

Visualising The Train Garbage Collector

Tony Printezis
tony@dcs.gla.ac.uk
Dept of Computing Science,
University of Glasgow,
17 Lilybank Gardens,
Glasgow, G12 8RZ,
Scotland

Alex Garthwaite
alex.garthwaite@sun.com
MS UBUR02-311,
Sun Microsystems Laboratories,
1 Network Drive,
Burlington, MA 01803-0902,
USA

ABSTRACT

This paper presents a novel method for visualising an incremental garbage collector, based on the well-known Train algorithm, that generates concise snapshots of its state and informative graphs of its operation over time. To obtain these visualisations we used GCspy, a generic heap visualisation framework. We show how this easy-to-use tool provided a visualisation model that was effective in both confirming our pre-existing beliefs about the collector's operation and, more interestingly, highlighting unexpected patterns in its behaviour. Based on this successful experience, we advocate the use of similar visualisation approaches to better understand and ultimately tune, profile, and improve other equally complex garbage collectors.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*memory management (garbage collection)*; D.2.5 [Software Engineering]: Testing and Debugging—*tracing*

General Terms

Measurement, Experimentation

1. INTRODUCTION

Automatic memory management, or garbage collection [11, 30], improves the robustness of software. By removing the burden of explicitly reclaiming memory from the programmer, well-known classes of errors, including dangling references and certain kinds of memory leaks, can be avoided. Further, garbage collection can simplify the interfaces between separately built components, as these libraries no longer need to negotiate responsibility for memory reclamation.

However, garbage collectors themselves can be quite complex, difficult to understand, and exhibit almost 'chaotic' behaviour, as small changes in application behaviour might radically affect a collector's performance. Several tools [4, 17, 27, 22] have been developed and marketed that help capture some information about

the heap, e.g., to track memory usage and, through inspection, to identify potential memory leaks. However, these are targeted mainly to application developers and, traditionally, there have been no generic tools that help visualise the internal workings of the collectors themselves.

The *GCspy framework* [19] was developed exactly for this purpose. It separates the problem of visualising a collector into two parts: gathering of the collector's state inside the collector and its visualisation inside a separate process. Instrumenting a collector, so that it can be visualised with GCspy, involves writing a small amount of 'glue code'; this is an easy task, as the framework has been designed to be adaptable and to provide support for platform-independent, still highly-customised, visualisations. GCspy also provides facilities that show the effects of a collector's operation over time and not only for particular moments in time.

This paper reports on how we adapted GCspy to visualise an incremental collector that has been implemented inside the Sun Microsystems Laboratories Virtual Machine for Research, henceforth *ResearchVM*¹. This collector is based on the well-known Train algorithm [9], which is notorious for having many subtleties in its operation and being difficult to analyse and tune. Using the GCspy-generated visualisations, which both confirmed expected patterns in the collector's behaviour and revealed new, previously undiscovered ones, we obtained a considerably better understanding of this particular collection technique.

Furthermore, GCspy itself was heavily stressed to accommodate the collector in question, given the complex nature of its behaviour. However, the fact that it was successfully incorporated into it reinforces the claims of generality and adaptability of the framework. Additionally, the fact that its visualisations provided interesting insights into the collector's operation also provides proof of the framework's usefulness.

1.1 Paper Overview

Section 2 gives an overview of systems and techniques for visualising and analysing program and heap behaviour. Sections 3 and 4 provide a brief overview of the architecture of GCspy and of the operation of the Train algorithm respectively. Section 5 describes the way the GCspy framework was adapted to visualise the Train collector and section 6 enumerates the insights that we obtained from the generated visualisations. Finally, section 7 concludes the paper and suggests potential future directions.

¹This virtual machine has been previously known as the 'Exact VM' and has been incorporated into products; for example, the Java™ 2 SDK Production Release, for the Solaris™ operating environment.

2. RELATED WORK

For a good introduction to garbage collection the reader is referred to two excellent publications: Jones' book [11] and Wilson's survey [30].

The Train garbage collector, also referred to as the *mature object space* or *MOS* algorithm, was first proposed by Hudson and Moss [9]. It addresses a weakness in generational collection [13, 14, 25]: collection of the oldest generation may result in long, disruptive pauses to the application. The basic idea is to divide the oldest generation into roughly equal fixed-sized regions and collect these regions independently, a few at a time. To aid in this incremental collection, references between regions are maintained in remembered sets. Through careful placement policies that retain a proper logical ordering on the regions, both progress and the ability to collect arbitrarily-sized garbage structures are guaranteed. Further, the logical ordering allows us to only maintain references from objects in higher regions to ones in lower regions. A more detailed description of the Train collector is given in section 4.

Seligmann and Grarup [21] were the first to implement and study a collector based on this technique. In addition to clarifying and correcting some issues related to making progress, they studied the pause-time and heap behaviour of a number of applications written in the Beta language. They also reported some statistics justifying the policies they chose.

Collectors based on the Train algorithm remain an active area of research. For example, because the Train algorithm allows for the natural migration of related data structures, and because its remembered sets allow collection of a given region to only impact others containing references to objects in it, extensions to the basic algorithm for persistent [15, 16, 31] and distributed [8] environments have also been proposed.

Moving to heap visualisation, there are several runtime systems that have built-in heap visualisers, e.g., SELF [26] and Modula-3 [3]. In such systems, however, the visualiser is hard-coded with the layout of that particular heap, it is not extensible or customisable, and it has limited features (usually just showing heap occupancy). This is a more limited approach, compared to the one adopted by the GCspy framework (see section 3). As far as the authors are aware, there have not been any previous attempts to visualise the Train collector.

Apart from heap visualisers, a number of other tools have been developed to support object management, both for Java and for other languages. Some tools are source-level, such as class browsers [12]; others offer application debugging support, profiling, and other analysis of run-time behaviour (Sun's HAT [23], the serviceability agent in Sun's Java HotSpot™ virtual machine [20], Geodesic's Great Circle [7], IBM's Jinsight [4, 10], ParaSoft's Inuse [17], Sitraka's JProbe [22], VMGEAR's OptimizeIt [27]; interestingly, the majority of them are commercial products). Of the latter, even though some are intended for the runtime system implementer [20], most focus on the needs of the application programmer. Because of this, their results are independent of the garbage collector used, hence they are not appropriate for analysing different aspects of the behaviour of the collector itself. In contrast, the GCspy framework was designed for exactly this purpose.

3. GCspy FRAMEWORK OVERVIEW

GCspy is a heap visualisation framework. It has been designed with the following goals in mind:

1. to be scalable and be able to visualise large heaps of realistic applications and not only be limited to 'toy systems';

2. to be easily incorporated into a large number of different virtual machines and language runtime systems; and
3. to be able to visualise a wide variety of memory managers, whether they have a garbage collector or rely on explicit memory de-allocation.

In order to meet the scalability requirement, the visualisation granularity of GCspy is coarse, i.e., it visualises information about heap regions (referred to as *blocks*) and not about individual objects. Even though this approach imposes some information loss, it provides better scalability when the heap size and object number increase to gigabytes and tens of millions respectively. Further, the granularity (i.e., the block size) can be adjusted according to the required heap size and available screen size in order to reach the best possible trade-off between them.

GCspy's architecture and abstractions, introduced to meet the portability and adaptability requirements mentioned above, are briefly overviewed in the remainder of this section. A more detailed description of GCspy and its goals can be found elsewhere [19].

3.1 Abstractions

GCspy heap blocks are represented on the screen by rectangular *tiles*. These can be colored and shaded according to some attribute of the corresponding blocks (e.g., number of objects, percentage of used space, state of the card table). The number of tiles is expected to be low enough for all of them to fit on the user's screen and show a concise view of the state of the entire heap. This would not have been easily feasible if the visualisation was done at the object level.

A series of consecutive tiles on the screen is referred to as a *space*. Several distinct spaces might be required to visualise some systems (e.g., generational memory managers can be represented by one space per generation). GCspy can handle an arbitrary number of spaces and each space can have an arbitrary number of tiles, with the only limitation being the size of the user's screen. Even though the original rôle for a space was to represent a contiguous part of the heap as a series of blocks, this was later generalised further so that a space can represent any other relevant *component* of the system that can be viewed as a series of tiles (e.g., the free lists of an in-place de-allocating memory manager, one list per tile; the loaded classes of a Java virtual machine, one class per tile).

Each space is associated with a number of *streams*. Each stream is a series of values, one value per tile, that represents a particular attribute of the space (e.g., number of objects per heap block, length of each free list). Currently, GCspy can show views of each space based on only one of its attributes at a time. In the future such views might be extended to utilise several attributes. Additionally, when GCspy needs to present any of the stream data in textual format (see appendix C), some additional customisation information for each stream allows it to do so in a way that is appropriate for each particular situation, e.g., the value 128 can be presented as Used Space: 128 bytes or Length: 128 nodes depending on what that stream represents. The framework can also automatically calculate percentages given fixed or dynamically-generated maximum values (e.g., Used Space: 128 bytes (25%), given that the size of a block is 512 bytes) or interpret values as enumerations (e.g., the state of a card in the card table can be DIRTY or CLEAN; enumerations can also be used to express boolean values). Further, summary information on all the values of each stream can also be shown if necessary (e.g., overall number of objects in the heap or break-down of how many cards are dirty and how many are clean)².

