Incremental Mature Garbage Collection Using the Train Algorithm

Jacob Seligmann Steffen Grarup

Computer Science Department, Aarhus University
Ny Munkegade 116, DK-8000 Århus C, Denmark
E-mail: {jacobse,grarup}@daimi.aau.dk

Abstract

We present an implementation of the *Train Algorithm*, an incremental collection scheme for reclamation of mature garbage in generation-based memory management systems. To the best of our knowledge, this is the first Train Algorithm implementation ever. Using the algorithm, the traditional mark-sweep garbage collector employed by the Mjølner run-time system for the object-oriented BETA programming language was replaced by a non-disruptive one, with only negligible time and storage overheads.

1 Introduction

Many programming languages provide automatic *garbage collection* to reduce the need for memory management related programming. However, traditional garbage collection techniques lead to long and unpredictable delays and are therefore unsatisfactory in a number of settings, such as *interactive* systems, where *non-disruptive* behavior is of paramount importance. *Generation-based* collection techniques alleviate the problem somewhat by concentrating collection efforts on small but hopefully gainful areas of memory, the so-called *young* generations. This reduces the need for collecting the remaining large memory area, the *old*, or *mature*, generation, but in no way obviates it. Traditionally, conventional techniques have been employed for old generation collection, leading to pauses which, although less frequent, are still highly disruptive.

Recently, Hudson & Moss have introduced an exciting new algorithm, the *Train Algorithm*, for performing efficient incremental collection of old generation space [HM92]. Using the algorithm, generational collectors can be extended to provide non-disruptive collection of all generational areas.

In the following, we present the results from a practical implementation of the Train Algorithm [GS93]. Section 2 describes the basic ideas behind the algorithm. Section 3 concerns the practical implementation issues. Section 4 presents the measurement results. Section 5 points out future research directions.

2 The Algorithm

This section deals with the theoretical aspects of the Train Algorithm. For a more detailed presentation of the algorithm including a couple of enlightening examples, see [HM92, GS93].

Section 2.1 outlines the main ideas behind the algorithm and introduces the terminology needed in later sections. (Readers familiar with the algorithm may wish to skip this part.) Section 2.2 identifies and corrects a subtle error in the original algorithm. Section 2.3 addresses the treatment of highly referenced objects.

2.1 The Train Algorithm

2.1.1 The Train Metaphor

The *Train Algorithm* is an incremental garbage collection scheme for achieving non-disruptive reclamation of the oldest generational area, the *mature object space*. The algorithm achieves its incrementality by dividing mature object space into a number of fixed-sized *blocks* and collecting one block at each invocation. The key contribution of the algorithm lies in showing how all garbage (even cyclic structures larger than the size of an individual block) can be recognized and reclaimed while only processing a single block at a time.

To achieve this, the algorithm arranges the blocks into disjoint sets. With a striking metaphor, Hudson & Moss refer to the blocks as *cars*, and to the set of blocks to which a car belongs as its *train*. Mature object space can then be thought of as a giant *railway station* with trains lined up on its tracks, as illustrated in Figure 1.

Just as in real life, cars belong to exactly one train and are ordered within that train. The trains, in turn, are ordered by giving them sequence numbers as they are created. This imposes a global lexicographical ordering on the blocks in mature object space: One block precedes another if it resides in a lower numbered (i.e. older) train; or if both blocks belong to the same train, then if that block has a lower car number (i.e. was added to the train earlier on) than the other. In the example structure shown in Figure 1, Car 1.1 precedes Car 1.2, Car 1.2 precedes Car 1.3, Car 1.3 precedes Car 2.1, and so on.

Intuitively, the Train Algorithm works by constantly clustering sets of related objects. In this way, it eventually collapses any interlinked garbage structure into the same train, no matter how complex. In the following, we shall see how this is achieved.

2.1.2 Car Collection Strategy

Each invocation of the Train Algorithm processes the lowest numbered car of the lowest numbered train in the system. Its space is reclaimed as follows.

First, a check is made to see whether there are any references into the train to which the car being collected belongs. If this is not the case, then the entire train contains only garbage and all its cars are reclaimed immediately. (This

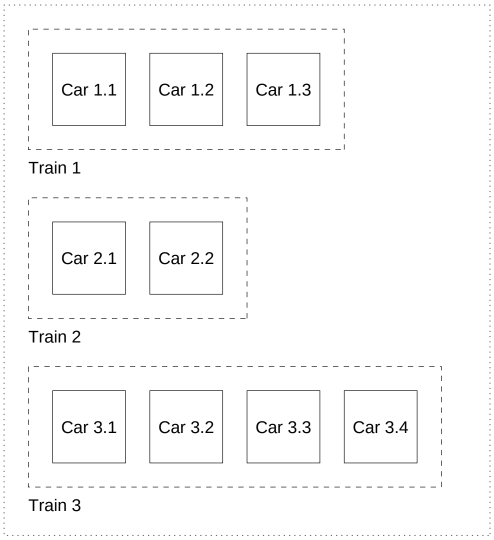

Figure 1: Mature object space structure

is the part of the algorithm which enables large cyclic garbage structures to be recognized and reclaimed, even if they are too big to fit into a single car. Section 2.2 explains why such structures always end up in the same train.)

