A Real-time Garbage Collector with Low Overhead and Consistent Utilization

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. We present a mostly non-moving, dynamically defragmenting collector that overcomes both of these limitations: by avoiding copying in most cases, space requirements are kept low; and by fully incrementalizing the collector we are able to meet real-time bounds. We implemented our algorithm in the Jikes RVM and show that at real-time resolution we are able to obtain mutator utilization rates of 45% with only 1.6–2.5 times the actual space required by the application, a factor of 4 improvement in utilization over the best previously published results. Defragmentation causes no more than 4% of the traced data to be copied.

General Terms

Algorithms, Languages, Measurement, Performance

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

Read barrier, defragmentation, real-time scheduling, utilization

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 [9]) 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.

We present a design for a real-time garbage collector for Java, an analysis of its real-time properties, and implementation results that show that we are able to run applications with high mutator utilization and low variance in pause times.

The target is uniprocessor embedded systems. The collector is therefore concurrent, but not parallel. This choice both complicates and simplifies the design: the design is complicated by the fact that the collector must be interleaved with the mutators, instead of being able to run on a separate processor; the design is simplified since the programming model is sequentially consistent.

Previous incremental collectors either attempt to avoid overhead and complexity by using a non-copying approach (and are therefore subject to potentially unbounded fragmentation), or attempt to prevent fragmentation by performing concurrent copying (and therefore require a minimum of a factor of two overhead in space, as well as requiring barriers on reads and/or writes, which are costly and tend to make response time unpredictable).

Our collector is unique in that it occupies an under-explored portion of the design space for real-time incremental collectors: it is a *mostly non-copying* hybrid. As long as space is available, it acts like a non-copying collector, with the consequent advantages. When space becomes scarce, it performs defragmentation with limited copying of objects. We show experimentally that such a design is able to achieve low space and time overhead, and high and consistent mutator CPU utilization.

In order to achieve high performance with a copying collector, we have developed optimization techniques for the Brooks-style read barrier [10] using an "eager invariant" that keeps read barrier overhead to 4%, an order of magnitude faster than previous software read barriers.

Our collector can use either time- or work-based scheduling. Most previous work on real-time garbage collection, starting with Baker's algorithm [5], has used work-based scheduling. We show both analytically and experimentally that time-based scheduling is superior, particularly at the short intervals that are typically of interest in real-time systems. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The paper is organized as follows: Section 2 describes previ-

Reprinted from the proceedings of POPL'02 with permission.

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.

ous approaches to real-time collection and some of the common problems encountered. Section 3 presents an informal overview of our collector. Section 4 analyzes the conditions under which real-time bounds can be met. Section 5 analyzes the space requirements of our collector and compares them to other real-time collectors. Section 6 describes the implementation of the collector, and Section 7 presents our experimental results. Section 8 discusses issues in real-time garbage collection that are raised by our work. Finally, we present our conclusions.

2. PROBLEMS WITH PREVIOUS WORK

Previous approaches to real-time garbage collection have generally suffered from a variety of problems. In this section we will describe these problems.

2.1 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 [6] also only handles a single object size.

Johnstone [17] 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. [14] 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 [23] 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.

2.2 High Space Overhead

To avoid the problems resulting from fragmentation, many researchers have used copying algorithms [5, 10] as the basis for real-time collection. Such collectors typically have a high space overhead. First of all, when a full collection is performed a complete semi-space is required for the target data, so the minimum space overhead is a factor of 2. Secondly, space is required so that the mutator can continue to run (and allocate) while the collector operates. In order to achieve good mutator utilization while the collector is running, a space overhead of a factor of 3-5 is typical [12].

For Johnstone's non-copying collector [17], space overhead is often a factor of 6–8.

2.3 Uneven Mutator Utilization

Much of the literature has focused on maximum pause times induced by collection, but in fact an equally important metric is mutator utilization (the fraction of the processor devoted to mutator execution). If there is a period of low utilization, the mutator may be unable to meet its real-time requirements even though all individual pause times are short.

Uneven utilization is endemic to collectors that use a to-space invariant (that is, the mutator only sees objects in to-space). Such collectors are implemented with a read-barrier that checks if an object being accessed is in from-space, and if so, copies it into to-space before returning the pointer to the mutator. There is therefore a tight coupling between the operations of the mutator and the scheduling of operations by the collector.

Examples are Baker's copying algorithm [5] which uses an explicit read-barrier, and the Appel-Ellis-Li collector [2], which uses virtual memory protection. Both of these collectors have the property that mutator utilization is very poor right after the collector starts, when the "fault rate" is high.

An alternative is to use a replicating collector which maintains a from-space invariant, and to perform mutator updates on both from-space and to-space, as in the ML collectors of Nettles and O'Toole [21] and Cheng and Blelloch [12]. However, this requires a fairly costly replication for all updates, rather than a simple write barrier on pointer updates. As a result, the strategy is better suited to mostly functional languages like ML, and less well-suited to imperative languages like Java.

2.4 Inability to Handle Large Data Structures

Some algorithms attempt to avoid the factor of 2 space overhead in copying collectors by doing the work incrementally — collecting only a portion of the heap at a time. The most notable example is the Train algorithm [16]. Recently, Ben-Yitzhak et al. [7] have implemented a parallel incremental collector that operates on a fixed fraction of the heap at a time to minimize pause times for large heaps.

The fundamental problem with all algorithms that attempt to collect a subset of the heap at a time is that they can be defeated by adversarial mutators. Large cyclic structures, objects with high in-degree, and high mutation rates are ways to force such collectors to perform work without fixed bound.

3. OVERVIEW

Our collector is an incremental uni-processor collector targeted at embedded systems. It overcomes the problems of the previous section by using 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.

3.1 Overview 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 [10]. 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 [24] 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 avoid 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.

3.2 Object Allocation and Fragmentation

Allocation is performed using a simple segregated free-list approach. When a free list is empty, a new page is chosen, broken into equal-size blocks, and the resulting blocks are placed onto that free list. Note that the allocator page size Π is not necessarily the same as the operating system page size. We use $\Pi = 2^{14} = 16$ KB.

Internal fragmentation is regulated by using a geometric progression of free list sizes, such that if there is a free list whose blocks are of size s, the next larger size is $s(1+\rho)$. We generally choose $\rho = 1/8$, resulting in worst-case fragmentation of 12.5%. However, the measured internal fragmentation in our collector has never exceeded 2% at $\rho = 1/8$.

Most programs obey a "locality of size" property, that is, the object sizes allocated frequently in the past will tend to have a high correlation with object sizes allocated in the future. Therefore, we expect that in the normal case, the garbage collector will find unused blocks in a particular size class that can simply be re-used. Only in relatively rare cases will object allocation cause external fragmentation.

Because our collector performs defragmentation, we can choose a ρ that results in low internal fragmentation, but allows a relatively large number of size classes. Most collectors based on segregated lists must be concerned about external fragmentation, and therefore keep the number of size classes small by choosing $\rho = 1$, leading to power-of-two size classes with high internal fragmentation.

The only overhead to decreasing ρ is that we may need to have one under-utilized page per size class. Assuming a 4-byte word size, the number of size classes C is bounded by

$$C \leq \frac{\ln \Pi\rho/4}{\ln(1+\rho)}$$

The free lists are actually kept as chains of pages, rather than chains of blocks. Each page has an associated mark array. The allocation cursor is actually a pair pointing to a page and a block within the page. This organization allows formatting of pages to be performed lazily and therefore avoids a full sweep of the memory on each collection.

3.3 Defragmentation

At the end of the sweep phase, we determine whether there is a sufficient number of free pages to allow the mutator to continue to execute for another collection cycle without running out of memory, assuming a worst-case selection of object sizes by the mutator (that is, we assume that the mutator will act adversarially to maximize external fragmentation).

If the number of free pages drops below this threshold, we perform a defragmentation that will free at least that many pages. Defragmentation is performed as follows: for each page, we compute the number of live objects. Then the pages within each size class are sorted by occupancy. Finally, we move objects from the least occupied to the most occupied pages within a list (note that this never causes new pages to be allocated; it only transfers objects between pages within a size class).

3.4 Read Barrier

We use a Brooks-style read barrier [10] to maintain a to-space invariant in the mutator: 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, but the sets comprising from-space and to-space have a large intersection, rather than being completely disjoint as in a pure copying collector.