²Interestingly, these summaries cannot be accurately calculated by the client. Consider a stream that represents the number of objects per heap block. If an object spans two blocks, it will be calculated in the stream val-

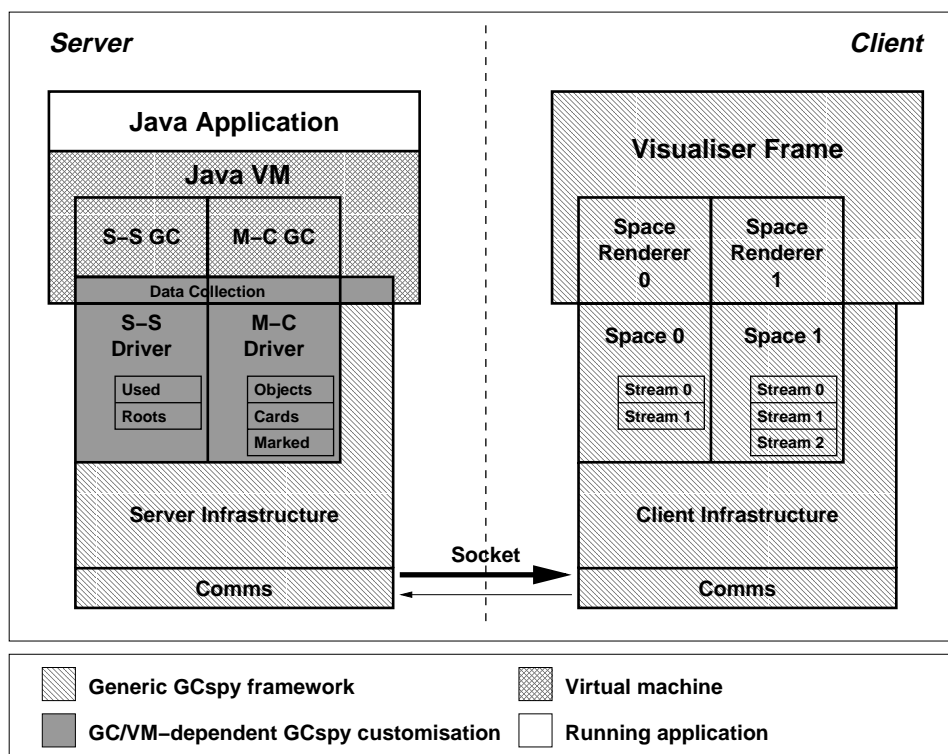


Figure 1: The GCspy client-server architecture.

The two main GCspy abstractions, spaces and streams, were introduced to facilitate the adaptability and portability of the framework. A wide variety of memory managers can be expressed in terms of these abstractions. This is simply done by defining how many spaces will be needed, how many streams per space, and adding some additional presentation information to customise the look of the space. After that, the state of the system can be visualised by just providing the data values for all the streams.

3.2 Architecture and Drivers

A client-server architecture has been adopted for GCspy, with the system that is being visualised being the server and the visualiser being the client. Separating the visualiser from the visualised system has several advantages. First, running them as different processes, especially when run on separate machines, ensures that the visualiser's data structures and operation do not interfere with the process being visualised. Second, both systems can be written in a different language, which further facilitates the portability and maintenance of the framework. Finally, the visualiser can be connected to and disconnected from the server as and when necessary.

Figure 1 illustrates the GCspy architecture when incorporated into a Java virtual machine. The assumption is that this Java virtual machine has a generational memory system with a semi-spaces (S-S) collector managing the young generation and a mark-compact (M-C) collector managing the old generation. For a collector (or any other component) to be visualised with GCspy, a small module, called a *driver*, has to be written. It maps the state of the component to the GCspy abstractions, i.e., it sets up the spaces and streams and

ues for both blocks. However, a sum of those values will ultimately yield an inaccurate result, as that object will be calculated twice. Because of this, the summaries are calculated by the driver at the server-side.

generates the stream data every time a new transmission to the visualiser must take place (this happens at particular *events* inside the server, such as the start of a young collection or the start of an old collection). A driver can manage one or more spaces, depending on the situation (drivers for most collectors require only one space; however the Train driver, described in section 5, requires three).

An important point illustrated in figure 1 is that the GCspy framework assumes that it will be incorporated inside the system that it will visualise. This is achieved by writing the required drivers (these will be specific for that particular system) and inserting a small amount of code into each component to be visualised which, when necessary, collects its state and communicates it to the corresponding driver. The rest of the framework (not only the remaining parts of the server but also the entire visualiser) is general-purpose.

The communication between client and server is performed using a custom binary protocol over standard TCP sockets. When the visualiser connects to a server, it receives some bootstrap information (number of spaces, number of streams in each space, presentation information per stream, etc.) that allows it to be dynamically configured for that particular server. This is how the framework can be incorporated into different systems, without requiring the visualiser to be customised. After the bootstrap phase has been completed, a 'push' model is employed: the server collects the stream data via the drivers and sends it to the visualiser whenever certain events (which are server-dependent but customisable for each server) are reached. A small amount of communication from the client to the server can also take place, mainly driven by user actions (e.g., in order to pause and restart execution).

The GCspy framework imposes a performance penalty to the system that is being visualised only when the visualiser is connected. Filtering facilities are also available. They allow the user

to turn specific events on and off or enable them periodically; any events that are skipped do not impose a performance penalty either. This filtering is useful to allow the user to obtain a better trade-off between the amount of visualisation performed and the amount of progress the system makes.

3.3 Traces and Histories

Another useful facility that is provided with the GCspy framework is the ability to store transmissions from a server and replay them at a later time. This is performed by attaching a special client to the server that simply receives transmissions and stores them to a *trace file*. A custom binary format is used for these trace files. It is transparent to the server whether the client connected to it is the trace-storing one or the visualiser. Given the coarse granularity of the visualisation, which decreases the amount of information transmitted from the server to the client, trace files are very compact and also compress very effectively. To replay a trace file, a special server reads it and transmits the information to the visualiser using the standard protocol. It is transparent to the visualiser whether it is connected to a ‘live’ system or the trace-replaying server.

Additionally, it is interesting to discover how the state of a system changes over time (as the standard GCspy window, illustrated in figure 4, only shows the state of the system at one particular point in time; subsequent transmissions will overwrite previous ones). To achieve this, GCspy can produce images which are referred to as *history graphs*. A history graph is a two-dimensional grid of very small tiles (large tiles would generate prohibitively large images). Each row corresponds to a single GCspy transmission and the brightness of its tiles is adjusted according to the values of the stream being visualised. When a new transmission reaches the visualiser, a new row is added to the end of the grid (essentially, the y-axis of the graph represents time). History graphs provide concise images of a system’s behaviour over time and are very helpful when, say, comparing the effects of different algorithms. Examples obtained from the Train algorithm are included and analysed in section 6.

3.4 Achievements

The generality and ease-of-adoption claims of the GCspy framework have been proven since it has been incorporated, so far, in three different Java virtual machines: Sun’s ResearchVM [29], Sun’s Java HotSpot virtual machine [24], and IBM’s Jikes RVM [1]. It has also visualised at least eight different garbage collectors³ using different drivers but the same general framework. The visualiser is written in the Java™ programming language using the Swing toolkit (its layout, when customised for the Train collector, is briefly described in section 5.5). There are two different implementations of the server: one written in C (incorporated in ResearchVM and the Java HotSpot virtual machine) and one written in Java (incorporated in Jikes RVM).

Scalability-wise, the framework has visualised a variety of heap sizes and it has been successfully tested with heaps up to 1GB with 128KB blocks; this required 8,192 tiles. Further, it was also tested using a 50MB heap with 512-byte blocks (this was the granularity of the card table of that Java virtual machine); this required 104,400 tiles. A very small tile size had to be used for the latter test, in order to fit all the tiles on the screen at the same time, and the response of the system was prohibitively slow. Nevertheless, the framework operated as expected.

Additionally, the trace files are also as compact as expected. For

³Different flavours of the standard mark-sweep, mark-compact, and semi-spaces collectors [11], the generational mostly-concurrent collector [18], and of course the Train collector, described in this paper.

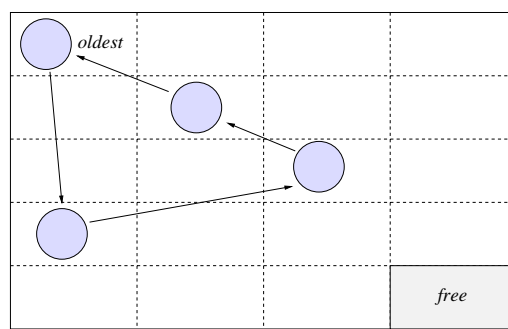


Figure 2: Garbage cycle in an incrementally-collected heap.

example, a large javac job (compiling all the standard Java libraries of JDK1.2.2 and generating around 4,500 class files), that uses a 139MB heap (with three spaces: 4MB semi-spaces young generation, 135MB mark-sweep old generation, and 70 free lists) and runs for 3 minutes 44 seconds (without the GCspy overhead), sends 4,560 GCspy transmissions, but generates only a 2.6MB compressed trace file.