Otherwise, all objects residing in the car being collected referenced from outside the train are evacuated as follows. Objects referenced from other trains are moved to those trains; objects referenced from outside mature object space are moved to any train except the one being collected. If a receiving train runs full, a new car is simply created and hooked onto its end. Then, in typical copy collector style, evacuated objects are scanned for pointers into the car being collected, moving the objects thus found into the train from which they are now referenced, and so on.

With the transitive closure of all externally referenced objects having been evacuated, the only live objects in the car being processed are those referenced (exclusively) from cars further down the train being collected. Such objects are evacuated to the last car of the train, as are the objects they reference, etc.

At this point, none of the objects remaining in the car being collected are referenced from the outside and are therefore garbage. Thus, the space occupied by the car is reclaimed and the collection is finished.

2.1.3 Tenuring Strategy

The tenuring strategy imposed by the Train Algorithm is simple: Objects promoted from younger generations may be stored in any train except the one currently being collected, or one or more new trains may be created to hold them.

2.1.4 Technical Issues

To facilitate collection of individual cars, each car has an associated *remembered set* containing information about all references residing outside the car pointing into it. Old generation cars will only be processed when all younger generations are collected, so the remembered sets need only contain references from other old generation cars. Since cars are processed in lexicographical order, one can further optimize the remembered set handling by only recording references from higher numbered to lower numbered cars. By the time a car comes up for collection, it will have the lowest number in the system, and thus its remembered set will be complete. This gives the advantage of not having to purge out stale remembered set entries in other parts of the system when a car is reclaimed.

The remembered sets are maintained by extending the generational pointer assignment run-time check (the *write barrier*). In addition to the traditional recording of references from older to younger generations, all pointer assignments between mature objects must now also be examined and possibly recorded in an old car remembered set. By letting the size of the old generation cars be of the form 2^n bytes, allocated on a 2^n byte boundary, one can maintain a *train table* over the train and car number associated with each mature object space block. Given a pointer address, the train table information can be quickly accessed by right-shifting the address n bits and using the result as an index [HMDW91].

2.2 Correctness

Given a garbage structure contained in mature object space, denote the trains holding it the set of *garbage trains*. The structure of this set will not be changed by the mutator because garbage objects are per definition unreachable and therefore immutable. The collector, on the other hand, will cause the set of garbage trains to shrink over time. As each garbage train is processed, the objects residing there are either recognized as garbage and reclaimed or evacuated to higher numbered garbage trains holding references to them. When processing reaches the highest numbered garbage train, the garbage structure will therefore have been collapsed and will be reclaimed.

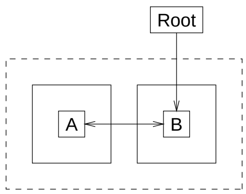

Figure 2: Error situation

This argument, of course, requires that all trains are eventually processed. Unfortunately, the Train Algorithm as presented so far provides no such guarantee. Consider the situation shown in Figure 2. A-B is a live structure, with object B referenced from some root. For the sake of simplicity, objects A and B are assumed to be too big to fit into the same car (more realistically, they could be large interlinked structures). When the car containing A is collected, A is simply moved into a new car at the end of the train. We call such collections *futile* because they neither evacuate, nor reclaim, objects. Now, before the next invocation of the algorithm (in which we would have hoped to evacuate object B), the mutator may change the root reference to point to A instead of B. Thus, at the beginning of the next collection, the overall object structure may be entirely unchanged. If the mutator continues to swap references behind the collector's back, it will force the collector to keep processing the A-B structure, thus never getting around to any of the other trains in the system.

This situation may be prevented as follows. Whenever a futile collection occurs, one of the external references to objects further down the train are recorded. (Such a reference must exist, otherwise the entire train would have been reclaimed.) A reference thus recorded is then used as an additional root for subsequent collections and is not discarded until a non-futile collection occurs. In the example above, this would cause the root reference to be recorded after the first collection, causing object B to be evacuated from the train at the next invocation of the algorithm even though it is no longer directly referenced.

Using this extension, it can be shown that each pass over a train will either reclaim or evacuate at least one object, and the entire Train Algorithm may be proven formally correct [GS93].