Note that while we use a read barrier and a to-space invariant, our collector does not suffer from variations in mutator utilization because 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 will show that this is not the case when they are implemented carefully in an optimizing compiler and the compiler is able to optimize the barriers.

A fundamental design choice for the read barrier is whether it is "lazy" or "eager". A lazy barrier has the property that registers and stack cells can point to either from-space or to-space objects, and the forwarding operation is performed at the time of use.

An eager barrier, on the other hand, maintains the invariant that registers and stack cells always point into to-space: the forwarding operation is performed eagerly as soon as the quantity is loaded. Eager barriers have a major performance advantage in that if a quantity is loaded and then dereferenced many times (for instance, a reference to an array of integers loaded and then used in a loop), the eager barrier will only perform the forwarding operation once, while the lazy barrier will perform the forwarding operation for every array access.

Of course, there is a cost: because the eager invariant is more strict, it is more complex to maintain. Whenever the collector moves objects, it must find all outstanding register and stack cells and re-execute the forwarding operation on them.

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 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 a mean cost of only 4% for the read barriers, as is shown in Section 7.

3.5 Arraylets

Large objects pose special problems for garbage collectors. In copying collectors, if they are repeatedly copied, the performance penalty can be very high. In non-copying collectors, external fragmentation can make it impossible to allocate a large object. For instance, a single small object in the middle of the heap can make it impossible to satisfy a request for an object slightly larger than half the heap.

Furthermore, in incremental and real-time collectors, large objects pose an additional problem because they can not be moved in a reasonably bounded amount of time.

Siebert [23] has suggested using fixed-size blocks of 32 or 64 bytes for all object allocations, and creating large arrays by using a tree structure. Unfortunately, this requires rewriting every array access as a loop, and can have a severe performance penalty for array-intensive programs since common loop optimizations are defeated.

Our mostly non-copying collector allows a different approach: we represent small arrays contiguously, and large arrays as two-level structures consisting of a sequence of *arraylets*. Each arraylet (except the last) is of a fixed size, which is chosen to be a power of two so that the division operation required for indexing can be implemented with a shift. In our case, arraylets are $\Sigma = \Pi\rho = 2$ KB.

We therefore have the advantage of never needing to allocate large objects contiguously, and are therefore not subject to external fragmentation. On the other hand, access to array elements is still efficient, and when combined with strip-mining optimizations is usually as efficient as contiguous layout.

The arraylet size must be chosen carefully and there are some tradeoffs involved. With a sufficiently large size, one can assume that all objects will be contiguous and smaller than the arraylet size, simplifying the implementation. The maximum array size that can be represented with a single root of size Σ is $\Sigma^2/4$, or 1 MB in our case. However, note that if necessary we can simply allocate an entire block to be the root of the array, because the wasted space at the end of the block will be negligible compared to the total size of the array. Thus we can accommodate arrays of size up to $\Pi\Sigma/4$ or 8 MB. For larger objects we can scan the free block list for the necessary number of contiguous free blocks. If the system must be able to return objects larger than 8 MB in real time, the maximum size can be tuned by varying Π and ρ.

All arrays are represented in a uniform manner: arraylet pointers are laid out in reverse order to the left of the array header. If the array is contiguous, there is only one arraylet pointer and it points to the data field to the right of the header.

Arraylets are implemented in the system presented in this paper, but not yet highly optimized. However, we can use Arnold's thin guards [3] to eliminate the indirection for array types that do not exist as arraylets, so that most array accesses will operate at full speed. For arraylets, we can strip-mine regular iterations to the arraylet size. Thus arraylets should only suffer performance penalties when they are used *and* when the access pattern is irregular.

3.6 Open Issues

The main issue we have not addressed in our collector is making stack processing incremental. This is an issue in two parts of the system: root scanning and maintenance of the eager invariant for the read barrier.

Stacklets [13] break stacks into fixed-size chunks to quantize the associated work. However, they only provide a partial solution: if we only copy the top stacklet of the running thread and return to the mutator, the mutator can then begin either pushing or popping at a very high rate.

A high rate of popping is problematic because the collector must halt the mutator while it copies each popped stacklet, and if many stacklets are popped in a short interval the mutator utilization will temporarily become very low. It can also force the memory consumption of the stack to double (due to the snapshots).

A high rate of pushing is problematic because the collector may have trouble keeping up with the mutator. In this case, the solution is to model stack pushes that enter new stacklets to be modelled as allocation, and to use the associated methods for measuring and controlling allocation rates.

For the benchmarks available to us the stacks remained small, and the limiting factor in pause time was the resolution of the operating system clock. Therefore the implementation presented in this paper does not include stacklets. We intend to address the issue of incrementalizing stack operations in future work, in particular by exploring alternative write barriers and termination conditions.

4. REAL-TIME SCHEDULING

In this section we derive the equations for CPU utilization and memory usage for our collector using two different scheduling policies: one based on time, the other based on work.

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

- $A^\star(\tau)$ is the instantaneous memory allocation rate at time τ (MB/s).
- $G^\star(\tau)$ is the instantaneous garbage generation rate at time τ (MB/s).
- P 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.

By convention, upper-case letters refer to primitive quantities; lower-case quantities are derived. The only other primitive parameters required are the relative rates of mutator and collector.

From these basic parameters we can define a number of important characteristics of the application relevant to real-time garbage collection.

The amount of memory allocated and garbage generated during the interval (τ_1, τ_2) are

$$\alpha^\star(\tau_1, \tau_2) = \int_{\tau_1}^{\tau_2} A^\star(\tau) d\tau \tag{1}$$

$$\gamma^\star(\tau_1, \tau_2) = \int_{\tau_1}^{\tau_2} G^\star(\tau) d\tau \tag{2}$$

The maximum memory allocation for an interval of size $\Delta\tau$ is

$$\alpha^\star(\Delta\tau) = \max_{\tau} \alpha^\star(\tau, \tau + \Delta\tau) \tag{3}$$

and the maximum memory allocation *rate* is

$$a^\star(\Delta\tau) = \alpha^\star(\Delta\tau)/\Delta\tau \tag{4}$$

The instantaneous memory requirement of the program (excluding garbage, overhead, and fragmentation) at time τ is

$$m^\star(\tau) = \alpha^\star(0, \tau) - \gamma^\star(0, \tau) \tag{5}$$

4.1 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

$$m(t) = m^\star(\Phi(t)) \tag{6}$$

and the maximum memory requirement over the entire program execution is

$$m = \max_t m(t) = \max_\tau m^\star(\tau) \tag{7}$$

4.2 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:

- 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. In Section 7 we will show how close we are able to get to this ideal in practice.

Cheng and Blelloch [12] 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

$$u_T(\Delta t) = \frac{Q_T \cdot \left\lfloor \frac{\Delta t}{Q_T + C_T} \right\rfloor + x}{\Delta t} \tag{8}$$

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

$$x = \max\left(0, \Delta t - (Q_T + C_T) \cdot \left\lfloor \frac{\Delta t}{Q_T + C_T} \right\rfloor - C_T\right) \tag{9}$$

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

$$\lim_{\Delta t \to \infty} u_T(\Delta t) = \frac{Q_T}{Q_T + C_T} \tag{10}$$

A plot of the MMU for a perfectly scheduled system using 10 millisecond mutator and collector quanta is shown in Figure 1. It is important to note that at the small time scales of interest in real-time systems, the x term is very significant: at $\Delta t = 20$ ms the MMU is $1/2$ (the maximum value), while at $\Delta t = 30$ ms, it drops to $1/3$. Also, the higher the scheduling frequency of the collector, the more quickly it converges to the theoretical limit.

In practice, at large time intervals $u_T(\Delta t)$ is only a lower bound on the utilization, since in most cases the collector only runs intermittently.

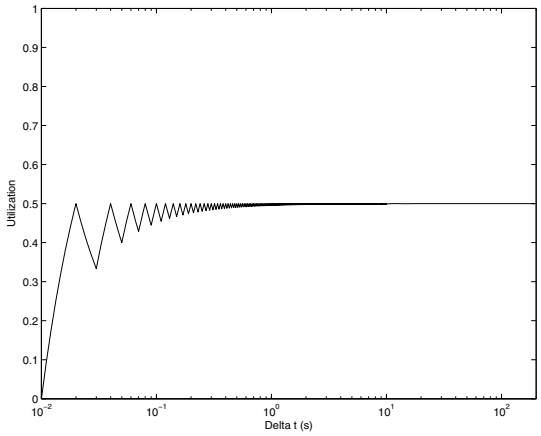

Figure 1: MMU for a perfectly scheduled time-based collector. $Q_T = C_T = 0.01$ (10 ms).