4. THE TRAIN ALGORITHM

The Train algorithm is an incremental copying collection technique that allows a portion of the heap to be collected at a time. The basic idea is simple: choose a region and copy all the reachable objects out of it. However, the devil is in the details.

The problem with this naïve approach can be seen in figure 2. Suppose we simply divide the heap into fixed-sized regions and incrementally collect each region in turn. While we easily bound the amount of data we must copy at each collection step, we may not be able to collect certain large, unreachable data structures if these cannot be collapsed into a single region.

The idea of collecting memory with an incremental collection technique dates back to Bishop’s work on the ORLSA system [2]. Bishop’s thesis describes the design and implementation of an operating system to support large address spaces. He introduces the notion of short- and long-lived areas of memory and uses remembered sets to track references into as well as out of each area. His collector is able to collect arbitrarily-sized structures by migrating objects to areas containing references to those objects and by imposing no bounds on the sizes of areas. Once a given structure is isolated within a single area, its collection is straightforward.

The importance of Hudson and Moss’ collection technique [9] is that they are able to bound the size of the set of regions they collect at any one step and this ability gives them the hope of bounding pause times. The key insight they had is that if the generation is divided into many fixed-sized *regions*, these regions can be lexicographically ordered:

- **Cars.** Each region, or *car*, is logically linked into a list.
- **Trains.** The lists of cars, or *trains*, are then linearly ordered.

The order of the trains reflects the order in time of their creation from oldest to youngest. Cars are added as needed to a given train as objects are evacuated or placed into that train. Since each car must be collectable independently of the others, we associate a *remembered set* with each car to track the references into it from elsewhere in the same generation. Through careful placement and collection policies, the algorithm guarantees that we need only collect a bounded amount of memory at each step and that we can still collect large garbage structures.

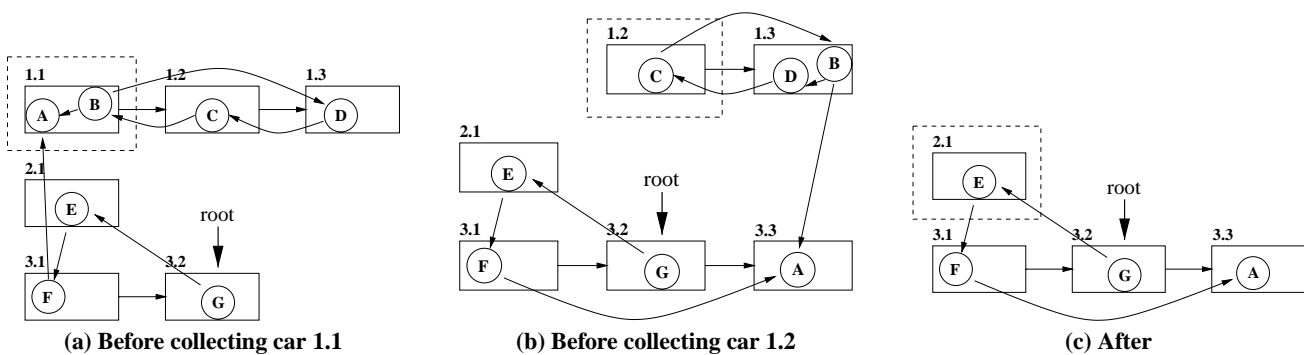


Figure 3: Example of two collection steps using the Train algorithm.

Collection commences by choosing some initial prefix of the set of logically-ordered cars, referred to as the *collection set*. By examining roots external to the generation, as well as those stored in the remembered sets of the cars in the collection set, collection proceeds by evacuating reachable objects to cars outside the collection set. To guarantee progress, we restrict the placement of objects:

- An object reachable only from other cars may be placed no higher than the highest train that contains a reference to that object,
- once a train becomes the oldest train, no newly-allocated objects may be placed in it, and
- no object may be evacuated to the oldest train if that object has references from outside that train.

Large garbage cycles are detected and recycled by noting when the oldest train has no references from outside the generation or from any younger train. In such cases, we know that the entire oldest train is no longer reachable and we can immediately recycle all the cars in it.

To help illustrate how the collection proceeds using the Train algorithm, consider figure 3. Initially, we have three trains. In the first collection step, we collect the reachable objects in the oldest car, car 1.1. Because object A is reachable from object F and object F has the highest reference to A, we copy A to some car in the third train. Because B is only referenced by another object in the same train, however, we evacuate B to a car in the oldest train. When we go to collect car 1.2, in the second collection step, we see that there are no references from elsewhere to any objects in the oldest train and so we are able to immediately reclaim both cars 1.2 and 1.3.

Hudson and Moss make the claim that the Train algorithm is a real-time collection technique because it bounds the amount of memory collected at any step. By collecting the cars in a specific order, the algorithm only needs to track references from younger cars to older cars, greatly reducing the cost of remembered-set maintenance. Further, because trains are processed in roughly age-order, objects in cars are given time to become unreachable before they are first subject to collection. Since copying collection only performs work proportional to the set of reachable objects, this strategy increases the efficiency of collection. Finally, by using a relocating collection technique, the Train algorithm avoids some of the fragmentation problems suffered by non-moving collectors.

4.1 Policies and Challenges

There are a number of aspects of the Train algorithm, however, that undermine these claims and make collectors based on the algorithm somewhat difficult to tune. For example, while the size

of each collection set is easily bounded, the size of the remembered sets, associated with cars in a given collection set, may be proportional to the size of the entire generation. Further, we have measured that the cost of a single insertion into a remembered set is roughly as expensive as copying the typical 8–10 word object. This means that maintenance of remembered sets imposes a significant cost on the application’s execution time. Finally, it is well known that the cost of copying objects that are part of a structure may be quadratic in the number of cars over which the structure is distributed [9, 21].

Some typical policy decisions include determining:

- how many cars to collect at each step,
- where to place objects in cars and trains,
- how to handle highly-referenced, or *popular* objects, and
- how to reduce the cost of insertions into, and scanning of, remembered sets.

These policy decisions also play off each other. For example, better placement of objects may allow groups of objects to be collected in a more timely fashion and reduce the overhead of floating garbage. Better placement may also reduce the number of references recorded in remembered sets. However, the same placement policy may optimise one of these effects at the expense of the other. Likewise, there are pressures to collect as little as possible at each step (e.g., allowing more objects to die, smaller pause times), but there are also pressures to collect more aggressively (e.g., keeping up with the rate of allocation into the generation, reclaiming remembered set structures). The hope is that by visualising the effects of these policy choices, we can obtain a better appreciation for their impact.

4.2 Collector Implementation

We have implemented a collector based on the Train algorithm in Sun’s ResearchVM [29]. It operates as the old generation of a generational memory system. Its implementation assumes that most allocation to the old generation will take place during young generation collections that move objects out of the young to the old generation; this is referred to as *promotion*. In addition, the application may directly allocate some objects in the old generation, if these objects are sufficiently large. In this collector, we use the simple placement policy that all newly-promoted or allocated objects are placed in the youngest train.

The Train collector in the ResearchVM is an early snapshot of a more sophisticated implementation of the Train algorithm [6]. However, because the latter is not yet publicly available, we chose to instrument the existing collector as it is. While this means that

we cannot examine, for example, very many placement policies, it does include some interesting optimisations. We mentioned, for example, that remembered-set maintenance may be costly. To help address this issue, our collector employs two strategies. First, it dynamically detects and isolates popular objects in cars by themselves. Second, it employs two techniques for increasing the *capacity* of remembered sets when they expand:

- **Increasing size.** A remembered set is expanded by a constant factor up until some threshold (say, 4,096 entries) is reached. Thereafter, we preferentially *coarsen* the entries falling back on increasing the size only if coarsening does not sufficiently decrease the load factor.
- **Coarsening.** A remembered set is coarsened by rehashing all its entries into a fresh remembered set whose size is the same as the original. Before the entries are rehashed, we first mask off some constant number of bits causing each entry to represent a larger area of memory, possibly containing references of interest.

Coarsening aids both in reducing the amount of memory used in representing remembered sets and also in reducing the cost of insertions into the remembered sets. One reason for its success lies in a kind of locality property: if a car's objects are referenced from some portion of the generation, then there are often references from objects near that area into the same car. We are currently investigating methods for using visualisation to better understand this property.

Given these two dimensions, we will use *size* to refer to the total number of slots in a remembered set and *capacity* to refer to the product of the size and the level of coarsening of the remembered set.