2.3 Popular Objects

Since the Train Algorithm processes only a single fixed-size car at a time, there is an upper limit on the number of bytes that are copied at each invocation. However, moving *popular* objects (objects with many pointers to them) could still make the algorithm disruptive. Whenever a popular object is moved, a large remembered set needs to be traversed and all the pointers to the object must be updated. Thus, it is impossible to bound the potential amount of work needed to move even a single object. The classic solution to this problem is to require that all objects be addressed indirectly so that whenever an object is moved, only a single cell needs to be updated. This technique is known to induce significant run-time overheads [Ung86].

As an alternative approach, Hudson & Moss tentatively suggest that cars containing popular objects should not be collected at all. Instead, such cars should be retained, logically (but not physically) moving them to the end of the newest train [HM92, Section 6]. Unfortunately, a closer analysis shows that this approach may leave garbage undetected, so a number of refinements are needed.

First, one cannot simply *retain* popular cars, but must evacuate all non-popular objects from them. Otherwise, garbage contained in or referenced from popular cars will not be detected and reclaimed.

Second, one should not move popular cars to the end of the *newest* train. This would cause large garbage structures scattered across several trains each containing a popular member to survive forever because such structures would never be collapsed. Instead, popular cars should only be moved to the end of the highest numbered train from which they are referenced.

Third, cars containing several unrelated popular objects pose a problem. Such cars will have to be *split* in some way so that each popular object can be moved to the end of the highest numbered train from which it is referenced.

With the popular object treatment outlined above, the Train Algorithm can be shown to correctly identify and reclaim all garbage [GS93]. However, such schemes may be quite costly to implement in practice. Since the remembered sets associated with popular cars may be arbitrarily large, one cannot afford to run through them to find references to non-popular objects, or to find the highest numbered train from which a popular object is referenced. Also, logically split cars are not easily handled in a train table setting. A naive way to overcome these problems would be to associate a remembered set with each object, to always maintain the highest numbered train from which an object is referenced, and to extend the train table with balanced binary search trees for split areas. In practice, more sophisticated solutions are undoubtedly needed to make the popular car approach a viable alternative to pointer indirection.

3 The Implementation

In order to test the Train Algorithm in a realistic setting, we incorporated it into the Mjølner run-time system for the object-oriented BETA programming language [MMN93]. To the best of our knowledge, this is the only implementation of the Train Algorithm to this date.

This section describes the implementation process. Section 3.1 presents the memory layout used in the two systems. Section 3.2 concerns the practical issues that had to be addressed. Section 3.3 discusses the overall implementation efforts.

3.1 Memory Layout

3.1.1 Original Mjølner BETA System

The Mjølner BETA System uses a modern generation-based collector with two generations [KLMM93]. The young generation (the *infant object area*, or *IOA*) consists of two fixed-sized semi-spaces reclaimed using copying collection. The old generation (the *adult object area*, or *AOA*) consists of a series of linked fixed-sized blocks reclaimed using a sophisticated three-phase mark-sweep collection scheme [Bar88]. In addition, large arrays of pointer-less objects are kept in a separate area (the *large value-repetition area*, or *LVRA*) consisting of linked fixed-sized blocks managed using free-lists and periodic compaction.

The advancement age between the two generations is determined adaptively using demographic feedback-mediated tenuring [UJ92]. To record references from old to young objects, a hash table-based remembered set is maintained, using a traditional but quite efficient write barrier to examine pointer assignments at run-time.

3.1.2 New Mjølner BETA System

The Train Algorithm implementation adhered to the general memory layout sketched in Section 2.1, including a train table. We used a car size of 64K bytes.

The global remembered set (recording references from the entire old generation to the young one) was replaced by a remembered set for each car (recording references from that car to the young generation). This approach enabled us to immediately discard the relevant entries when a car was reclaimed, rather than run through a global table to purge out expired references.

3.2 Practical Considerations

There are a number of issues which are left open by the general Train Algorithm description, but which must nevertheless be addressed when implementing it. For instance, one must decide when to create new trains, how often to invoke the algorithm, and whether to treat popular objects specially.

3.2.1 Evacuation Strategy

During a car collection, where should objects referenced from several trains be moved? We chose to evacuate objects to the last car of the train from which they were first seen during the scavenge (or to the last car of the newest train, if referenced from outside mature object space). Moving an object immediately to the highest numbered train referencing it might have given less superfluous copying, but seemed hard to implement efficiently.

Where should objects be moved when they are promoted? As objects were tenured into mature object space, we moved them into the last car of the newest train until the total size of the objects residing there exceeded a certain threshold (the *fill limit*) in which case we chose to create a new train. In this way, we tried to achieve a balance between initially having only a few objects in each train (to avoid accidental object structure infiltration which would require extra passes to disentangle) and putting many objects into each train (to avoid the storage overhead of having many sparsely filled cars in the system). We chose a default fill limit of 90%. The implications of this choice are presented in Section 4.5.

