and ν is twofold: first of all, they are more intuitive for the user of the system; and second, since they are independent of heap size, they are more likely to be stable across both multiple runs of the same program and across different programs. This provides us with the opportunity to start the system with reasonable default values and only require tuning of those parameters that are atypical.

Thus we can express $T_M(t)$ in terms of these simple parameters and the maximum live memory as

$$T_M(t) \leq m(t) \cdot \mu \left(\frac{\Psi_{M_1}}{\omega} + \frac{\pi}{\mathcal{W}} \cdot (\Psi_{M_2} + \Psi_{M_3} \nu) \right) \tag{28}$$

5.2.2 Root Scan Cost

In order to characterize the root scanning phase precisely we define the following quantities:

- $\sigma(t)$ is the amount of memory devoted to thread stacks,
- $r_\sigma(t)$ is the number of references on the stacks, and
- $r'_\sigma(t)$ is the number of non-null references on the stacks.

We can then define the cost of stack scanning as

$$T_R(t) = \Psi_{R_1} \cdot \sigma(t) + \Psi_{R_2} \cdot r_\sigma(t) + \Psi_{R_3} \cdot r'_\sigma(t) \tag{29}$$

where Ψ_{R_1}, Ψ_{R_2}, and Ψ_{R_3} are the cost coefficients of scanning stack memory, examining pointers on the stack, and placing non-null pointers on the mark stack, respectively.

As we did with our estimate of the cost of the mark phase, we can use the parameters describing the application to obtain bounds on the cost of each component in more intuitive terms:

$$\sigma(t) \leq (1 - \mu) \cdot m(t) \tag{30}$$

$$r_\sigma(t) \leq \sigma(t)/\mathcal{W} \cdot \pi \tag{31}$$

$$r'_\sigma(t) \leq \nu \cdot r_\sigma(t) \tag{32}$$

which allows us to express the cost of the root scanning phase as

$$T_R(t) \leq m(t) \cdot (1 - \mu) \left(\Psi_{R_1} + \frac{\pi}{\mathcal{W}} (\Psi_{R_2} + \Psi_{R_3} \nu) \right) \tag{33}$$

5.2.3 Sweep Phase Cost

In the sweep phase we must examine every live page, and then for every live page we must examine and possibly update the mark vector associated with the objects on the page. Thus if there are P live pages and B blocks on those pages the cost of the sweep phase is

$$T_S(t) = \Psi_{S_1} \cdot P(t) + \Psi_{S_2} \cdot B(t) \tag{34}$$

where Ψ_{S_1} and Ψ_{S_2} are the cost coefficients of processing a page and a block, respectively.

The number of pages can be bounded by $s_T(t)/\Pi$, but since $s_T(t)$ depends on $e_T(t)$, which in turn depends on the time required for collection, using $s_T(t)$ would lead to a recurrence, which we wish to avoid.

We have found experimentally that the space required by the collector has not exceeded $2.5 \cdot m$, and since the cost of the sweep phase is a relatively small component of the overall cost, we can afford to be conservative in our cost estimation.

Therefore, to avoid the necessity of introducing a recurrence into the formula for the space required by the collector, we simply assume that there is an "expansion factor" parameter ϕ which bounds the size of the heap. By default we set $\phi = 2.5$. Therefore, the size of the heap is $\phi \cdot m(t)$ and the number of live pages is

$$P(t) \leq \frac{m(t)}{\Pi} \cdot \phi \tag{35}$$

and the number of blocks on live pages is bounded by

$$B(t) \leq \frac{\Pi \cdot P(t)}{\omega} \tag{36}$$

	m	s	Collector: Measured Cost						Parameter					
Benchmark	(MB)	(MB)	T_{GC}	T_I	T_R	T_M	T_S	T_D	μ	ω	π	ν	ϕ	λ
javac	86	172	2.21	0.001	0.061	1.973	0.137	0.124	0.99997	208	0.11	0.62	2.00	0.95
db	82	137	2.63	0.001	0.043	2.408	0.148	0.063	0.99993	162	0.09	0.62	1.67	0.82
jack	82	146	1.73	0.001	0.042	1.076	0.094	0.047	0.99994	481	0.08	0.52	1.78	0.97
mtrt	80	122	1.59	0.001	0.046	1.386	0.115	0.078	0.99996	150	0.10	0.64	1.53	0.68
jess	73	126	0.63	0.001	0.186	0.554	0.046	0.031	0.99996	149	0.09	0.64	1.73	0.89
fragger	72	151	3.20	0.001	0.147	1.700	0.175	1.295	0.99999	744	0.17	0.13	2.10	0.03