5. TRAIN VISUALISATION

The driver for the Train collector is the most complex GCspy driver implemented so far. It has been customised for and incorporated into the Train collector of Sun's ResearchVM, described in section 4. The collector is visualised with three spaces comprising forty separate streams. The three spaces are: one for the region information, one for the car information, and one for the train information. We felt that all three spaces were necessary, as each shows a different aspect of the collector. The region space provides a physical view of the contiguous address space that comprises the Train generation. The car space provides a logical view of the same generation. The train space is the smallest of the three and essentially provides summary information over the car space. They are described in more detail in sections 5.1 to 5.3.

5.1 Region Space

Each tile in this space represents a 64KB memory region. The tiles appear in address order, providing a physical view of the Train generation. The visualisation clearly indicates which tiles correspond to regions that are currently not allocated. There are nine streams associated with the region space.

- **Used Space.** The amount of space allocated for objects in that region.
- **Card Table State.** The summary of the state of the cards in the card table that correspond to the region.
- **Train ID / Car ID.** The train and car IDs of the car that corresponds to the region.
- **Region Type.** It indicates whether the region is a *standard* one, an *oversized* one (in which case it is one of several contiguous regions that correspond to a single car), or one for *popular* objects.

- **Allocation Info.** It indicates in which regions allocation has taken place. There are four types of allocation and there is a stream for each of them: *direct* (for large objects that do not fit in the young generation and are allocated directly to the old one), *promotion* (for objects that are promoted from the young generation), *evacuation* (for objects that are copied from other cars during a Train collection cycle), and *weak evacuation* (for objects that are copied from other cars as a result of weak-reference processing).

Several aspects of the behaviour of the Train algorithm can be observed from different views of the region space: the presence of any fragmentation in the generation (used space view), pointer mutation patterns (card table view), the way that cars and trains are spread around the generation (train / car ID views), allocation patterns (allocation views), etc.

5.2 Car Space

Each tile in this space represents a single car. The tiles are first ordered according to train ID and then according to car ID. This groups all the cars of the same train together and there are visual separators that indicate train boundaries. Because of the way the IDs are generated, the tile order also reflects the age of the trains and cars, i.e., older trains are towards the front of the space, younger ones towards the end; similarly with cars within each train. The car space is the most elaborate of the three and has seventeen streams associated with it.

- **Used Space.** The amount of space allocated for objects in that car.
- **Remembered Set Info.** Details on the remembered set of that car, split into five streams: remembered set coarseness, current number of entries, current capacity, maximum size, and whether there are references to that car from a higher (i.e., younger) train.
- **Train ID / Car ID.** The train and car IDs of that car.
- **Car Type.** It indicates whether the car is a *standard* one, an *oversized* one, or a *popular* one.
- **Size.** The amount of space allocated for that car.
- **Objects.** Current number of objects in that car.
- **Collection Set Info.** There are two streams. One indicates whether the car is in the current collection set, the other how much space the car contributes towards the collection set.
- **Allocation Info.** As described in section 5.1.

The car space is an alternative view of the generation, with similarities to the region space (as most of the time there is a one-to-one correspondance between a car and a region, with the exception of oversized and popular cars), but organised in a logical rather than a physical order. Because of this, there are different aspects of the behaviour of the Train algorithm that can be observed from this space: distribution of used space across cars of the same train (used space view), behaviour of the remembered sets (remembered set views), behaviour of the heuristic that chooses which cars to collect (collection set views), allocation patterns across cars (allocation views), etc.

5.3 Train Space

Each tile in this space represents a single train and essentially summarises the information per train presented in the car space. There are fourteen streams associated with it and most of them are similar to the ones available in the car space.

- **Used Space.** The amount of space allocated for objects in that train.

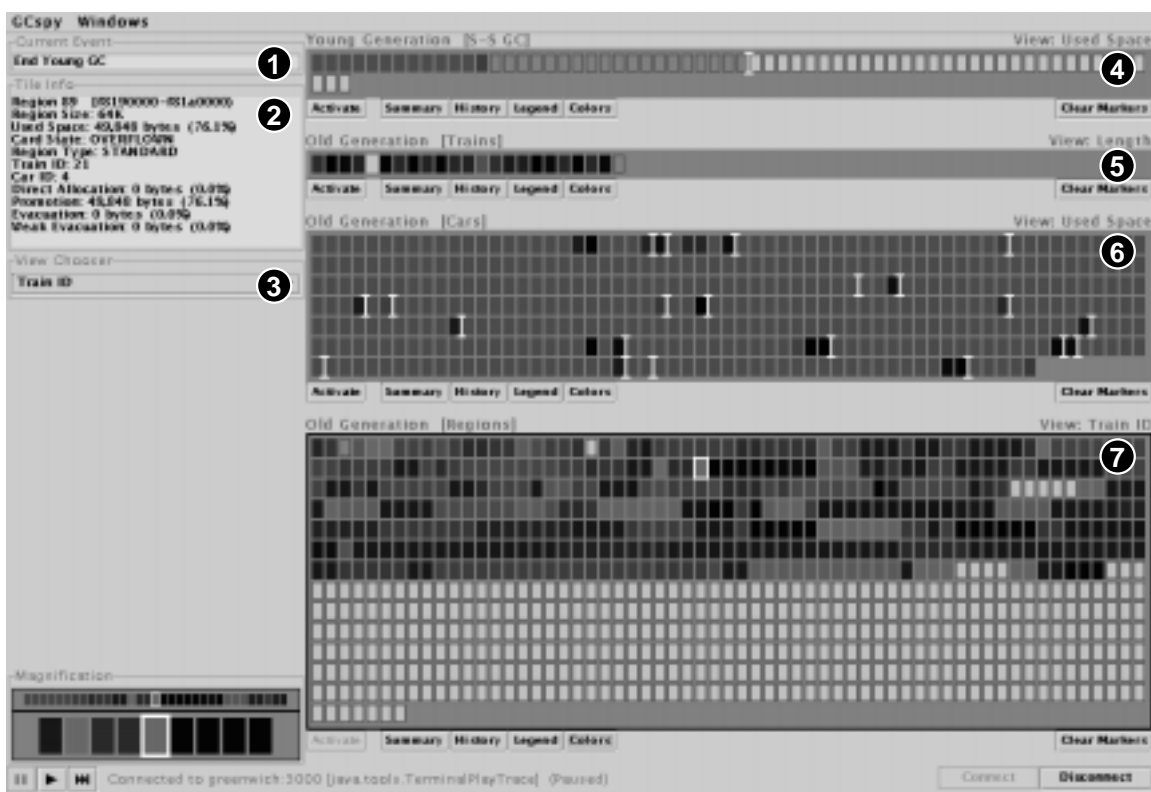


Figure 4: GCspy main window.

- **Remembered Set Info.** As described in section 5.2, but only the streams that represent remembered set length and whether the train has higher (i.e., younger) references.
- **Train ID.** The ID of that train.
- **Length.** The number of cars that belong to that train.
- **Size.** The amount of space allocated for all the cars of that train.
- **Objects / Popular Objects.** Number of objects and popular objects in that train.
- **Collection Set Info.** As described in section 5.2.
- **Allocation Info.** As described in section 5.1.

The train space can provide insights similar to the ones obtained from the car space, but is more useful on some occasions as the visualisations it provides are presented in a more concise way, e.g., the length view of the train space can indicate more efficiently which trains are currently very long, compared to trying to obtain the same information from the car space.

5.4 Data Collection

The state of the heap is collected and transmitted to the visualiser at four different points (these are the four GCspy events defined in the Java virtual machine for the Train collector).

1. **Young Collection Start.** It shows the effects of the mutator on the heap (mainly allocation to the young generation, changes to the card table, and possibly direct allocation to the old generation).
2. **Young Collection End.** It shows the effects of the young collection on the heap (mainly allocation to the old generation,

processing of the card table, and updates to the remembered sets).

3. **Train Collection Start.** It shows which cars have been chosen to be collected (i.e., the collection set).
4. **Train Collection End.** It shows the effects of a Train collection cycle on the heap.

Notice that there are some streams for which data will only be available at some of the above events and not all of them. For example, promotion information (i.e., allocation of objects that have just been evacuated from the young generation) will only be available at the end of a young collection. Such streams have default values for the remaining events.

5.5 User Interface

Figure 4 shows the main GCspy window, when customised for the Train algorithm. The seven most important areas of the interface are described below (the numbers correspond to the numbered areas in the figure).

- ❶ **Current Event.** It shows which event inside the server generated the transmission that is visualised in the interface. In the example, this event is the end of a young collection.
- ❷ **Tile Info.** At any time, one of the spaces is 'active'; this is denoted by a black frame around it (in the figure, the region space is the active one). A tile on the active space can be selected (denoted by the white frame around it) and textual information about the entity it represents is displayed in this text area.
- ❸ **View Chooser.** It allows the user to choose the view of the active space. A label at the top-right corner of the space also

indicates the selected view. Currently, there is an one-to-one correspondence between views and streams defined on each space.