3.2.2 Invocation Frequency

While traditional batch collection algorithms are invoked only when memory runs full, their incremental counterparts must be called often enough to ensure that storage resources are never exhausted. Otherwise, they mutator must be suspended, and the collection becomes disruptive. Similarly, one must choose a collection frequency for the Train Algorithm which on one hand ensures that mature object space does not grow unacceptably full with garbage, but on the other hand does not waste too much time moving live objects around from car to car.

Our approach to finding the right balance was to select a fixed *acceptable garbage percentage* specifying the desired trade-off between speed and storage.

Whenever the amount of garbage detected in mature object space was larger than expected, the collection frequency was automatically increased in order to keep memory consumption down; and whenever the garbage ratio was lower than expected, the collection frequency was decreased to improve the overall performance. In addition, we introduced an upper limit on the number of young generation collections between each old generation collection step. Without such a limit, the algorithm would be called much to seldomly if little or no garbage was detected for a while.

Technically, there were a few obstacles to calculating the garbage ratio. Whereas batch algorithms can compare the size of mature object space before and after a collection, this is not possible in the Train Algorithm setting, where only a small area is collected at each invocation, where some objects may be scanned more frequently than others, and where there are always parts of memory that have not yet been processed. By associating with each train a counter holding the amount of objects tenured directly into it, one can, however, calculate the desired figure. The result is somewhat delayed because it is first obtained when the newest train in the system at counter initialization time is reclaimed, that is, after an entire sweep over mature object space. By initializing a new counter each time the collection of a new train commences, estimates nevertheless become available quite often. This approach turned out to work well in practice [GS93].

3.2.3 Popular Object Treatment

Because of the complications outlined in Section 2.3, we did not implement a full popular object scheme. However, all Mjølner BETA programs contain a special object, the *basic environment object*, which is referenced from a large number of other objects. The basic environment object is always alive by virtue of being one of the garbage collection roots. We therefore isolated it in a specially-marked car for which no remembered set was maintained. Whenever that car came up for collection, we logically moved it to form a new train, rather than physically moving the basic environment object and updating the large number of references to it. The benefits of this approach are demonstrated in Section 4.6.

3.3 Implementation Efforts

Incorporating the Train Algorithm into the Mjølner BETA run-time system meant having to discard practically all the original mature object space reclamation code. However, the rest of the system was left virtually untouched. This was an invaluable asset in debugging and fine-tuning the new system. All in all, the implementation was therefore surprisingly straightforward. The code for the new collector constituted about 2,000 lines of C (including extensive debug consistency checks), or about the same as the original mark-sweep collector.

4 The Results

In this section, we present the main results obtained from our Train Algorithm implementation. For a more detailed account, see [GS93].

Section 4.1 describes the benchmark system. Section 4.2 presents the most important result of our implementation, namely that disruptive old generation collection pauses can be completely removed using the Train Algorithm. Sections 4.3, 4.4, and 4.5 show the reverse side of the coin, namely the time, storage, and copying overheads induced. Section 4.6 concerns the special treatment of popular objects. Section 4.7 looks at the virtual memory behavior of the algorithm.

4.1 Benchmarks

4.1.1 Benchmark Programs

As our main reference point we chose the compilation of the entire Mjølner BETA compiler version 4.4.1 using itself. This program run enjoyed a number of desirable properties. First, it was easily reproducible, thereby ensuring that figures obtained across several runs remained comparable. Second, it ran for a relatively long period of time, so that many objects were promoted during the program execution, and so that quite a few of them died before the compilation was finished. Third, even though the application was not an interactive one, it was still instructive to examine the pause times incurred by the various area collections.

To ensure a wider exposure, we also measured the performance of an interactive run with the Sif hyper-structure editor [MIA90] and a discrete event simulation system modeling a series of orders pipelining through a set of machine queues.

4.1.2 Benchmark System

The tests were carried out using release 2.5.1 of the Sun SPARC variant of the Mjølner BETA System under SunOS 4.1.1. Default system parameters were used for both the original and new implementations, with block sizes of 512K bytes for IOA, AOA, and LVRA (except for the 64K bytes cars in the new implementation) and a 10% tenuring threshold. For the new system a 10% acceptable mature garbage ratio was adaptively aimed for, with an upper limit of ten young generation collections between each Train Algorithm invocation.