Table 1: Cost of a single garbage collection phase (in seconds) and the measured values for the parameters that characterize the collection cost. μ is the fraction of memory devoted the heap; ω is the average object size; π is the pointer density; ν is the fraction of non-null pointers; ϕ is the expansion factor for the heap; and λ is the locality of size.

and thus

$$T_S(t) \leq m(t) \cdot \phi \left(\frac{\Psi_{S_1}}{\Pi} + \frac{\Psi_{S_2}}{\omega} \right) \quad (37)$$

5.2.4 *Defragment Phase*

In some ways, the cost of the defragment phase is the most difficult to model. In theory, defragmentation could require the relocation of all live memory $m(t)$; however, in practice we find that even adversary programs cause less than 3% of live memory to be moved. In order to accomadate the potential for large variation in cost, we introduce an additional parameter, in the same manner as we did for the mark phase cost estimation.

λ is a measure of the *locality of size* of the application. If $\lambda = 1$, the program always re-uses blocks with objects of the same size, and therefore there is no need for defragmentation. On the other hand, if $\lambda = 0$, there is no re-use and the maximum amount of defragmentation work must be performed at each collection.

Locality of size is a bit more complex to define than the other parameters we have used to characterize the application. We define λ as follows: for each size class i, we consider the number of bytes freed by a collection f_i and the number of bytes allocated during that collection cycle a_i. If $f_i = a_i$ then the collector has freed exactly as much memory in size class i as it has allocated, and no defragmentation is necessary. On the other hand, if $f_i \neq a_i$, then the smaller of the two denotes the amount of memory that can be reused in size class i (if a_i is smaller, then we do not reuse all available memory; if f_i is smaller then all available memory is reused).

In order to obtain a measure of locality we scale the sizes by the total memory. The total bytes freed and allocated across all size classes are denoted f and a. Then the locality of size across all n size classes is

$$\lambda = \sum_{i \in 1 \ldots n} \min \left(\frac{f_i}{f}, \frac{a_i}{a} \right) \quad (38)$$

Then the cost of defragmentation is

$$T_D(t) \leq (1 - \lambda)(\Psi_{D_1} \cdot P(t) + \Psi_{D_2} \cdot l(t) + \Psi_{D_3} \cdot m(t)) \quad (39)$$

or in reduced form

$$T_D(t) \leq m(t) \cdot (1 - \lambda) \left(\frac{\Psi_{D_1}}{\Pi} \cdot \phi + \frac{\Psi_{D_2}}{\omega} + \Psi_{D_3} \right) \quad (40)$$

5.3 A Tighter Time Bound

Based on these characterizations, we are able to define a tight bound on the time required for garbage collection, namely

$$T_{GC}(t) \leq T_I + m(t) \cdot \left(\begin{array}{l} \mu \left(\frac{\Psi_{M_1}}{\omega} + \frac{\pi}{\mathcal{W}} \cdot (\Psi_{M_2} + \Psi_{M_3} \nu) \right) + \\ (1 - \mu) \left(\Psi_{R_1} + \frac{\pi}{\mathcal{W}} (\Psi_{R_2} + \Psi_{R_3} \nu) \right) + \\ \phi \left(\frac{\Psi_{S_1}}{\Pi} + \frac{\Psi_{S_2}}{\omega} \right) + \\ (1 - \lambda) \left(\frac{\Psi_{D_1}}{\Pi} \cdot \phi + \frac{\Psi_{D_2}}{\omega} + \Psi_{D_3} \right) \end{array} \right) \quad (41)$$

Of course, this equation is horribly complex. Fortunately, the user of our system does not need to be aware of this complexity. They can simply specify the small number of parameters that accurately characterize their application. In the following section we will show experimentally which parameters need to be specified and how well the model fits reality.

5.4 Measurements

Table 1 shows the measured cost of each phase of collection, and the measured values of parameters which are used to characterize the cost of collection. The benchmarks include a selection from SPECjvm98 as well as a synthetic benchmark designed to act as a fragmentation adversary program.

The results are well in line with expectations: we expected the time required for the mark phase to dominate all other phases of collection.

The sweep phase only consumes 5-7% of the total collection time. The defragmentation phase consumes only 2-6% of collection time for realistic programs. However, for the adversary program `fragger` it consumes 40% of the total collection time. None of the programs is deeply recursive, so stack scan time is negligible.