- ④ **Young Generation Space.** It illustrates the state of the young generation, which is managed by a semi-spaces collector [11]. This space is managed by a different driver to the Train driver (which manages the next three spaces). The view shown in the figure is the amount of space allocated for objects per tile. The white vertical marker separates the two semi-spaces; the visualisation shows that the left semi-space is the active one, being about 40% full.
- ⑤ **Train Space.** This is the first space managed by the Train driver and is described in section 5.3. The view shown in the figure is the length of each train, with brighter tiles denoting longer trains. In the figure, the fifth train is the longest.
- ⑥ **Car Space.** This is the second space managed by the Train driver and is described in section 5.2. Vertical white separators group together the cars that belong to the same train. The view shown in the figure is the amount of space per car allocated for objects, with darker tiles denoting low utilisation. Notice that cars with low utilisation appear mostly towards the end of each train, as expected.
- ⑦ **Region Space.** This is the third space managed by the Train driver and is described in section 5.1. The view shown in the figure is the train ID of each region, with brighter tiles denoting older trains and darker tiles denoting younger trains. The light grey tiles with a light frame around them denote regions that are not currently allocated. Notice that the number of allocated tiles in this space is roughly⁴ the same number as the tiles in the car space; however, they are ordered in a different manner (physically in this case, logically in the case of the car space).

Figure 4 shows one possible configuration of the three Train spaces. A number of alternative views are also available over them, as described in sections 5.1 to 5.3 (fourteen for trains, seventeen for cars, and nine for regions), which can visualise different aspects of the behaviour of the Train algorithm.

In addition, figure 10 in appendix C shows examples of the tile information and figure 11 shows examples of the summary information (i.e., aggregate information for all the tiles of the space, as described in section 3.1) that are available on the three train spaces. More details on the operation of the GCspy interface are discussed elsewhere [19].

5.6 Achievements

The Train driver has been operating robustly for some time and has visualised a number of different applications, giving us unique insights into the operation of the Train algorithm (see section 6 for more information on this). The largest heap that has been visualised was 0.5GB (440MB of that was used) and had a maximum of 245 trains, 6,836 cars, and 8,192 regions. All the tiles could fit on the screen at the same time, albeit with a smaller tile size, compared to the one illustrated in figure 4, and using a large 24"-wide monitor. The trace file obtained from this application is just under 40MB compressed, which is larger than the ones generated by all other collectors tried so far; this is understandable, however, given the amount of information required to visualise the Train collector.

During the development of the Train driver, which is by far the most elaborate one written so far, the GCspy framework proved

⁴It is not exactly the same due to the presence of large cars (which will comprise several regions) and popular cars (several of which will comprise a single region).

very versatile, easily customisable, and could accommodate the complicated visualisations required. Admittedly, a small number of extensions were made to the framework to enhance the detail of the visualisations. However, none were Train-specific and were all applied in a generic manner so that other drivers can also take advantage of them.

6. INSIGHTS

Having instrumented our Train collector so that it can be visualised with GCspy, we set out to see what we could learn from the generated visualisations. The lessons, so far, fall into two broad categories: insights into the applications themselves and insights about choices in policies inside the collector. The hope is that by viewing applications, particularly ones that are known to be problematic in terms of their pause-time or space-requirement behaviour, we can gain insights into how to better tune the behaviour of a collector based on the Train algorithm.

6.1 Applications

For the purposes of illustrating the benefits of visualisation, we have chosen to use five applications:

- **DB:** The `_209_db` benchmark, part of the standard SPECjvm98 benchmark suite.
- **Reptile:** The kernel of an Escher tiling package written in Haskell and transformed into Java bytecode by David Wake-ling [28].
- **GCold:** A benchmark written to evaluate the performance of incremental GCs. It creates disjoint tree structures and performs operations on them, creating short- and long-lived objects, while rendering some others garbage. It is described in more detail elsewhere [18].
- **Pretzel:** A benchmark that constructs several identical boxes of 'pretzels'. A pretzel is a binary tree with a thin left and right arm and lots of pointers back to the top node; each box has a number of pretzels of differing heights, from a minimum to a maximum value in a geometric progression.
- **Paraffins:** A benchmark that calculates the number of kinds of paraffin molecules that may be constructed given some number of carbon atoms. For the experiments in this paper, we chose 23 for this parameter.

These applications were chosen because they either exhibit simple and predictable behaviour and hence can be used as examples of how the resulting visualisations can be interpreted (DB), or they exhibit behaviour that reveals interesting patterns in the operation of the collector (Reptile and GCold), or they are known to be problematic for collectors based on the Train algorithm (Pretzel and Paraffins).

6.2 Examples of Simple History Graphs

As described in section 3.3, GCspy can generate history graphs that illustrate how the state of the heap changes over time. However, from past experience, these graphs are not immediately intuitive for people who see them for the first time. Because of this, we have included two simple ones in figure 5 to familiarise the reader with the way they are interpreted. Both have been obtained from the DB benchmark.

The left-hand graph in figure 5 visualises the amount of used space (i.e., amount of space allocated to objects) in the Train generation. The space visualised is the region space (see section 5.1). The horizontal axis represents a physical view of Train generation (low addresses to the left, high addresses to the right) and the vertical axis represents time (earlier in time is at the top of the graph,

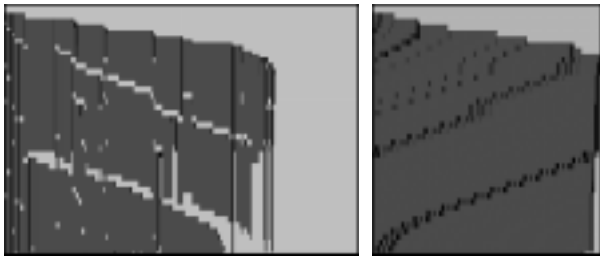


Figure 5: History graphs showing the used space attributes of the region (left) and car (right) spaces when running the DB benchmark.

later is at the bottom). A light grey color corresponds to regions in the generation that are not allocated and do not contain any objects (referred to as unused), dark grey corresponds to regions with a high amount of used space, black corresponds to regions that have a low amount of used space⁵. What this graph shows is that unused regions do appear throughout the Train generation. This is because, when cars are collected and live objects are evacuated from them, the space they were taking up becomes unused. However, the collector tries to first re-allocate unused regions that are located low into the generation. This explains why all used regions in the generation are grouped mostly close together and there are no signs of heavy fragmentation. Finally, the vertical black lines that appear throughout the graph denote regions that are low in used space and remain in that state until the end of the execution.

The right-hand graph in figure 5 is taken from the same execution of the DB benchmark as the left-hand one, but it shows the car space instead (see section 5.2). Again, the vertical axis denotes time. However, in this case the horizontal axis represents cars in the generation, ordered according to their age, oldest to the left, youngest to the right (as described in section 5.2). The graph again shows the percentage of used space in the cars. Light grey denotes cars that are not allocated; they are grouped to the right of the youngest car, due to the way the cars are sorted (oldest is always first). The meaning of dark grey and black colors is as described above. What this graph shows first is that the number of cars in the generation sharply increases in the beginning of the execution and then slowly but steadily decreases until the end. Further, most allocated cars are pretty full (they appear as dark grey) and only a small percentage of them are almost empty (black), again indicating the absence of fragmentation. An interesting observation, which is also evident in other history graphs later on, is that some black lines have a bottom-left direction. These correspond to almost empty cars that become older during execution. It is always the oldest cars that are collected at every Train collection cycle (i.e., the left-most ones in the graph), hence all other cars seem to be ‘moving left’ during execution, as they become older. Finally, notice that one of these lines seems to ‘wrap round’ once it reaches the left edge of the graph. This is most likely because those cars contain small, popular objects and, when they are collected (i.e., when they become the oldest and reach the left edge of the graph), the objects are found to still be popular, and so they are relinked to the end of the youngest trains containing references to them.

The rest of the history graphs included in this section show ei-

⁵These color choices might seem inappropriate and counter-intuitive. However, the graph looks a lot better in color (the dark grey is in fact red). The reader is encouraged to look at the color versions of this and all other history graphs included in this paper, which are available here: <http://www.dcs.gla.ac.uk/~tony/gcs.py.www/ismm2002-figures/>

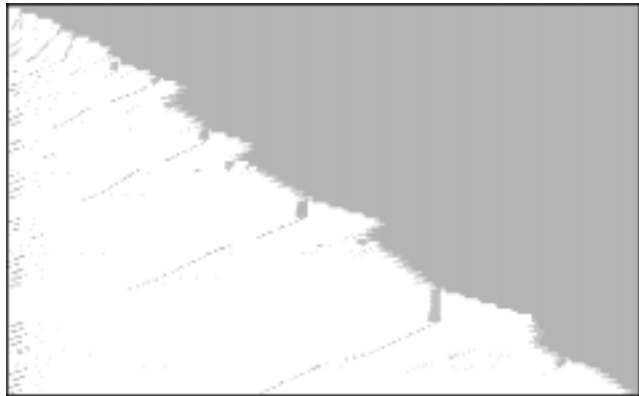


Figure 6: History graph showing the evacuation attribute of the car space when running the Reptile benchmark.

ther the behaviour over time of different attributes of the car space, as described above, or the train space (which is similar to the car space, however the horizontal axis denotes trains instead of cars).