Except for the benchmark described in Section 4.7, all measurements were carried out on a 40 MHz Sun SPARC IPX 4/50 workstation equipped with 32M bytes of primary memory. Apart from a few page faults at the beginning of each run, there was practically no virtual memory activity. The times reported in the following are therefore the sum of the user and system CPU times. In practice, this turned out to correspond quite closely to elapsed wall-clock time.

4.2 Collection Pauses

This section presents the old generation collection pauses obtained. Table 1 shows the minimum, median, 90% percentile, maximum, and average amounts of time spent in each Train Algorithm invocation for each of the three benchmark programs. Figure 3 gives a more detailed impression of the pause time distribution for the longest-running benchmark, the compiler compilation.

Application	Compiler		Editor		Simulation	
Method	batch	train	batch	train	batch	train
Pause (min.)	0.27s	0.00s	0.31s	0.00s	0.11s	0.00s
Pause (med.)	1.39s	0.04s	0.49s	0.03s	0.16s	0.01s
Pause (90%)	3.06s	0.06s	0.49s	0.05s	0.17s	0.03s
Pause (max.)	3.21s	0.12s	0.57s	0.08s	0.18s	0.05s
Pause (avg.)	1.71s	0.04s	0.46s	0.03s	0.15s	0.01s

Table 1: Old generation collection pause distribution

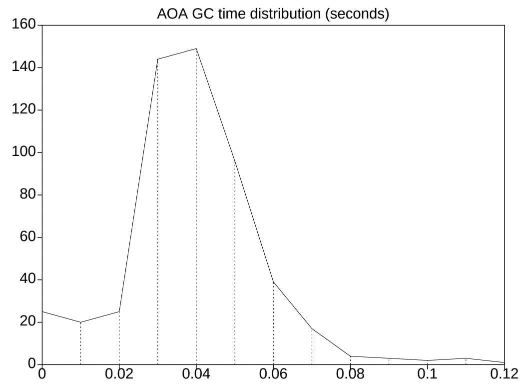

Figure 3: Pause distribution, compiler run

As can be seen, the new collector turned out to be extremely effective. For all three benchmark programs, it replaced a number of highly disruptive old generation collections with a series of non-disruptive ones, each just a few dozen milliseconds long. Thus, the Train Algorithm lived up to its promises of delivering pause times suitable for use in interactive systems.

4.3 Time Overhead

We now take a look at the time overhead incurred by using the Train Algorithm. In addition to the overhead caused in the old generation collector as a direct result of replacing the collection algorithm, we also examine the indirect time overhead caused in the remaining parts of the system (the young generation collector and the mutator) by changing the number of remembered sets, by requiring a more complex write barrier, and so on.

Application	Compiler		Editor		Simulation	
Method	batch	train	batch	train	batch	train
Old collector	17.11s	20.55s	1.37s	1.15s	1.05s	1.86s
Young collector	140.26s	150.14s	8.25s	8.95s	44.26s	32.88s
Mutator	816.09s	808.88s	93.51s	94.02s	183.40s	176.06s

Table 2: Total time distribution

Table 2 shows the total time spent in the various system components. In addition, approximately one second was spent on large object area free-list compaction in the compiler and editor runs. Since the Train Algorithm did not influence large object area management in any significant way, we shall not go further into that issue here.

4.3.1 Old Generation Collector

As could be expected, the price for achieving non-disruptive behavior was a general increase in the total old generation collection time. For the compiler compilation the increase was just above 20%. For the simulation run the increase was more dramatic, nearly 80%. This was mainly because many tenured objects died during the execution, causing the new collector to perform frequent steps (on average once every six young generation collections), thereby scanning nearly twice the amount of mature object space as its batch counterpart. Thus, the time overhead could probably have been replaced by a storage overhead by increasing the acceptable garbage percentage. For the editor session the old generation collection time *fell*. However, the last batch collection occurred just before the end of the run, so estimates obtained over a longer period of time would probably have revealed that the new collector was still somewhat slower.

In any case, the differences in old generation collection times were practically negligible in the overall picture. The extra time spent in the compiler and simulation runs corresponded to merely 0.4% of the total execution time of the programs, and the time (seemingly) saved in the editor session amounted to only 0.2%.

4.3.2 Young Generation Collector

The Train Algorithm approach had its implications on the young generation collector, too. First, the roots for each young object scavenge now had to be found in a number of distinct remembered sets associated with each car, rather than in a single remembered set associated with the entire mature object space.

Second, extra inter-car remembered set insertions had to be performed when objects were promoted.

For the compiler and editor runs, this amounted to an increase of about 0.0025 seconds for each young generation scavenge, or between 7% and 8%. For the simulation run, the result was a *decrease* in young generation collection time of about 25%. In all three cases, we believe that the main reason for the difference was the time spent scanning the remembered sets for roots. In the compiler and editor runs, the accumulated size of the remembered sets associated with the old generation cars far exceeded that of the single remembered set originally employed; in the simulation run, the accumulated size was significantly smaller.