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 $m(t)/P$ 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 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

$$e_T(t) = \alpha^\star\left(\Phi(t), \Phi(t) + \frac{m(t)}{P} \cdot \frac{Q_T}{C_T}\right) \tag{11}$$

We further define the maximum excess space required as

$$e_T = \max_t e_T(t) \tag{12}$$

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; 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

$$s_T(t) \leq m(t) + 3e_T \tag{13}$$

and the overall space requirement is

$$s_T \leq m + 3e_T \tag{14}$$

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

4.3 Work-Based Scheduling

Work-based scheduling interleaves the collector with the mutator based on fixed amounts of allocation and collection. A work-based real-time collector is parameterized by

- Q_W is the work-based mutator quantum: the number of MB the mutator is allowed to allocate before the collector is allowed to run.
- C_W is the work-based collector quantum: the number of MB the collector must process each time the mutator yields to it.

Then the excess space required to perform a collection at time t is

$$e_W(t) = m(t) \cdot \frac{Q_W}{C_W} \tag{15}$$

and the excess space required for a collection over the entire execution is

$$e_W = m \cdot \frac{Q_W}{C_W} \tag{16}$$

Note therefore that it must be the case that $Q_W < C_W$ or else the space may grow without bound.

Consequently, the space requirement of the program at time t is

$$s_W(t) \leq m(t) + 3e_W \tag{17}$$

and the space requirement for the entire program execution is

$$s_W = m + 3e_W \tag{18}$$

4.3.1 Work-based CPU Utilization

Computing mutator CPU utilization when collector scheduling is work-based is inherently problematic, because the operations of the mutator may affect the amount of time allocated to the mutator. In other words, there is a time dilation from t to τ that is linear and fixed in time-based scheduling but variable, non-linear, and application-dependent in work-based scheduling.

Due to these problems it is not possible to obtain a closed-form solution for the utilization. We begin by noting that each mutator pause involves the collector processing C_W memory at rate P. Hence each mutator pause will be $d = C_W/P$. In our simplified model, this will be a constant. Each mutator quantum will involve allocation of Q_W memory, so the minimum total mutator time $\Delta\tau_i$ for i quanta will be given by the minimum $\Delta\tau_i$ that solves the equation

$$\alpha^\star(\Delta\tau_i) = iQ_W \tag{19}$$

As the time interval increases the maximum amount of allocation in that time does not decrease, so $\alpha^\star(\Delta\tau)$ is a monotonically increasing function and hence $\Delta\tau_i > \Delta\tau_{i-1}$. Therefore, the solution to (19) can be found with an iterative method. This is analogous to the iterative solution to rate monotonic scheduling in real-time systems [18].

Let k be the largest integer such that

$$kd + \Delta\tau_k \leq \Delta t \tag{20}$$

so the minimum mutator utilization over an interval of size Δt is

$$u_W(\Delta t) = \frac{\Delta\tau_k + y}{\Delta t} \tag{21}$$

where the first term in the numerator is the time taken by k whole mutator quanta in the interval and the y term corresponds to the size of the remaining partial mutator quantum (if any), which is defined as

$$y = \max(0, \Delta t - \Delta\tau_k - (k+1) \cdot d) \tag{22}$$

Note that in a work-based collector, utilization will be zero for $\Delta t < d$. In fact, any large allocation of nQ_W bytes will lead to zero utilization for time nd. This simply expresses analytically the fact that in a work-based collector, there is a much larger burden on the programmer to achieve real-time bounds by making sure that memory allocation is sufficiently discretized and evenly spaced.

4.4 Mutation

In addition to allocation, the other form of work by the mutator that can interact with the operation of the collector is the actual heap mutation. Mutation can be thought of as an alternate way for roots to be added, along with stack scanning.

We impose the following division of labor between the mutator and the collector: the mutator's write barrier is responsible for making sure that only non-null, unmarked objects are placed into the write buffer. This ensures that the work performed by the collector attributable to mutation is $O(N)$, where N is the number of objects, while keeping the overhead of the write barrier constant.

The collector periodically processes the write buffer and treats the entries like any other potential roots: it marks the objects gray and places them into the work queue for scanning.

Note that in the worst case, the work queue can reach size N.

Now we must account for mutation in the formulas for collector performance that we have derived, because mutation consumes memory just like allocation by the mutator. To do this, we simply redefine $A^\star(\Delta\tau)$ to comprise both directly allocated memory and indirectly allocated memory due to mutation, where each mutation consumes memory of the size of one object pointer. If desired, the formulas could all be broken up to account for each kind of space consumption individually.

4.5 Sensitivity to Parameters

The degree to which each collector is able to meet its predicted behavior will depend quite strongly on the accuracy of the parameters which are used to describe the application and the collector strategy. These are the application parameters $A^\star(t)$ and $G^\star(t)$, and the collector parameters, P and either Q_T and C_T or Q_W and C_W for the time-based or work-based collectors, respectively.

In practice, the user describes the application in terms of its maximum memory consumption m and its maximum allocation rate $a^\star(\Delta\tau)$.

4.5.1 Sensitivity of the Time-based Collector

The CPU utilization rate u_T of the time-based collector is strictly dependent on the quantization parameters Q_T and C_T, so the utilization will be very steady (depending only on implementation-induced jitter, and subject to the minimum quantization that the implementation can support).

On the other hand, the space required to perform a collection $e_T(t)$ which determines the total space s_T required to run the application is dependent on both the maximum memory usage by the application and the amount of memory allocated over an interval. Thus if the user under-estimates either m or $a^\star$, then the total space requirement s_T may grow arbitrarily. In particular, time-based collectors are subject to such behavior when there are intervals of time in which the allocation rate is very high. Furthermore, the estimate of the collector processing rate P must also be a lower bound on the actual rate.

However, note that the space consumed by the application is over a relatively long interval of time, namely the amount of time the application runs while a single collection takes place or

$$\Delta\tau = \frac{m(t)}{P} \cdot \frac{Q_T}{C_T}$$

and therefore the allocation rate in that time will typically be close to the average allocation rate of the program and the variation will tend to be low.

Therefore, to a first order, a time-scheduled collector will meet both its time and space bounds as long as the user estimate of m is correct.

4.5.2 Sensitivity of the Work-based Collector

In the work-based collector, the space overhead for collection

$e_W(t)$ is straightforward to compute, and it will be accurate as long as the user estimate of the total live memory m is accurate.

On the other hand, the CPU utilization rate for a given interval Δt depends on the allocation rate $a^\star(\Delta\tau)$ where $\Delta\tau \leq \Delta t$ as well as on the collector processing rate P.

The interval Δt is the interval over which we require real-time performance, for instance $20ms$. Since this interval is small, the peak allocation rate for this interval size is likely to be quite high, as we will show in Section 7. Thus we expect that the CPU utilization of the work-based collector will vary considerably with the allocation rate.

In particular, note that the $\Delta\tau$ in which the time-based collector is dependent on allocation rate is on a much larger scale, namely the amount of time for a garbage collection.

Therefore, to a first order a work-scheduled collector will meet its space bound as long as the user estimate of m is correct, but its CPU utilization will be heavily dependent on the allocation rate over a real-time interval.

4.5.3 A Robust Collector

A robust real-time collector should primarily use a time-based scheduling policy, but as memory resources become scarce (indicating that the input parameters to the collector may have been incorrect), if graceful degradation is desirable then the collector should begin slowing down the allocation rate.

This can be done in a number of ways. A classical approach in real-time systems is to separate threads into priority classes, and as the system becomes unable to meet real-time bounds, low-priority threads are successively suspended [15].

Another approach is to begin using a hybrid strategy which becomes progressively more work-based as the collector comes closer to its memory limit. This approach will not guarantee that real-time bounds are met, but is robust even if the allocation rate and memory utilization of the top-priority threads have been underestimated.

We have not done this; instead we have implemented pure time-based and work-based collector scheduling policies, and in Section 7 we compare them experimentally so that the tradeoffs can be evaluated.

5. SPACE COSTS

We now compare the relative space costs of the different types of real-time collectors. Since purely non-copying algorithms are subject to high (and often unbounded) fragmentation, they are not suitable for use in true real-time systems.