6.3 The Problem of Long Trains

While visualising the Train algorithm, the most surprising aspect that struck us was how frequently long-lived data would cluster forming a few very large trains of cars. This is illustrated in figures 6 and 7.

Figure 6 shows the pattern of evacuation in the car space (i.e., object-copying caused by Train collection cycles) for the Reptile benchmark. Again, in this graph the horizontal axis denotes cars in the generation and the vertical axis denotes time. For each car that is used, a white shade indicates that no evacuation to it took place during that particular collection event, whereas a darker shade indicates that evacuation to it did take place. From the figure we can see that, periodically, objects are evacuated to cars at the end of the same train (these are the horizontally-barred patterns to the far right). Then, as these cars move in toward the oldest set of cars (i.e., towards the left edge of the graph), objects continue to be clustered with them.

We obtain similar insights by examining the behaviour of the GCold benchmark in figure 7. The three history graphs in that figure represent the behaviour of trains over time (i.e., in this case, each horizontal line depicts the state of all trains, ordered according to their age, oldest to the left, youngest to the right). The left-hand graph shows the evacuation behaviour in the GCold benchmark over time (the colors are the same as in figure 6). The middle graph distinguishes trains with references from younger trains (in white) and trains lacking such references (in black). Finally, the right-hand graph shows the relative lengths of trains: the darker the color, the shorter the train. From figure 7 we can see that there is a single data structure that, over time, causes an even larger number of objects to be clustered with it (note the longer and longer periods required to collect this train when it becomes the oldest in the generation). This observation is obvious by comparing the left-hand (evacuation pattern) and right-hand (train length) graphs: notice that most of the evacuation activity is to the single longest train. From the middle graph we see that, as this train moves toward the collection set at the left-most side of the graph, the relocation policy results in relatively few trains with younger references. Then, once this train begins to be collected again, we see that a few key objects are relocated and immediately cause all older trains to have references from younger trains.

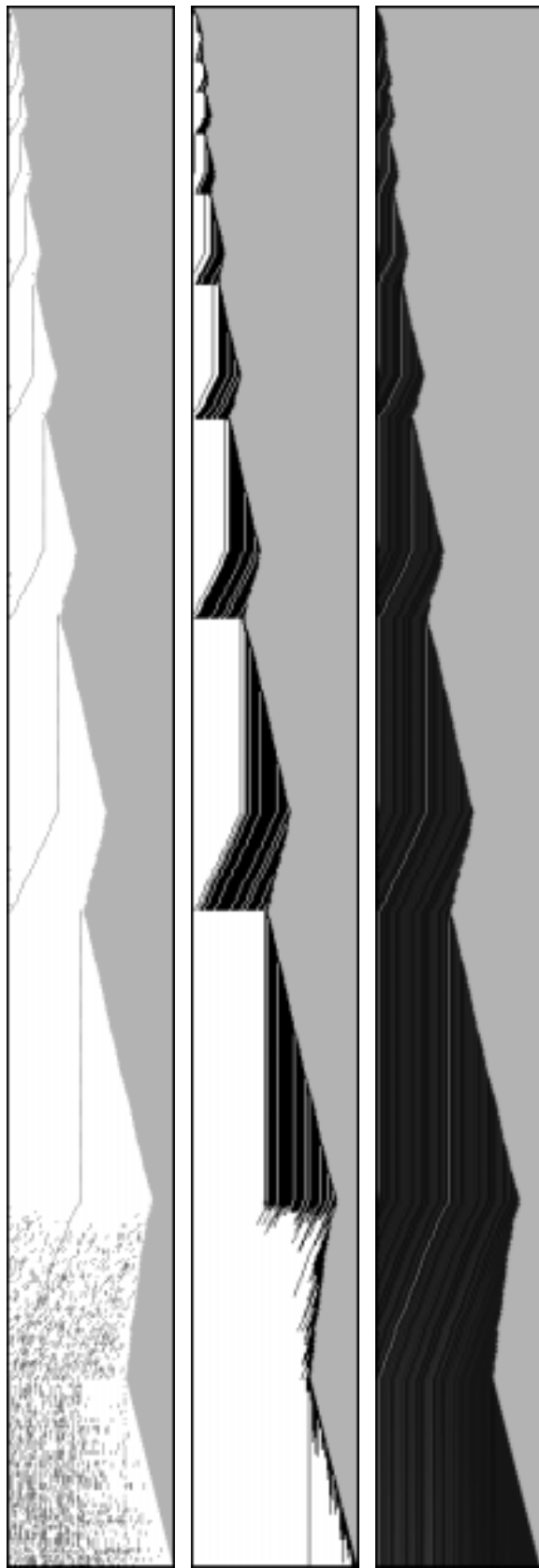


Figure 7: History graphs showing (from left to right) the evacuation, higher references, and the length attributes of the train space when running the GCold benchmark.

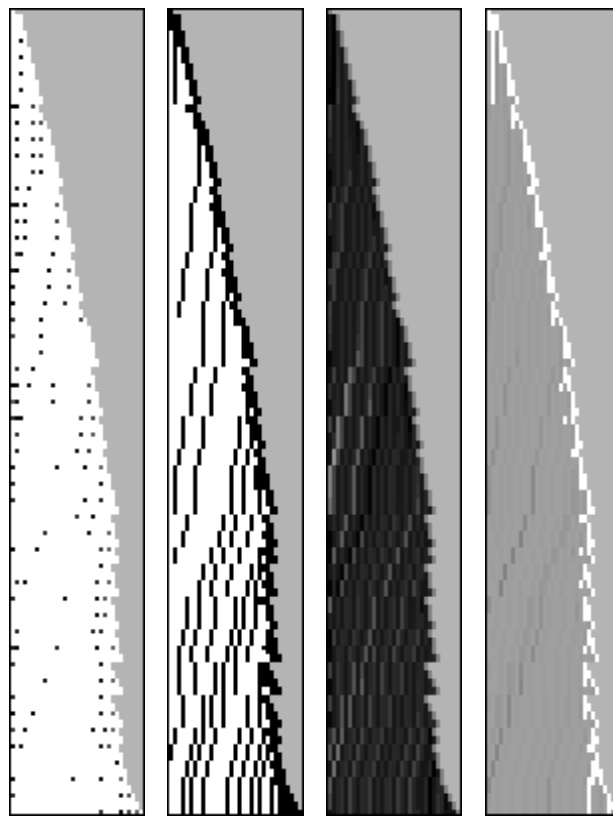


Figure 8: History graphs showing (from left to right) the evacuation, higher references, length, and remembered set length attributes of the train space running the Pretzel benchmark.

We have also seen this pattern repeated in history graphs obtained from other applications. What this tells us is that policies that depend on the lengths of trains may be difficult to apply in a collector based on the Train algorithm. For example, several of the policies investigated by Seligmann and Garup [21] rely on the fact that trains are typically short. In particular, they track a running estimate of the ratio of garbage encountered to aid in determining how aggressively to collect. A low garbage ratio indicates an overly aggressive rate of collection and, in such cases, the rate is reduced. Likewise, a high garbage rate indicates that more memory can be collected, while maintaining good pause times. In the presence of long trains of long-lived data, such strategies fail twice: first, by not recognising that however slowly one collects cars of this kind of train, the ratio of garbage is unlikely to increase; and, second, by only adjusting the running estimate of the garbage ratio once the train is completely collected.

A final note on figure 7 is the change in behaviour towards the end of the benchmark execution. GCold initially builds a single very large data structure. Once it has been constructed, it then processes parts of it. This explains the less canonical behaviour that is clear at the bottom of the evacuation and higher references graphs.

6.4 Keeping a Reserve

Another pattern of behaviour relates to how we allocate popular regions. Recall that popular regions are subdivided into fixed-sized subregions and these subregions each hold at most one single popular object. By allocating these popular regions on demand, they

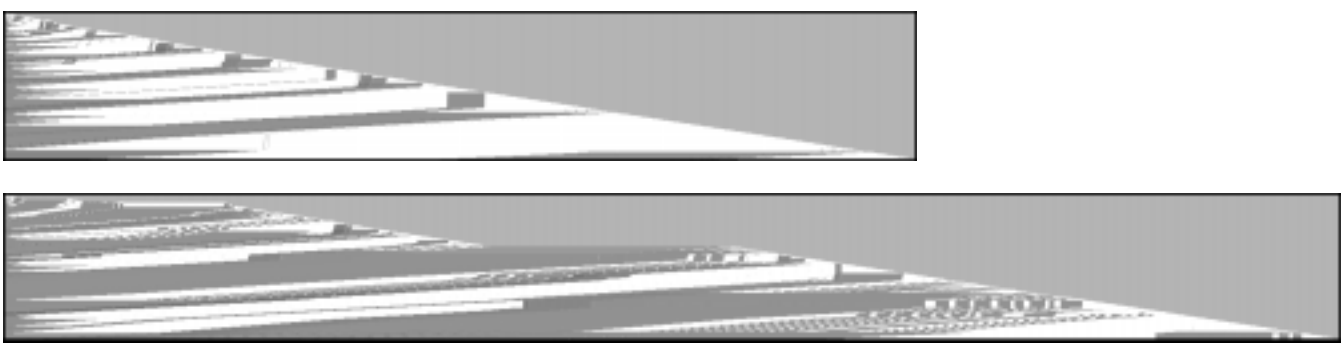


Figure 9: History graphs showing the remembered set length attribute of the car space when running the Paraffins benchmark, showing the difference between dealing (bottom) and not dealing (top) popular objects specially.

are typically distributed throughout the generation. This placement policy impedes our ability to contract the generation when space usage falls because we only support contraction at the upper end of its address range.