The important result from the measurements of the parameters in the right-hand side of the table is that as we had predicted, many of them are very stable across a variety of benchmarks. In particular, the heap fraction of memory μ, the pointer density π, and the fraction of non-null pointers ν show very little variation.

The parameter λ tracks the cost of defragmentation quite well, except that it is abnormally low for `mtrt`. It accurately shows that the locality of size for `fragger` is essentially 0, because no reuse is possible. On the other hand, the locality of size is greater than 80% for all benchmarks except `mtrt`, even though it does not spend much time defragmenting. The reason is that λ is conservative in the sense that it does not account for defragmentation that is avoided due to entire pages becoming free when all objects on them die. Better definitions of λ are a subject of continued research.

One of the difficulties we encountered in evaluating the system is that because Jikes RVM system in which we implemented our collector is written in Java [1], a large portion of the heap consists of system data structures. In particular, the "boot image", which is the compiled Java code image which is dumped into a memory

segment and loaded at virtual machine boot time, contains 55 MB. This is part of the heap and is marked and swept on every collection, even though dead objects in the boot heap are not freed (the boot heap is mutable). Thus the m and s sizes include this data from the boot heap.

The result is that the boot heap has an undue influence on our measurements and on our collector performance. This is most obvious in the surprisingly high value for the average object size ω. The boot heap contains a very large proportion of compiled code objects, which tend to be large and have a low (zero) pointer density. In future work we plan to treat the boot heap generationally, as an old generation which is never collected. This will allow almost all collector work on the boot heap to be eliminated, and should both improve the performance of our collector and give more application-specific data for the parameters in Table 1.

However, we stress that while the presence of the boot heap has the unfortunate effect of skewing some of the measured values, it has no effect on the fundamental real-time nature of the system, or on the effectiveness of the algorithms for scheduling or performing defragmentation which follow. In particular, it does not affect the locality of size λ.

Dieckmann and Hölzle [11] have performed a study of a set of Java benchmark characteristics which measured some of these parameters. Their results were significantly different. This is due to two factors: they studied an abstract "perfect" virtual machine with no overheads (such as the Brooks forwarding pointer in our object headers), and our virtual machine is written in Java and the VM structures are subject to collection like everything else.

6. SPACE BOUNDS AND TRIGGERS

We now refine the space requirements for the collector by reasoning more carefully about equation 21. Our previous analysis [3], shown in Section 5.1.2, required $3e_T$ extra space in order to run the application while meeting real-time bounds. By considering defragmentation separately we are able to considerably improve upon this bound.

To begin with, we separate e_T into two components: the space required to perform a defragmentation phase of the collection $e_d(t)$, and the space required to perform all the other phases $e_g(t)$ (principally consisting of mark and sweep):

$$e_T(t) = e_g(t) + e_d(t) \tag{42}$$

By equations 23, 39, and 41 we are can define the components as

$$e_g(t) = \alpha^\star \left(\Phi(t), \Phi(t) + (T_{GC}(t) - T_D(t)) \cdot \frac{Q_T}{C_T} \right) \tag{43}$$

$$e_d(t) = \alpha^\star \left(\Phi(t), \Phi(t) + T_D(t) \cdot \frac{Q_T}{C_T} \right) \tag{44}$$

and as before we will denote the maximal values by e_g and e_d.

Let e_{d0} be the value of e_d assuming that there is no locality of size ($\lambda = 0$). Then

$$e_d = (1 - \lambda) e_{d0} \tag{45}$$

because $T_D(t)$ is scaled by $1 - \lambda$ and we assume that the allocation rate is nearly constant over the time scale of a defragmentation cycle, allowing the $1 - \lambda$ factor to be pulled out of the equation for $T_D(t)$.

In the worst case, all memory allocated during a garbage collection will be floating garbage (that is, $e_g + e_d$ bytes). Furthermore, that memory will not become available until the end of the sweep phase of the following collection, during which time an additional e_g bytes will have been allocated. So the total floating garbage can be as high as $2e_g + e_d$ bytes.

In addition, in a steady state, we will have to defragment as many pages as we allocate. The size in bytes of free pages allocated during a collection is $(1 - \lambda)(e_g + e_d)$, because due to locality of size, $\lambda(e_g + e_d)$ will be allocated out of existing formatted pages. Because defragmented pages consume memory for an entire collection cycle, this must be added to the memory overhead.