Overall, the differences in young generation collection times corresponded to an increase in execution time of about 1% for the compiler and editor runs, and a decrease of almost 5% for the simulation session.

4.3.3 Mutator

The Train Algorithm approach also affected the mutator performance. On one hand, the run-time pointer assignment checks were slightly costlier, and the new inter-car remembered sets had to be maintained. On the other hand, the large old-to-young generation remembered set was now replaced with one for each car, which typically gave faster insertion times. Also, the train table approach enabled us to handle old and large objects without having to traverse long linked block lists.

For the compiler run, the net result was a 0.9% decrease in mutator time. For the editor session, a 0.5% increase was observed, while the total simulation mutator time decreased by 4%. In the overall picture the differences were, of course, slightly less.

4.3.4 Overall Result

Using the Train Algorithm, the total execution time for the compiler compilation increased by 0.6%, the total execution time for the editor session increased by 1.0%, and the total execution time for the simulation run decreased by 8%.

4.4 Storage Overhead

In addition to the performance overhead, Train Algorithm-based systems affect storage requirements in several ways. First, the inter-car remembered sets take up extra memory. Second, cars are not filled to capacity, but only hold related object structures; also, there is a trade-off between the car collection frequency and the storage required to hold garbage contained in yet unprocessed cars. Third, a train table, if employed, also takes up space.

4.4.1 Inter-Car Remembered Set Overhead

There were an average of 350 entries in each inter-car remembered set for the editor and simulation sessions, and 700 for the compiler run. With the hash table implementation used, this resulted in storage overheads of 6K to 8K bytes per car.

Even though BETA objects are generally highly interlinked because they all contain an environment reference (the *origin*) to their statically enclosing object, this was a surprisingly large figure. However, by fine-tuning the hash function and rehash strategies, we believe that the overhead could be significantly reduced.

4.4.2 Garbage Overhead

To measure the extra storage required to hold unprocessed garbage in the two systems, we performed a series of exhaustive mature object space traversals to determine how many of the objects residing there were actually alive. The result from the compiler run is shown in Figures 4 and 5.

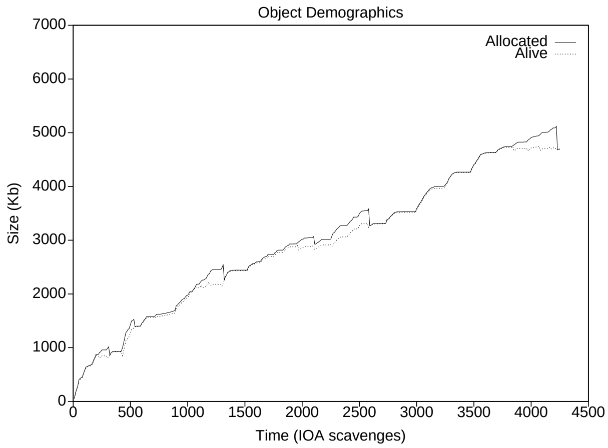

Figure 4: Mark-sweep garbage overhead

In the original system, new mature object space blocks were only allocated if the total old generation capacity was very close to being exhausted after a mark-sweep collection. Therefore, the difference between the two graphs was always less than the size of an old generation block, 512K bytes, as seen in Figure 4. In the Train Algorithm-based system, we chose to aim for a 10% garbage ratio and were quite successful, as seen in Figure 5.

Thus, the storage overhead caused by unprocessed garbage was about 10% for the compiler run. The same behavior was achieved for the editor session, while the old generation storage requirements *fell* a bit for the simulation run, simply because the program needed less old generation space than the size of the single block allocated in the mark-sweep implementation, and because the Train Algorithm collector managed to keep up with the high mortality rate.

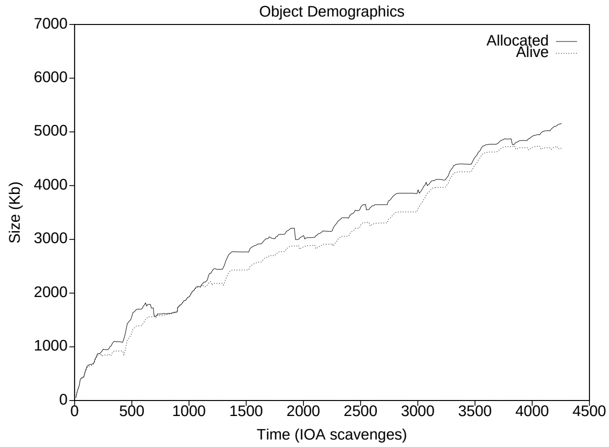

Figure 5: Train Algorithm garbage overhead

4.4.3 Train Table Overhead

Since the car size was 64K bytes and the address space had 2^{32} addresses, a train table with 64K entries was needed. As each entry consisted of two words (one for the train number and one for the car number), this gave a total train table size of 256K bytes.