Two possible solutions to this are either to support ‘holes’ in the middle of the generation or to place these regions together as much as possible and to keep them close to the start of the address space. Based on the fact that there are typically few such popular regions—borne out by visualisation of a number of test applications—we have decided to implement a reserved set of regions at the start of the address space. Here, we have two options: make the generation grow at both ends as needed or allow the reserve to be used for normal allocation during periods when the Train generation is close to exhaustion.

6.5 The Effects of Churning

One application that causes particular problems for the Train algorithm is the Pretzel benchmark. This benchmark repeatedly constructs a number of cyclic data structures of various sizes, only to then kill all references to them. Repeating this strategy forces the Train collector to *churn* through many objects before discovering that they are part of a garbage structure or have been co-located with other mutually referring objects in some train. Churning is an inefficient activity, since it copies objects, when they are parts of the collection set, to locations that will soon be part of the collection set again and leads to the worst-case quadratic-cost copying behaviour, mentioned in section 4.

Figure 8 shows four history graphs depicting changes on the state of trains during a run of Pretzel. The first of these graphs shows the pattern of evacuation: clearly, a significant amount of evacuation is to relatively old trains, located to the left of the history graph, which will be part of the collection set soon. The second, representing which trains have references from younger ones (white for yes, black for no), shows the not-so-surprising fact that, if one train has many cyclic data structures, then many trains will have references from younger trains, at least until most of the cyclic structures have been co-located in particular trains. The third history, showing the relative length of trains, shows a clear correlation between the length of trains and the likelihood that a given train has no references from higher trains. Finally, the last history depicts how many entries are stored in the remembered set of each train (a grey shade represents a totally empty remembered set, a light grey shade represents a short remembered set, and a dark shade represents a full one). By comparing this graph to the previous one it is clear that the longer trains tend to have longer remembered sets. For our current purposes, however, the fact that so much evacuation

happens close to the oldest set of trains indicates a relatively high rate of churning.

This pattern of churning can be seen in most applications, but not usually to the same extent as in Pretzel. Such a pattern also appears periodically in the graph of figure 6 (where evacuations take place towards the left edge of the graph) and, to a lesser extent, in the evacuation graph in figure 7.

6.6 Comparing Policies for Remembered Sets

Finally, as we mentioned in section 4, our Train generation relies on a number of techniques for reducing the cost of remembered set maintenance, both in terms of space and time. For the first time we have been able to visualise the impact of these policy choices on applications like the Paraffins benchmark.

Paraffins is a good test of any collector based on the Train algorithm. It computes the number of possible paraffin molecules using graph enumeration. Beginning with the simplest component (the radical CH_3), it constructs the new components through combinations of existing ones. The result is, for example, that one object (the one representing CH_3 , in fact) ends up being referred to by half of all references in the generation. The next two smallest components share half the remaining references and so on. At the other end, the increasingly large arrays of references are allocated to aggregate the sets of components. Ultimately, arrays as large as 42MB are allocated before the result is produced for an input of 23 carbons. These extremes place an enormous burden on the management of remembered sets.

Figure 9 shows two history graphs of the car space. The top graph shows the number of entries per car in a Train generation that supports neither the isolation of popular objects nor the notion of coarsening. In both graphs, the white regions represent cars with no remember set information at all, light grey regions represent short remembered sets, dark regions represent full ones. Two observations can be made. First, the relocation strategies work well for this application in that many cars have no remembered set information until they are close to being collected. Second, there are a significant number of fairly large remembered sets (this is more obvious when viewing the graph in its original size). This confirms earlier studies where we were able to correlate these large remembered sets (and their processing) with some surprisingly long pause times.

The bottom graph shows the same information for Paraffins running with a Train generation that does support the optimisations mentioned above. First, it is obvious from the figure that there are considerably more cars, many being popular, compared to the top graph. Although it is again clearer when viewing it in its origi-

nal size, the graph shows that the average number of entries per car is much lower and much more evenly distributed than before. These facts help explain why these optimisations help smooth out the pause time behaviour of the collector and confirm the results of earlier investigations.

6.7 Evaluation

We believe that these results demonstrate the usefulness of applying the GCspy framework to the study of the Train collector. It has helped identify desirable properties of policies (for example: “do not dismiss the existence of large trains”), motivated specific changes in other policies, such as how and where popular regions are allocated, and given insight into why certain applications are problematic for the Train algorithm, as well as how certain policies can eliminate classes of problems (such as popular objects).

7. CONCLUSIONS AND FUTURE WORK

This paper has introduced GCspy, an extensible heap visualisation framework, and the way it was adapted to visualise a very complex, incremental garbage collector, based on the Train algorithm. It also described the effectiveness of the generated visualisations in providing useful insights into the behaviour of the collector.

The paper includes two original contributions. The first and more specific one is the model of visualisation for the Train algorithm (described in section 5), which provides very concise snapshots of its state, as well as informative graphs of its behaviour over time. The second contribution is that it advocates the use of a specialised heap visualiser to obtain insights into the operation of a garbage collector. This proved very effective when analysing the Train algorithm (as described in section 6) and we are confident it will be equally effective in the case of other collectors too.

The GCspy framework has been developed relatively recently and it is still growing, with new facilities being added to it regularly. Some of them, and how they can improve the visualisations of the Train algorithm, are described below.

General visualisation improvements include allowing simple user-defined expressions to map stream values to color shades and visualising several streams at the same time by splitting the tiles into subtiles and coloring each subtile according to a different stream. Additionally, the ‘standard’ tile view of a space (as illustrated in figure 4) can be replaced with other alternatives, e.g., line or bar charts and scatter plots. As the raw data values for each stream are available at the visualiser, it will be easy to just present them in a different way. One can envisage an open architecture that allows the easy incorporation of different visualisation ‘plug-ins’.

Another interesting facility is for the visualiser to automatically highlight tiles in all spaces that are associated with the currently-selected tile. In the case of the Train collector, this will allow, for example, all tiles in the car and region spaces that belong to the same train to be automatically highlighted, when a train is selected in the train space. This association can be very easily done, based on the values of a particular stream per space (e.g., notice that all three train spaces contain a train ID stream, as described in section 5).

There are also a number of improvements that can be applied to the history graphs. It will be useful to automatically join history graphs of different spaces in order to correlate their behaviour (e.g., imagine obtaining a single graph that contains the state of both the train and the car spaces). Additionally, plotting a line chart of the summary value, available for each stream, at one side of a history graph might also increase our understanding of the information the graph contains. Consider, for example, the value of plotting the total amount of objects evacuated in relation to figure 6. Further, post-

processing of the history graphs with 2D filters [5] might provide additional information about trends on them (e.g., edge-detection could give a clearer picture of the shape of edges in the graph).

It may also be useful to augment histories with auxiliary data, such as the times for particular phases of collection. Consider, for example, the value of plotting the time to process remembered sets in relation to the histories in figure 9.

Finally, GCspy can only generate a history graph that shows the behaviour of a single stream over time (e.g., figure 6 shows the behaviour of only the evacuation stream). It is, in fact, desirable to plot several streams on the same graph in order to correlate their behaviour. In the case of the Train collector, it will be interesting to plot on the same history graph all four streams that correspond to allocation activity (e.g., direct allocation, promotion from the young generation, evacuation during a Train collection cycle, and evacuation during the processing of weak references), in order to identify patterns in their behaviour and, based on such patterns, to possibly improve the allocation policies.

APPENDIX

A. ACKNOWLEDGEMENTS

This work is supported by EPSRC grant GR/R57140. Tony Printezis was funded through Sun Microsystems’ External Research Program. Richard Jones has given considerable input into the development of GCspy and incorporated it into Jikes RVM. The authors are grateful to Huw Evans, Peter Dickman, and the anonymous referees who gave valuable feedback on this paper.