Therefore, the total memory requirement is

$$s_T \leq m + 2e_g + e_d + (1 - \lambda)\left[e_g + e_d\right] \tag{46}$$

which we can rewrite in terms of e_{d0} and simplify as

$$s_T \leq m + 2e_g + (1 - \lambda)\left[e_g + (2 - \lambda)e_{d0}\right] \tag{47}$$

Note that in the case when there is complete locality of size ($\lambda = 1$) then the equation degenerates into $m + 2e_g$, which is the standard formula for space overhead in an incremental collector. On the other hand, if there is no locality of size ($\lambda = 0$), the equation degenerates to $m + 3e_g + 2e_{d0}$, a slightly more precise version of equation 21. In general, we expect λ to be high, so the space requirement in practice is not substantially higher than for a standard incremental collection algorithm.

6.1 When to Collect?

Given our mechanism for garbage collection and defragmentation, we still require an algorithm for determining the proper time to invoke a garbage collection, and for determining how many pages must be freed by the defragmentation phase in that collection.

The algorithm depends on tracking the number of *free pages* in the system, since in order to respond to a worst-case scenario, we must have free pages available that can be assigned to any size class.

Let p_g be the maximum number of free pages allocated during a garbage collection, excluding the defragmentation phase; and p_d be the maximum number of pages allocated during the defragmentation phase. Since we allocate e_g bytes during a collection (excluding the defragmentation phase) and due to locality of size λ of those bytes will be allocated from existing pages, and given the page size Π, then the number of new pages allocated is

$$p_g = \frac{(1 - \lambda)e_g}{\Pi} \tag{48}$$

and similarly

$$p_d = \frac{(1 - \lambda)e_d}{\Pi} = \frac{(1 - \lambda)^2 e_{d0}}{\Pi} \tag{49}$$

Let $f(t)$ be the number of free pages at time t and $d(t)$ be the number of defragmented pages at time t (these pages will become free at the conclusion of the next mark phase, which will update all the forwarding pointers, thereby removing references to the defragmented pages).

Now consider the requirements to run garbage collection to completion at time t. We require that the number of free pages

$$f(t) \geq p_g \tag{50}$$

In addition, the number of free and defragmented pages together must be sufficient to run until the end of the sweep phase of the following collection:

$$f(t) + d(t) \geq 2p_g + p_d \tag{51}$$

These two conditions together determine when garbage collection is triggered: if either condition is about to be violated, a collection is triggered. Thus the trigger is checked on the slow path of

allocation, when a new page is allocated (reducing the number of free pages).

Since $f(t)$ changes with every page allocation, but $d(t)$ is stable except at defragmentation, we combine these two equations into

$$f(t) \geq \max(p_g, 2p_g + p_d - d(t)) \quad (52)$$

6.2 How Much to Defragment?

In order to ensure that equation 52 can be satisfied while never requiring the mutator to wait for an allocation request, we must determine how many pages to defragment during each collection, thereby determining $d(t)$.

Since there is a time lag in the availability of pages freed by defragmentation, the number of pages defragmented must be sufficient to ensure that the *next* collection can run to completion (of necessity we have already ensured that the current collection can complete).

More precisely, the number of free pages plus the number of pages defragmented must be sufficient for two collection cycles, or expressed in terms of the pages to defragment:

$$d(t) \geq 2(p_g + p_d) - f(t) \quad (53)$$

7. RELATED WORK

Most so-called "real-time" garbage collection algorithms do not guarantee mutator utilization levels[2, 5, 8, 9, 14, 16, 18], and therefore do not need to calculate the collection trigger precisely as we do (because they effectively slow down the mutator in order to allow collection to terminate before running out of memory).

Henriksson [13] provides a careful analysis of schedulability in an environment where the application is divided into high- and low-priority tasks. It is assumed that high-priority tasks are periodic and very precisely characterized. The advantage of our approach is that it applies to much more general applications, and allows them to be characterized precisely using a small number of parameters.

7.1 Fragmentation

Early work, particularly for Lisp, often assumed that all memory consisted of CONS cells and that fragmentation was therefore a non-issue. Baker's Treadmill [5] also only handles a single object size.

Johnstone [14] showed that fragmentation was often not a major problem for a family of C and C++ benchmarks, and built a non-moving "real-time" collector based on the assumption that fragmentation could be ignored. However, these measurements are based on relatively short-running programs, and we believe they do not apply to long-running systems like continuous-loop embedded devices, PDAs, or web servers. Fundamentally, this is an average-case rather than a worst-case assumption, and meeting real-time bounds requires handling worst-case scenarios.