Had we had more control over which parts of memory were used for heap allocation, we could have allocated a smaller train table. For instance, had it been known that only the lowest 256M bytes of memory would ever be used, a 16K bytes table with 4K entries would have sufficed. However, this may not have been worth the effort since any unused parts of the train table would be swapped out by the virtual memory system as needed.

4.4.4 Overall Result

In total, the old generation storage overhead for the Train Algorithm implementation was between 10% and 20%. In the overall picture, this corresponded to increases in application storage requirements of between 4% and 8%.

4.5 Copying Overhead

Theoretically, the Train Algorithm may require $O(Mn^2)$ invocations to collect a train consisting of n cars, where M is the number of objects containing at least one reference that may fit into a single car [GS93]. In practice this behavior should not be expected because it would require extremely poor locality of reference.

In order to get an impression of the Train Algorithm's practical behavior in this area, we recorded information about the number of cars in each train and the number of collections it took to reclaim one using various fill limits (see

Section 3.2.1). As it turned out, the compiler benchmark gave rise to the most complex object structures. The results of this run are shown in Table 3.

Fill Limit	10%	25%	50%	75%	90%	95%	99%	none
Cars (avg.)	99.8	64.2	57.0	52.7	51.7	51.5	51.4	51.6
Cars (max.)	154	103	94	89	86	86	85	85
Trains (avg.)	59.2	23.8	13.9	9.7	8.3	6.3	3.1	2.9
Trains (max.)	94	36	19	15	13	10	5	3
Length (avg.)	1.7	2.7	4.1	5.4	6.5	8.2	16.8	17.6
Length (max.)	17	22	23	27	32	42	81	79
Passes (avg.)	1.04	1.13	1.10	1.14	1.13	1.15	1.17	1.19
Passes (max.)	2.50	2.00	2.33	2.00	2.00	1.80	1.75	1.65

Table 3: Train structure statistics

The first six rows show the average and maximum number of cars and trains in the system, as well as the train length. The last two rows show the number of passes needed to reclaim each train, expressed as the ratio between the number of invocations performed and the initial length of the train. Thus, a lower ratio signifies less superfluous copying.

As should be expected, lowering the fill limit resulted in more, shorter trains which required fewer passes to be reclaimed. The cost for this behavior was an increase in the number of cars, and thus in the overall storage consumption. However, even in the most storage preserving setting in which new trains were only created when the collection process reached the newest train and needed somewhere else to evacuate objects to, the average amount of redundant copying was less than 20%. Thus, we did not see any signs of worst-case behavior occurring in practice.

4.6 Popular Objects

As described in Section 3.2.3, we chose to create a special car to hold the basic BETA environment object in order to avoid moving it and having to update the multitude of references to it. To investigate the consequences of this approach, we temporarily disabled the special treatment for the compiler run (which had by far the most objects referencing to the basic environment object). The result is shown in Table 4 and Figure 6 (cf. the second column of Table 1, and Figure 3).

As can be seen, the maximum collection time doubled. This was because there were more than 50,000 references to the basic environment object, causing over 200K bytes of pointers scattered throughout mature object space to be updated whenever the car containing it was collected. In addition, the total time spent on old generation collection rose to 26.56 seconds, or nearly 30%. This was because hundreds of inter-car remembered set insertions were now typically required at each collection step as objects referencing the basic environment

Application	Compiler
Method	train
Pause (min.)	0.00s
Pause (med.)	0.04s
Pause (90%)	0.06s
Pause (max.)	0.23s
Pause (avg.)	0.05s

Table 4: No popular treatment, pause table

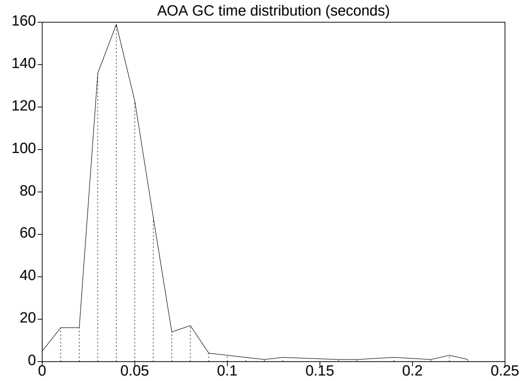

Figure 6: No popular treatment, pause plot

object were evacuated past it.

As can be seen from the figures above, our simple popular object scheme therefore turned out to be highly effective in practice.