Since our collector has a significantly different architecture from copying real-time collectors, its space bounds are quite different.

Incremental semi-space copying collectors have an inherent space overhead of $2 \cdot (m + e) + f + g$, where m is the maximum live heap memory, e is the space required to allow allocation to proceed during a single garbage collection, f is the maximum stack depth, and g is the maximum size of the global variable area.

Our collector has an expected-case space requirement of $m+e+f+g$ and a worst-case cost of $m+3e+f+g+N$, where N is the maximum number of uncollected objects (live or dead). The extra $2e+N$ space is incurred when: a data structure of size close to m is freed immediately after the beginning of a collection (the collector must run again to find it, requiring e extra space); all garbage found causes external fragmentation (requiring an extra collection cycle to relocate the data and make it available, which requires another e extra space); and the program traverses the heap in a pessimal fashion (forcing a maximum number of pointers to be pushed onto the work queue for each mark operation, which requires N extra words of memory).

There are two things to note about the worst-case memory requirements of our collector. First, the difference in the worst-case between our collector and a copying collector is $e + N$ versus m. The space e required to run a collection is typically lower than the maximum live memory m (and can be tuned). The maximum number of uncollected objects is the maximum uncollected space divided by the average object size in words A, or $(m + e)/A$. Since A is typically on the order of 8 for Java programs, N is typically small relative to m. Thus for most programs, the worst-case space requirements of our collector will still be smaller than those of a copying semi-space collector.

Second, the likelihood of more than one of these worst-case scenarios occurring concurrently is very low. In practice, this means that the amount of memory devoted to the system can be varied between the expected- and worst-case space requirements depending on the acceptable failure rates for the system in question.

These figures do not include the extra space overhead required to bound internal fragmentation with the parameter ρ, which we have set to $1/8$ in our implementation. This parameter can be further reduced at the expense of potentially requiring additional partially used blocks for the extra size classes. For $\rho = 1/8$, the number of size classes $C = 45$ and the measured fragmentation does not exceed 2% for our benchmarks.

We do *not* include the space overhead due to the forwarding pointer, since all high-performance copying algorithms also use a forwarding pointer. Bacon et al. [4] have shown that an extra header word leads to a 14% increase in space utilization (assuming one uses an object model with a single-word header as a basis).

6. IMPLEMENTATION ISSUES

We implemented a real-time collector based on the ideas introduced in the previous sections. Implementing the collector required both coding the collector proper as well as adding read barriers to the compiler. In certain cases, it was infeasible to introduce a read barrier. Omitting the barrier is correct as long as we pin the object to guarantee that it never moves. Fortunately, most objects that fall into this category are run-time data structures that are immortal. By maintaining a separate immortal heap, we can omit moving such objects without introducing any fragmentation.

6.1 Triggering a Collection

In the worst-case analysis of the collector, we can run the program in space $m + 3e$ where m is the amount of maximum live data and e is the space required to run a single collection ($e = e_T$ or e_W depending on the scheduling policy). However, executing with these boundary conditions will result in the collector always running. Even if the application utilization is at 50% during a collection, this will lead to an overall slowdown of the program by a factor of 2 which is likely unacceptable. For comparison, running a stop-the-world collector at m will result in a virtually infinite slowdown. The solution is to provide headroom so that the program can run for some time before a collection must occur. For example, if enough headroom is provided so that the collector runs only 25% of the time, then the overall utilization rises to 87.5%.

In our implementation, we have set the headroom to be e. A collection is thus triggered when the amount of memory in use is $m + e$.

6.2 Control of Interleaving

Ideally, in the time-scheduled collector we would use a precise timer to control the scheduling of the mutator and collector processes. Unfortunately, AIX does not allow user-level access to timers with a resolution of less than 10 ms. Therefore, we must

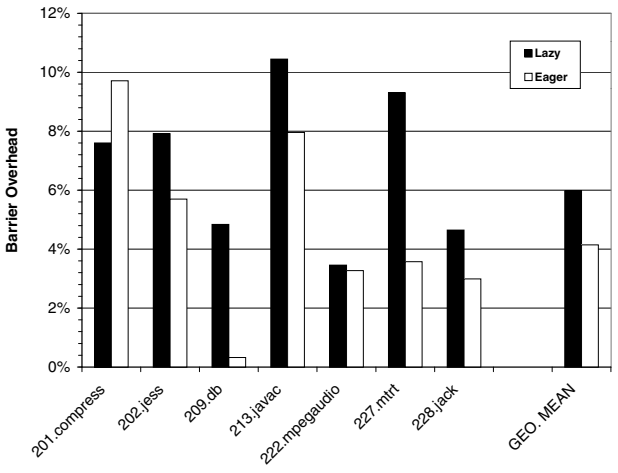

Figure 2: Relative overhead of lazy and eager read barriers in the Jikes RVM optimizing compiler.

use an approximate method based on polling.

The mutator polls the timer on the slow path of allocation (when it moves to a new page) or when the mutation buffer fills up. This keeps the polling out of the fast, in-lined cases, but is subject to some inaccuracy. However, as a practical matter, this is acceptable because we are increasing mutator utilization and doing it at a time when resource consumption is low.

The collector on the other hand performs work in progressively finer work quanta as it gets closer to the end of its time quantum C_T. When the time consumed is close to or exceeds the quantum, the mutator is resumed.

The work-scheduled collector is also subject to some inaccuracy because scheduling is only performed in the slow path through the allocator, even though a precise count of bytes allocated is kept on the fast (inlined) path.

7. MEASUREMENTS

We present empirical results in this section. All results were obtained on an IBM RS/6000 Enterprise Server F80 running AIX 5.1. The machine has 4 GB of main memory and six 500 MHz PowerPC RS64 III processors each with 4 MB of L2 cache.

The virtual machine runs on a single CPU. Experiments are run on an unloaded multiprocessor so that operating system processes are performed on different CPUs to avoid perturbing our measurements.

Our system is implemented as part of the Jikes Research Virtual Machine (RVM) version 2.1.1 at the IBM T.J. Watson Research Center [1]. All methods were compiled with the optimizing compiler (since the system is real-time, adaptive compilation is turned off). Measurements were started after a dummy run of the benchmark which forces all methods to be compiled.

Because the optimizing compiler often requires more space than the applications themselves, the heap is resized after compilation to the heap sizes given. In this way, we measure the intrinsic properties of the application rather than of the compilation.

7.1 Read Barrier Costs

Since our collector makes use of read barriers, and read barriers are often considered prohibitively expensive, we begin by showing that our optimized implementation of the Brooks-style read barrier with the eager invariant can achieve very low overhead.

We have implemented both lazy and eager barriers in the IBM Jikes RVM [1] and present their relative performance, both to each other and to a system without barriers.

Read barriers were initially considered so expensive as to only be practical with hardware support, as was done in a number of commercially available machines such as the Symbolics Lisp Machine [20].

The first implementation we know of the Brooks read barrier is that by North and Reppy [22] in their concurrent collector for Pegasus ML. However, they do not measure barrier cost but only the total cost.

Zorn [25] compared the cost of hardware, software, and page protection-based read barriers, and determined that software read barriers were much better than protection based read barriers, but still cost about 20%.

Zorn measured Baker-style read barriers that require on average four ALU/branch instructions. The straightforward implementation of our read barrier requires a compare, a branch, and a load. However, in most cases we are able to optimize away the compare and branch, and to perform common subexpression elimination on the remaining loads.

The results are shown in Figure 2. The geometric mean of the lazy barrier overhead is 6%, with a maximum of 11% overhead for `javac`. This is significantly better than previous results, but still not acceptable in our opinion.

On the other hand, the geometric mean of the eager barrier overhead is only 4%, with a maximum of less than 10% for `compress`. The mean overhead is an order of magnitude better than previous results, and in our opinion low enough for incorporation into a highly aggressive optimizing compiler, given the potential benefits in space utilization and incrementality, as shown in the following sections.

On the other hand, the variance is still too large: we do not consider the slowdown for `compress` to be acceptable. It turns out that the problem with compress is due to a shortcoming in the optimizer which is preventing it from performing loop-invariant code motion. Once this bug is fixed, we expect the overhead in `compress` to drop below 5%.

7.2 Collector Performance

We tested our real-time collector on the SPECjvm98 benchmarks and a synthetic `fragger` benchmark designed to act adversarially: it allocates at a high rate, uses a maximal amount of memory, and creates maximal fragmentation.