Furthermore, the use of dynamically allocated strings in Java combined with the heavy use of strings in web-related processing is likely to make object sizes less predictable.

Dimpsey et al. [12] describe the compaction avoidance techniques in the IBM product JVM, which are based on Johnstone's work. They show that these techniques can work quite well in practice. However, when compaction does occur it is very expensive.

Siebert [17] suggests that a single block size can be used for Java by allocating large objects as linked lists and large arrays as trees. However, this approach has simply traded external fragmentation for internal fragmentation. Siebert suggests a block size of 64 bytes; if there are a large number of 8-byte objects, internal fragmentation can cause a factor of 8 increase in memory requirements.

8. CONCLUSIONS

We have presented a hybrid real-time collector that operates primarily as a non-moving incremental mark-sweep collector, but prevents fragmentation via the use of limited copying (in real programs, no more than 6% of collection time is spent in defragmentation). Because fragmentation is bounded, the collector has a provable space bound yet retains a lower space overhead than a fully-copying real-time collector.

We have presented a detailed analysis of fragmentation, and have described how our collector bounds internal fragmentation and corrects external fragmentation by moving objects and freeing lightly used pages. We have presented measurements that show the efficacy of these techniques, which indicate that our defragmentation strategy is capable of substantially reducing the minimum heap size required by applications.

We have also presented a detailed model for the cost of collection, and shown how this can be used to tighten both the space and time bounds for real-time collection. The cost of collection is parameterized by six quantities that describe the application being collected. We have measured these quantities and shown that many of them are quite stable across a range of benchmarks, indicating that for many applications default values can be provided by the system, simplifying the user interface to the collector.

9. REFERENCES

[1] ALPERN, B., ET AL. Implementing Jalapeño in Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 1999). *SIGPLAN Notices, 34*, 10, 314–324.

[2] APPEL, A. W., ELLIS, J. R., AND LI, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988). *SIGPLAN Notices, 23*, 7 (July), 11–20.

[3] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003), pp. 285–298.

[4] BAKER, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

[5] BAKER, H. G. The Treadmill, real-time garbage collection without motion sickness. *SIGPLAN Notices 27*, 3 (Mar. 1992), 66–70.

[6] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)* (Cambridge, Massachussetts, Nov. 2000). *SIGPLAN Notices, 35*, 11, 117–128.

[7] BOLLELLA, G., GOSLING, J., BROSGOL, B. M., DIBBLE, P., FURR, S., HARDIN, D., AND TURNBULL, M. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000.

[8] BROOKS, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), G. L. Steele, Ed., pp. 256–262.

[9] CHEADLE, A. M., FIELD, A. J., MARLOW, S., PEYTON JONES, S. L., AND WHILE, R. L. Non-stop Haskell. In *Proc. of the Fifth International Conference on Functional Programming* (Montreal, Quebec, Sept. 2000). *SIGPLAN Notices, 35*, 9, 257–267.

[10] CHENG, P., AND BLELLOCH, G. A parallel, real-time garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices, 36*, 5 (May), 125–136.

[11] DIECKMANN, S., AND HÖLZLE, U. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proc. of the Thirteenth European Conference on Object-Oriented Programming* (1999), R. Guerraoui, Ed., vol. 1628 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 92–115.

[12] DIMPSEY, R., ARORA, R., AND KUIPER, K. Java server performance: A case study of building efficient, scalable JVMs. *IBM Syst. J. 39*, 1 (2000), 151–174.

[13] HENRIKSSON, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[14] JOHNSTONE, M. S. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Dec. 1997.

[15] JOHNSTONE, M. S., AND WILSON, P. R. The memory fragmentation problem: Solved? In *Proc. of the ACM SIGPLAN International Symposium on Memory Management* (Mar. 1999). *SIGPLAN Notices, 34*, 3, 26–36.

[16] LAROSE, M., AND FEELEY, M. A compacting incremental collector and its performance in a production quality compiler. In *Proc. of the First International Symposium on Memory Management* (Vancouver, B.C., Oct. 1998). *SIGPLAN Notices, 34*, 3 (Mar., 1999), 1–9.

[17] SIEBERT, F. Eliminating external fragmentation in a non-moving garbage collector for Java. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (San Jose, California, Nov. 2000), pp. 9–17.

[18] YUASA, T. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software 11*, 3 (Mar. 1990), 181–198.