4.7 Virtual Memory Behavior

To investigate the virtual memory behavior of the Train Algorithm, we ran the compiler benchmark on a 33 MHz Sun SPARC ELC 4/25 workstation equipped with only 12M bytes of primary memory. Since we only measured user and system time, not elapsed wall-clock time, we were unable to determine the exact amount of time spent on disk swapping. Instead, we monitored the page fault distribution.

The mark-sweep and Train Algorithm collectors scanned roughly the same amount of mature object space during the execution, about 30M bytes. This gave rise to 3,800 page faults in the mark-sweep implementation, and 4,500 in the Train Algorithm implementation. In both cases, this corresponded to around 95% of the total number of page faults. While there were nearly 20% more page faults in the Train Algorithm implementation, they were far better temporally distributed. For instance, the last mark-sweep collection caused over 1,000 pages to be swapped into primary memory, spending 6.4 seconds in the system kernel alone (not counting the disk access). In contrast, the Train Algorithm collector caused an average of less than 10 page faults per collection step. Still, a few collections of cars with large remembered sets caused nearly 100 page faults each, with up to 0.25 seconds spent in the system kernel.

To gain a better impression of the virtual memory behavior of the Train Algorithm, more detailed measurements are required. Nevertheless, our simple experiment seems to indicate that the algorithm is certainly a step in the right direction.

5 Future Work

To understand the full potential of the Train Algorithm, more experiments are needed. For instance, the remembered set implementation seems to play a crucial role. It would be highly instructive to examine the effects of using alternative strategies, such as dirty pages, cards, etc. [HMS92]. Also, further experiments with different car sizes and evacuation strategies would be interesting, as would more work concerning the treatment of popular objects. Finally, the Train Algorithm approach may be applicable in other areas, such as distributed garbage collection or collection of persistent stores.

6 Conclusion

Garbage collection simplifies programming, but has often been considered unfeasible in interactive systems. The most common argument has been that it is impossible to guarantee sufficiently short delays without introducing large overheads. With current hardware and sophisticated reclamation techniques, this is no longer true. Our Train Algorithm implementation shows that collection delays can be brought down to a few milliseconds, with very low maximum delay and a modest increase in storage requirements. Thus, for a large class of interactive applications, there is no need to exclude automatic memory management.

7 Acknowledgments

We wish to thank Ole Lehrmann Madsen and Eric Jul for encouraging us to write this paper. We are also grateful to Søren Brandt, Richard Hudson, Eric Jul, Ole Lehrmann Madsen, Eliot Moss, and Mario Wolczko for reading earlier drafts and providing us with many helpful suggestions.

References

[Bar88] Knut Barra: *Mark/sweep compaction for substantially nested Beta objects*, NCC-Note DTEK/03/88, Norwegian Computing Center, March 1988.

[GS93] Steffen Grarup and Jacob Seligmann: *Incremental Mature Garbage Collection*, M.Sc. thesis, Computer Science Department, Aarhus University, Denmark, August 1993. Also published as Technical Report DAIMI IR-122, Computer Science Department, Aarhus University, Denmark, September 1994. Electronic version available via anonymous ftp from `ftp.daimi.aau.dk` as `pub/thesis/gcthesis.ps.{Z,gz}`.

[HM92] Richard L. Hudson and J. Eliot B. Moss: *Incremental Collection of Mature Objects*, Proceedings of the International Workshop on Memory Management, September 1992, pp. 388–403.

[HMDW91] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight: *A Language-Independent Garbage Collector Toolkit*, COINS Technical Report 91-47, University of Massachusetts, Amherst, September 1991.

[HMS92] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanović: *A Comparative Performance Evaluation of Write Barrier Implementations*, OOPSLA '92 Proceedings, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 92–109.

[KLMM93] Jørgen L. Knudsen, Mats Löfgren, Ole L. Madsen, and Boris Magnusson (eds.): *Object-Oriented Environments: The Mjølner Approach*, Prentice Hall, 1993.

[MIA90] Mjølner Informatics Report MIA 90-11: *Sif: A Hyper Structure Editor – Tutorial and Reference Manual*, Mjølner Informatics ApS, Science Park Aarhus, 1990.

[MMN93] Ole L. Madsen, Birger Møller-Pedersen, and Kristen Nygaard: *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.

[Ung86] David Ungar: *The Design and Evaluation of a High Performance Smalltalk System*, Ph.D. thesis, University of California, Berkeley, May 1986.

[UJ92] David Ungar and Frank Jackson: *An Adaptive Tenuring Policy for Generation Scavengers*, ACM Transactions on Programming Languages and Systems, Vol. 14, No. 1, January 1992, pp. 1–27.