Of the SPEC benchmarks, `mpegaudio` was excluded because it performed little allocation and would have no necessary garbage collections. In addition, `compress` was excluded because our current implementation does not fully support arraylets and `compress` makes frequent use of large arrays.

Table 1 presents overall results for the benchmarks when run with a target utilization $u_T(22.2\text{ms}) = 0.45$, mutator quantum $Q_T = 10$ ms, and a collector quantum $C_T = 12.2$ ms. For each program, we include the high watermark of live data and the maximum memory actually used. The average allocation rate is the allocation rate over the entire execution $a^*(T)$ whereas peak allocation measures the maximum allocation rate during a mutator quantum $a^*(Q_T)$.

The collection rate P shows how quickly the collector can trace through the live data of that application. For each program, we show the target application utilization and the worst actual utilization that occurred. Average and maximum pause times are included. Finally, we show the total amount of moved and traced data as an indication of how much defragmenting work is necessary.

Benchmark	Maximum Memory			Allocation Rate		Coll. Rate	Min. Util.	Pause Time		Copied	Traced
	Live m	Used s_T	Ratio s_T/m	Avg. $a^\star(T)$	Peak $a^\star(Q_T)$	P	$u_T(\Delta t)$	Avg.	Max.		
javac	34	69.3	2.0	14.2	258.0	39.4	0.446	11.3	12.3	12.1	299.4
jess	21	52.4	2.5	19.2	94.2	53.2	0.441	11.0	12.4	2.0	324.0
jack	30	59.3	2.0	16.0	105.1	57.4	0.441	10.7	12.4	3.5	321.7
mtrt	28	44.4	1.6	9.6	114.3	45.1	0.446	11.0	12.3	2.3	176.9
db	30	54.8	1.8	14.2	82.1	36.7	0.441	11.4	12.4	1.3	144.6
fragger	20	47.7	2.4	17.5	185.9	38.4	0.441	11.0	12.4	12.6	307.0

Table 1: Overall Results for the Time-Based Collector. T is the total run-time of the program. Target mutator quantum $Q_T = 10$ ms, target collector quantum $C_T = 12.2$ ms, target utilization is 0.45, $\Delta t = 22.2$ ms. All sizes in MB, all rates in MB/s, all times in milliseconds.

All of our benchmarks had a similar amount of maximum live data (between 20 and 30 MB) but they required anywhere from 45 to 70 MB at some point in their execution. The variance in space usage arises from several factors. The heap size requirement appears to be primarily correlated to the average allocation rate — for instance note the high allocation rate for `jess` and the correspondingly high maximum memory ratio.

The measured values for the rate of collection P range from 36.7 to 57.4 MB/s. This is primarily due to variation in pointer density in the data structures of the programs, and shows that while our theoretical assumption that P is constant does not introduce large error, it is nonetheless significant.

Average allocation rates ranged from 9.6 to 19.2 MB/s while peak allocation rates ranged from 82.1 to 258 MB/s. These spikes in allocation rates demonstrate the infeasibility of using a purely work-based scheduling policy for the goal of maintaining a high minimum utilization.

For all benchmarks, we ran the collector with a target application utilization of 0.45 and obtained a minimum utilization of 0.441 to 0.446. Thus the maximum deviation is only 2%.

The last two columns in Table 1 show the amount of data copied and traced over the entire execution of the program. The maximum amount of data copied is about 4% of the data traced (interestingly, `javac` introduces about the same amount of fragmentation as `fragger`, which we wrote specifically as a fragmenting adversary program). Note that the amount of data traced by our collector is roughly comparable to the amount of data that would be copied by a semi-space collector, although such a collector would require a significantly larger heap to obtain the same performance.

Table 2 summarizes the results when we changed from time- to work-based collector scheduling. The table only shows those quantities that changed appreciably from the time-based collector. Also, since the utilization at our target Δt was often zero, we also give the utilization for an interval of 50 ms. Even at this longer interval, the best case is only half the target value.

While average pause times are considerably lower, the maximum pause times for the work-based collector are much higher (up to 92 ms for `fragger`) and at $\Delta t = 22.2$ ms the minimum mutator utilization is very poor. These measurements confirm experimentally the analytic results from Section 4.

7.3 Detailed Evaluation

We examine three benchmarks in detail: `mtrt`, `javac`, and `fragger`. These three were chosen because they represent a range of difficulty for the collector. For both time- and work-based scheduling, we compare the distribution of pause times, the utilization over time, the MMU over a full range of intervals, and the space consumption of these three benchmarks.

The pause time distributions are shown in Figures 3 through 8. These figures show that our time-based collector achieves highly uniform pause times, with the majority of all pauses at 12.2 ms. By comparison, the work-based collector has a much more uneven distribution (note the differences in scale on both the x and y axes). The work-based collector has considerably shorter average pauses, but the distribution is much more uneven and there is a much longer "tail" in the distribution.

The adversarial nature of `fragger` is clearly seen in Figure 8: while the work-based collector keeps the vast majority of pauses below 10 ms, the tail extends to almost 100 ms.

If one only considered maximum pause time, the pause time distribution graphs would give the impression that utilization under the work-based collector would be about 2-3 times worse for non-adversarial programs. However, Figures 9 through 14 show that for a short interval on the order likely to be of interest in real-time systems (22.2 ms), work-based scheduling produces very large variance in mutator utilization, often dropping to almost zero. This can easily occur when a single large object is allocated, forcing the collector to perform many collector quanta in a row.

On the other hand, the time-based collector performs extremely well. There is a small amount of jitter due to slight imprecision in our work predictor, but utilization during collection is almost exactly on or above the target.

For `mtrt` and `javac`, after the first collection the application enters a fairly regular cycle in which the concurrent collector is off for $1/3$ to $1/2$ of the time. However, the adversarial nature of `fragger` is once again apparent: in the time-based collector, it collects continuously, and in the work-based collector the utilization frequently drops to zero.

Figures 15 through 17 show the minimum mutator utilization (MMU [12]) of both time- and work-based collectors superimposed on one graph. The time scale ranges from 10 milliseconds to the length of the program run. At small time scales, the MMU for the time-based collector almost precisely matches the shape of the perfect curve shown in Figure 1. At larger time scales, the effect of the mutator being off come into play, and utilization rises above the target.

MMU for the work-based collector is much lower, and interestingly has much less of a "sawtooth" shape. At the time scale of a small number of collections, the work-based collector may briefly exceed the time-based collector in utilization, but as the number of collections becomes large they appear to approach the same asymptotic cost.

We compute the MMU precisely below $\Delta t = 10$ seconds using a quadratic algorithm; above 10 seconds we use an approximate algorithm which has very small error. Cheng and Belloch [12] used

Benchmark	Minimum Utilization		Pause Time	
	$u_W(\Delta t)$	$u_W(50\text{ms})$	Avg.	Max.
javac	0	0.118	5.1	31.9
jess	0	0.180	3.1	26.2
jack	0.001	0.152	2.8	13.2
mtrt	0.002	0.227	5.0	18.8
db	0	0.141	6.0	28.0
fragger	0	0	3.7	92.1

Table 2: Overall Results for the Work-Based Collector. Mutator allocation quantum $Q_W = 40$ KB, collector processing quantum $C_W = 120$ KB, $\Delta t = 22.2$ ms. All times in milliseconds.

a sampling technique and only plotted the MMU for certain values of Δt, thus hiding some of the irregularity of the curve.

Blackburn et al. [8] use a variant of the MMU which produces a monotonic curve which is strictly derivable from the MMU curve. This definition of utilization is appropriate at the large time scales at which these collectors operate (several hundred milliseconds and above) but hides information that is important at short time intervals of interest in true real-time systems.

Finally, Figures 18 through 20 show space consumption over time for both time- and work-based collectors. The maximum live data and the collector trigger threshold are also shown. What is surprising is how little difference there is between time- and work-based memory consumption given the large differences in behavior seen in the previous graphs. There is some variation, but there is no clear winner: each type of scheduling sometimes requires slightly more or slightly less space, but the shape of the space curves is very similar and only slightly translated.