Sun, Sun Microsystems, the Sun Logo, Solaris, Java, Java HotSpot, Java 2 SDK, and JDK are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

B. REFERENCES

- [1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Shepherd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of OOPSLA ’99*, pages 314–324, Denver, Colorado, USA, November 1999.
- [2] P. B. Bishop. Computer Systems with a Very Large Address Space and Garbage Collection. Technical Report MIT/LCS/TR-178, MIT Laboratory for Computer Science, MA, USA, 1977.
- [3] L. Cardelli, J. Donahue, L. Glassman, M. J. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report 52, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, September 1989. Revised.
- [4] W. De Pauw and G. Sevitski. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency — Practice and Experience*, 12:1431–1454, 2000.
- [5] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics — Principles and Practice*. Addison-Wesley, November 1993. Second Edition.
- [6] A. Garthwaite. *Making the Trains Run On Time*. PhD thesis, University of Pennsylvania, 2002. *In Preparation*.
- [7] Geodesic. Great Circle. <http://www.geodesic.com/solutions/> [February 7, 2002].
- [8] R. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage Collecting the World: One Car at a Time. In *Proceedings of OOPSLA ’97*, pages 162–175, Atlanta, GA, USA, October 1997.
- [9] R. L. Hudson and J. E. B. Moss. Incremental Garbage Collection of Mature Objects. In Y. Bekkers and J. Cohen, editors, *Proceedings of the First International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 388–403, St Malo, France, September 1992. Springer-Verlag.
- [10] IBM Research. Jinsight. <http://http://www.research.ibm.com/jinsight/> [November 12, 2001].
- [11] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd, 1996. With a chapter on Distributed Garbage Collection by R. Lins.

- [12] G. N. C. Kirby and R. Morrison. OCB: An Object/Class Browser for Java. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, pages 89–105, Half Moon Bay, CA, USA, August 1997.
- [13] H. Lieberman and C. E. Hewitt. A Real-Time Garbage Collector based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.
- [14] D. A. Moon. Garbage Collection in a Large Lisp System. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246. ACM Press, August 1984.
- [15] J. E. B. Moss, D. S. Munro, and R. L. Hudson. PMOS: a Complete and Coarse-Grained Incremental Garbage Collector for Persistent Object Stores. In R. C. H. Connor and S. Nettles, editors, *Persistent Object Systems: Principles and Practice — Proceedings of the Seventh International Workshop on Persistent Object Systems (POS7)*, pages 140–150, Cape May, New Jersey, USA, May 1996. Book was published in 1997.
- [16] D. S. Munro, A. L. Brown, R. Morrison, and J. E. B. Moss. Incremental Garbage Collection of a Persistent Object Store using PMOS. In R. Morrison, M. J. Jordan, and M. P. Atkinson, editors, *Advances in Persistent Object Systems — Proceedings of The Eighth International Workshop on Persistent Object Systems (POS8) and The Third International Workshop on Persistence and Java (PJW3)*, pages 78–91, Tiburon, CA, USA, August 1998.
- [17] ParaSoft. Inuse.
<http://www.parasoft.com/products/inuse/index.htm>
[February 7, 2002].
- [18] T. Printezis and D. Detlefs. A Generational Mostly-Concurrent Garbage Collector. In T. Hosking, editor, *Proceedings of the 2000 International Symposium on Memory Management*, pages 143–154, Minneapolis, MN, USA, October 2000. ACM Press.
- [19] T. Printezis and R. E. Jones. GCspy: An Adaptable Heap Visualisation Framework. Technical Report TR-2002-104, Department of Computing Science, University of Glasgow, Scotland, March 2002.
- [20] K. Russell and L. Bak. The HotSpot™ Serviceability Agent: An Out-of-Process High-Level Debugger for a Java™ Virtual Machine. In *Proceedings of the Usenix Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 117–126, Monterey, CA, USA, April 2001.
- [21] J. Seligmann and S. Grarup. Incremental Mature Garbage Collection using the Train Algorithm. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 235–252. Springer-Verlag, August 1995.
- [22] Sitraka Inc. JProbe.
<http://www.jprobe.com/> [November 12, 2001].
- [23] Sun Microsystems Inc. Java™ Heap Analysis Tool (HAT).
<http://java.sun.com/people/billf/heap/> [November 12, 2001].
- [24] Sun Microsystems Inc. The Java HotSpot™ Virtual Machine, 2001. Technical White Paper.
- [25] D. M. Ungar. Generation Scavenging: a Non-Disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.
- [26] D. M. Ungar and R. S. Smith. SELF: The Power Of Simplicity. In *Proceedings of OOPSLA'87*, pages 227–241, Orlando, FL, USA, October 1987.
- [27] VMGEAR. OptimizeIt.
<http://www.optimizeit.com> [November 12, 2001].
- [28] D. Wakeling. Compiling Lazy Functional Programs for the Java Virtual Machine. *Journal of Functional Programming*, 9(6):579–603, November 1999.
- [29] D. White and A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1999.
- [30] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In Y. Bekkers and J. Cohen, editors, *Proceedings of the First International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42, St Malo, France, September 1992. Springer-Verlag.
- [31] J. Zigman, S. M. Blackburn, and J. E. B. Moss. TMOS: a Transactional Garbage Collector. In A. Dearle, G. N. C. Kirby, and D. I. K. Søberg, editors, *Proceedings of The Ninth International Workshop on Persistent Object Systems (POS'9)*, pages 116–135, Lillehammer, Norway, September 2000.

C. FURTHER SCREENSHOTS

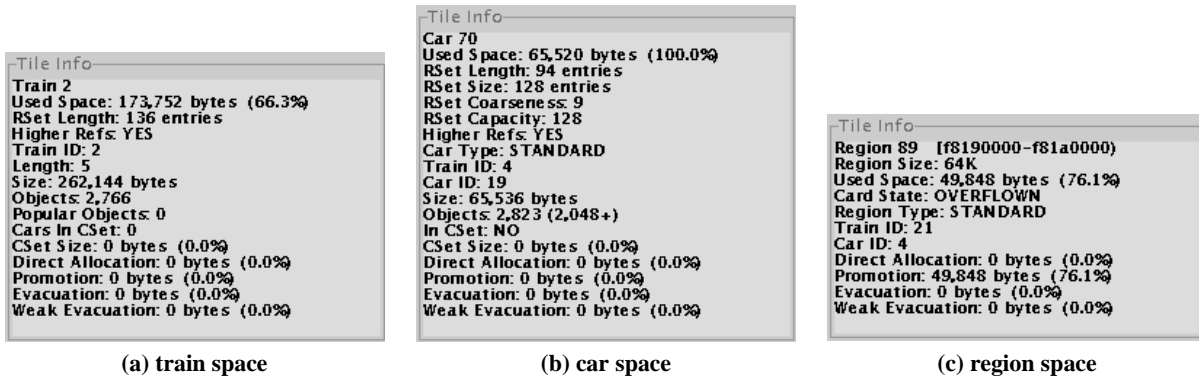


Figure 10: GCspy tile info text areas (they show information about a single tile in each space).

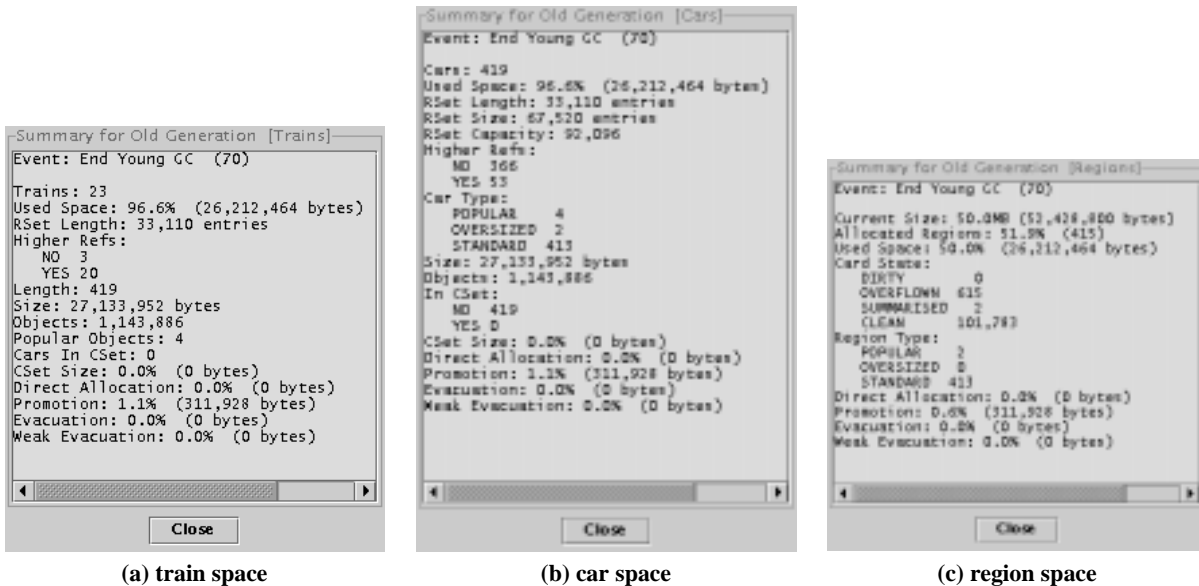


Figure 11: GCspy summary windows (they show aggregate information over all files in each space).