8. REAL-TIME ISSUES

In Section 2 we outlined some of the problems common to real-time collectors. The design choices made in our collector avoid these problems in the following ways:

- Fragmentation is avoided through a combination of means: internal fragmentation is limited by choosing a small ratio for adjacent size classes. External fragmentation is prevented by defragmenting the heap as needed, and by breaking up large arrays into arraylets.
- Space overhead is limited by using a mostly non-copying algorithm, so that from-space and to-space are mostly sharing physical storage.
- Uneven mutator utilization is avoided because we use a time-based scheduling policy, which is not sensitive to variations in average allocation rate at small (real-time) intervals but only at large intervals on the order of a full collection.
- Large data structures are handled by using arraylets, which effectively turns large objects into small objects.

8.1 Flaws in Baker's Real-time Definition

Baker [5] begins his seminal paper on real-time garbage collection by stating that "*a real-time list processing system is one in which the time required by the elementary list operations ... is bounded by a small constant*." This approach has been the basis for most of the later work on real-time collection [2, 6, 10, 11, 17, 19, 24]. However, this is implicitly a work-based approach, and as we have seen in Sections 4 and 7, at the small time intervals that are typically of interest in real-time systems, work-based collectors may be subject to very poor utilization.

Baker attempts to finesse this problem by interleaving the collector with the mutator in a very fine-grained manner, but this only hides the problem: it keeps individual pauses low, but does not prevent numerous closely-spaced pauses. In the case of Baker's copying collector, the read barrier converts what was originally a simple load instruction into a sequence of tests, loads, and possibly a copy of the object. Let us say that the cost of such a read with barrier is κ times the cost of the original read operation. Then if we consider a short interval Δt containing only read operations, the utilization will be $1/\kappa$.

Ultimately, it comes down to a question of what one means by "small". If $\kappa < 2$, then the utilization will probably be acceptable. However, more typical values are 10 to 20. In such short intervals, utilization may drop so low as to be useless, as we saw experimentally in Table 2.

There are fundamentally three ways to ameliorate this problem: increase Δt, decrease κ, or make κ bimodal. Increasing Δt is dependent on the real-time requirements of the application. An example of decreasing κ is Brooks' variant [10] of Baker's algorithm: a read only requires one extra load instruction, and the costly barrier is only performed on writes, which are considerably less frequent. However, at a resolution of 1 ms, there could be a lot of writes, and the κ for the write barrier is unlikely to be less than 10 (and is often much higher), so utilization could still be very low. Attempts have been made to further reduce the cost of the write barrier by using a store buffer [24] or by pre-allocating the space for the copied object and deferring the actual copy to collection time [15].

Nettles and O'Toole [21] introduced replicating copying collectors [12, 16], which represent another point in the tradeoff space. In these collectors, there is no read barrier, but the overall cost of the write barrier is more expensive because it may have to update both to- and from-space objects.

Baker attempted to keep performance uniform by interleaving the allocator with each CONS, CAR, and CDR operation. However, the more fine-grained the interleaving, the higher the relative cost of the operations. Many subsequent collectors have attempted to reduce the time overhead of concurrent collection by batching the work (the Appel-Ellis-Li collector [2], which uses virtual memory page traps, is an extreme example). However, this limits the resolution of Δt, and does not function well when the cost of the quantized work varies widely (for example, due to variation in object sizes) or when the quanta occur irregularly. If the variation *is* low, it should be possible, for a given Δt, to determine the best batch size analytically.

Ultimately, the distinction that is generally made in the literature between *hard real-time* and *soft real-time* is an over-simplification. There is really a continuum that depends on the required response time and the cost and variability of collector operations.

8.2 Time-based Collectors

While most previous work on real-time collection has focused on work-based scheduling, there are some notable exceptions. In particular, Henriksson [15] implemented a Brooks-style collector [10] in which application processes are divided into two priority levels: for high-priority tasks (which are assumed to be periodic with bounded compute time and allocation requirements), memory is pre-allocated and the system is tailored to allow mutator operations to proceed quickly. For low-priority tasks, no response-time goals are set.

Henriksson gives a schedulability analysis using the real-time scheduling techniques of Joseph and Pandya [18]. While his anal-

ysis is work-based, his formula for utilization is similar to our formula for time-based scheduling. This is because in his collector the high-priority mutators can always interrupt the collector when they are ready to run. Thus we see that interrupt-driven work-based scheduling is essentially the same as periodic time-based scheduling.

The garbage collectors of Nettles and O'Toole [21] and North and Reppy [22] run the collector in a separate thread, which appears to be a time-based approach. However, Nettles and O'Toole dynamically detect situations in which the mutator is allocating faster than the collector, in which case they pause the mutator while a fixed amount of work is performed.

North and Reppy's collector does not have any feedback, nor is there any way of balancing the mutator/collector quanta, so mutators with high allocation rates may fail.

9. 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 (no more than 4% of traced data in our measurements). Because fragmentation is bounded, the collector has a provable space bound yet retains a lower space overhead than a fully-copying real-time collector.

The key to fully incremental defragmentation is a low-overhead read barrier that maintains consistency without compromising the real-time bounds. We have shown that in an optimizing Java compiler, a highly efficient software read barrier can be implemented and will only cause a 4% mean slowdown.

We have implemented the collector and shown that for real applications it can achieve highly predictable mutator utilization rates with highly stable pause times at real-time resolution. It is generally able to achieve 45% utilization while the collector is on with only 1.6–2.5 times the actual memory high water mark of the application.

Acknowledgements

We thank David Grove for his assistance in implementing the read barrier optimizations, and the entire Jikes RVM team for providing the research platform which made this work possible. We also thank Rob O'Callahan, David Grove, Mike Hind, and the anonymous referees for their helpful comments.

10. REFERENCES

[1] ALPERN, B., ET AL. The Jalapeño virtual machine. *IBM Syst. J. 39*, 1 (Feb. 2000), 211–238.

[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] ARNOLD, M., AND RYDER, B. G. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming* (Málaga, Spain, June 2002), B. Magnusson, Ed., vol. 2374 of *Lecture Notes in Computer Science*, pp. 498–524.

[4] BACON, D. F., FINK, S. J., AND GROVE, D. Space- and time-efficient implementation of the Java object model. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming* (Málaga, Spain, June 2002), B. Magnusson, Ed., vol. 2374 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 111–132.

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

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

[7] BEN-YITZHAK, O., GOFT, I., KOLODNER, E. K., KUIPER, K., AND LEIKEHMAN, V. An algorithm for parallel incremental compaction. In *Proc. of the Third International Symposium on Memory Management* (Berlin, Germany, June 2002), pp. 100–105.

[8] BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. E. B. Beltway: getting around garbage collection gridlock. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, 2002), pp. 153–164.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] CHENG, P., HARPER, R., AND LEE, P. Generational stack collection and profile-driven pretenuring. In *Proc. of the Conference on Programming Language Design and Implementation* (June 1998). *SIGPLAN Notices*, *33*, 6, 162–173.

[14] 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.

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

[16] HUDSON, R. L., AND MOSS, E. B. Incremental garbage collection for mature objects. In *Proc. of the International Workshop on Memory Management* (St. Malo, France, Sept. 1992), Y. Bekkers and J. Cohen, Eds., vol. 637 of *Lecture Notes in Computer Science*.

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

[18] JOSEPH, M., AND PANDYA, P. K. Finding response times in a real-time system. *Computer Journal 29*, 5 (1986), 390–395.

[19] 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.

[20] MOON, D. A. Garbage collection in a large LISP system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, Aug. 1984), pp. 235–246.

[21] NETTLES, S., AND O'TOOLE, J. Real-time garbage collection. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 1993). *SIGPLAN Notices*, *28*, 6, 217–226.

[22] NORTH, S. C., AND REPPY, J. H. Concurrent garbage collection on stock hardware. In *Functional Programming Languages and Computer Architecture* (Portland, Oregon, Sept. 1987), G. Kahn, Ed., vol. 274 of *Lecture Notes in Computer Science*, pp. 113–133.

[23] 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.

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

[25] ZORN, B. Barrier methods for garbage collection. Tech. Rep. CU-CS-494-90, University of Colorado at Boulder, 1990.

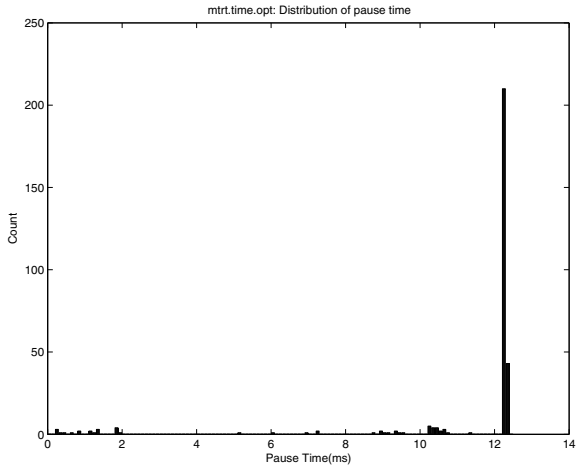

Figure 3: Time-based collector pause times for mtrt

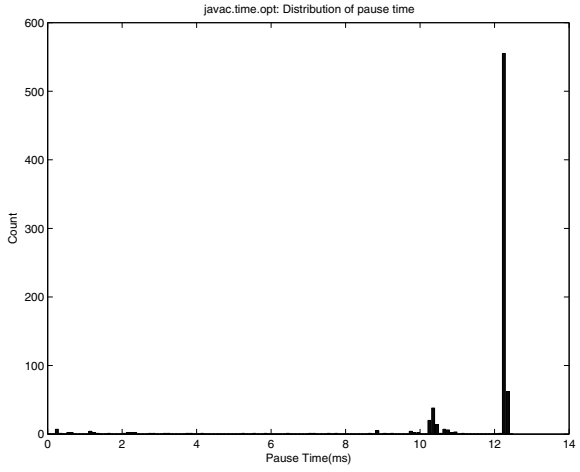

Figure 4: Time-based collector pause times for javac

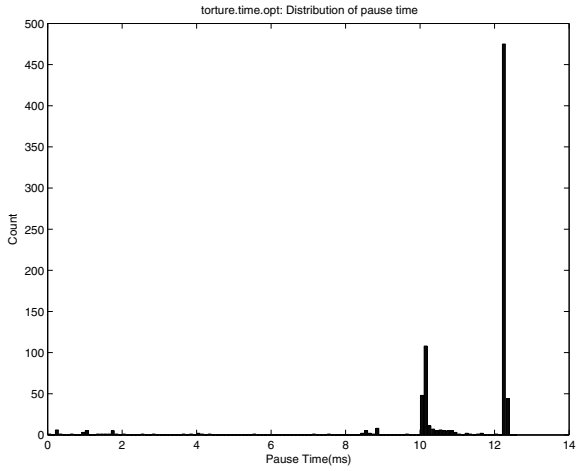

Figure 5: Time-based collector pause times for fragger

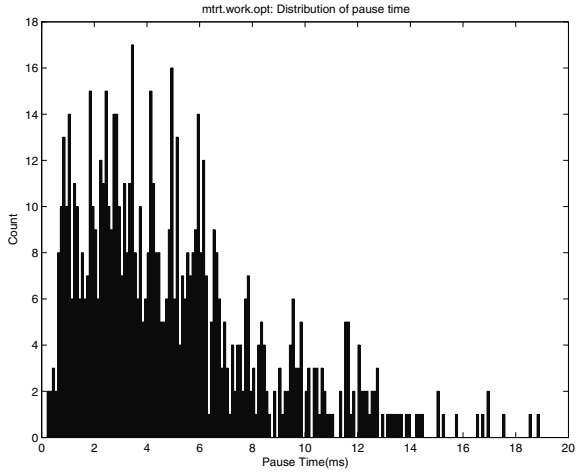

Figure 6: Work-based collector pause times for mtrt

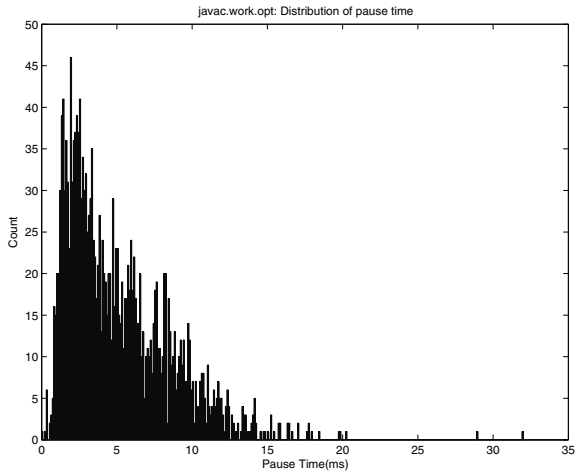

Figure 7: Work-based collector pause times for javac

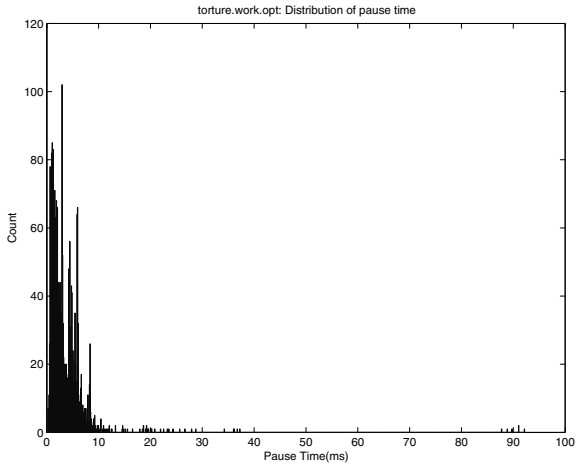

Figure 8: Work-based collector pause times for fragger

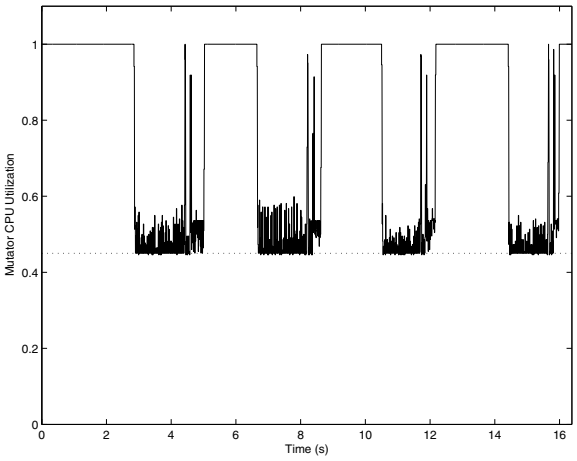

Figure 9: Time-based utilization for mtrt, $\Delta t = 22.2$ ms.

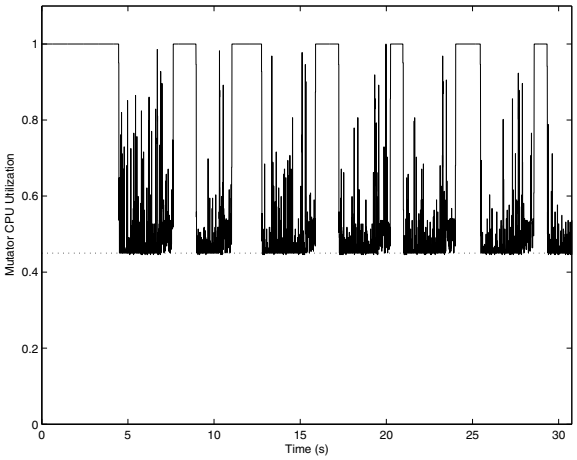

Figure 10: Time-based utilization for javac, $\Delta t = 22.2$ ms.

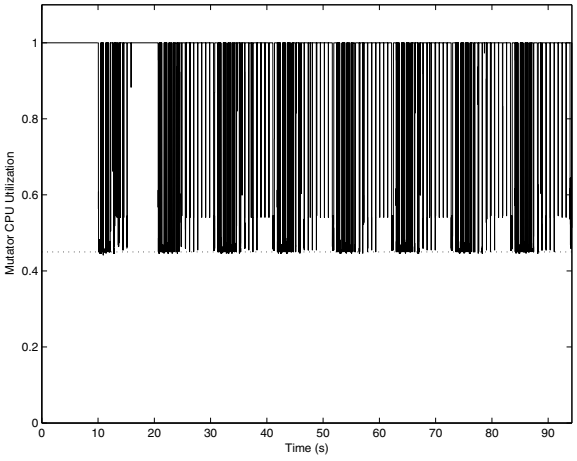

Figure 11: Time-based utilization for fragger, $\Delta t = 22.2$ ms.

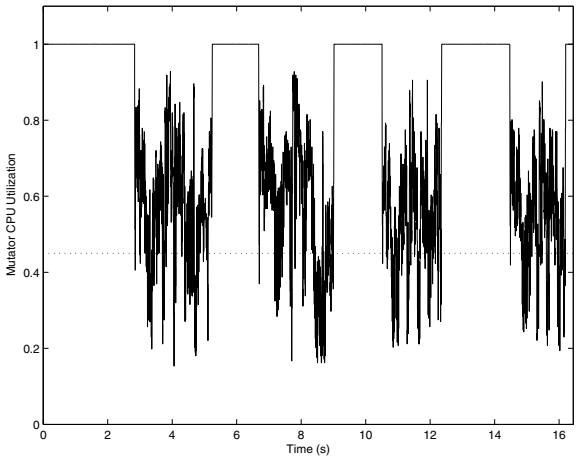

Figure 12: Work-based utilization for mtrt, $\Delta t = 22.2$ ms.

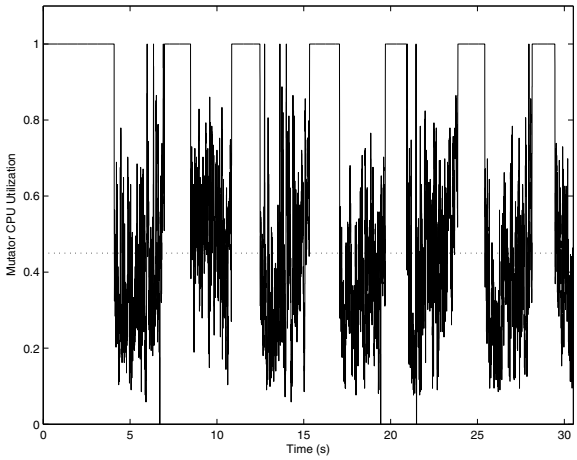

Figure 13: Work-based utilization for javac, $\Delta t = 22.2$ ms.

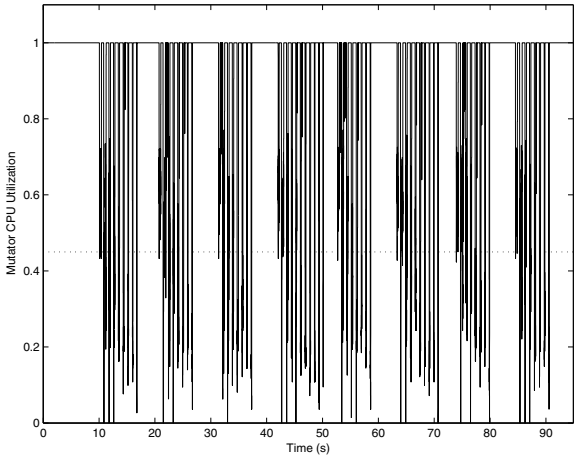

Figure 14: Work-based utilization for fragger, $\Delta t = 22.2$ ms.

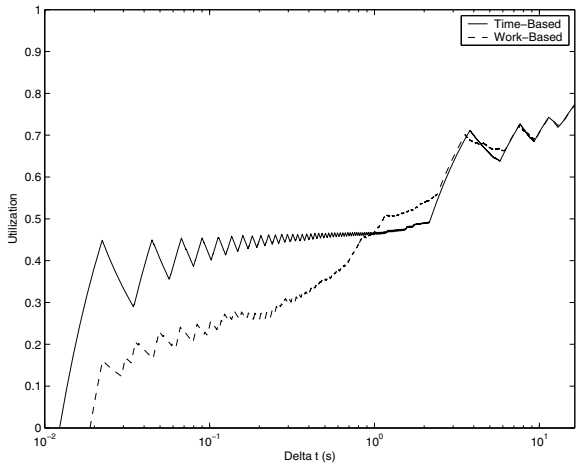

Figure 15: MMU for mtrt.

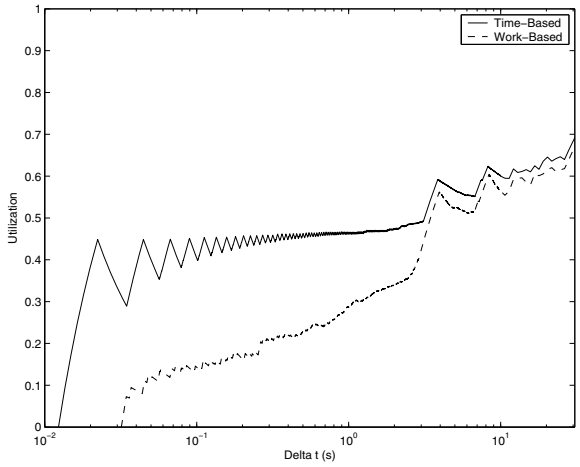

Figure 16: MMU for javac.

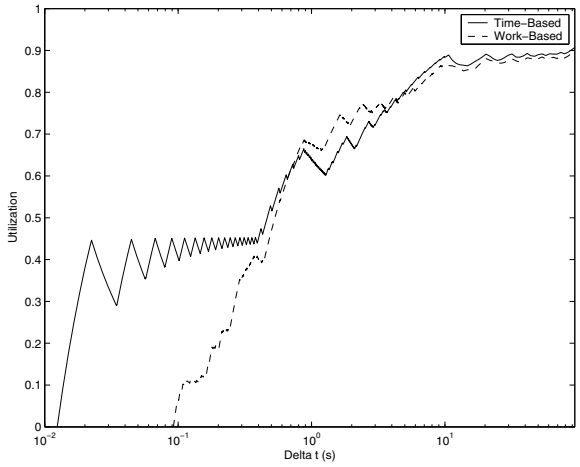

Figure 17: MMU for fragger.

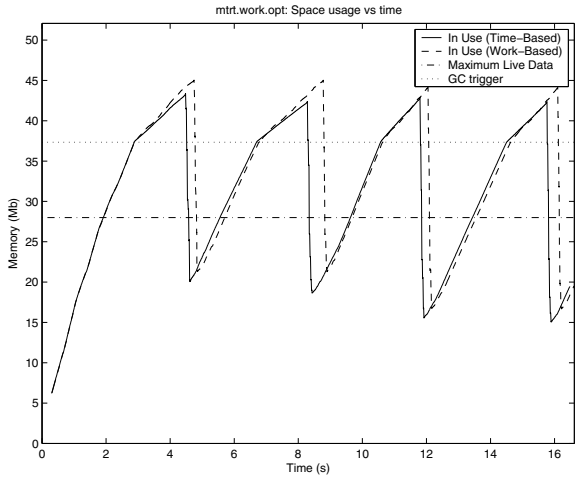

Figure 18: Space consumption by mtrt.

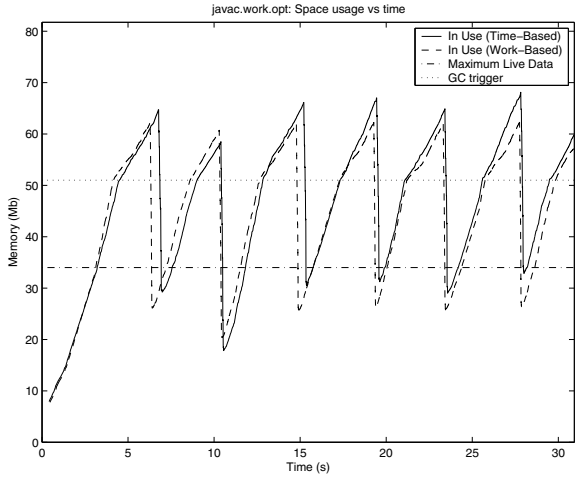

Figure 19: Space consumption by javac.

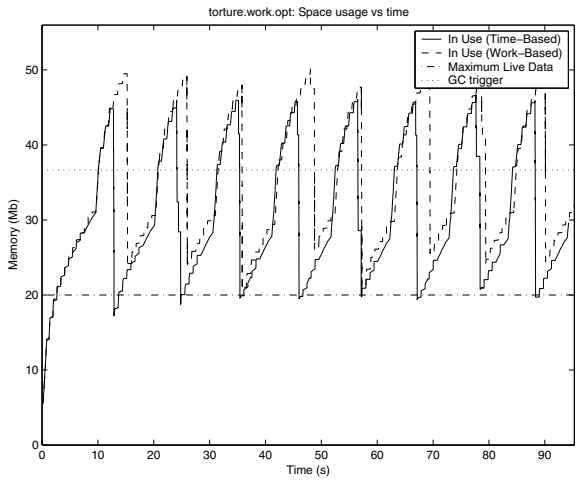

Figure 20: Space consumption by fragger.