GCspy: An Adaptable Heap Visualisation Framework

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

Richard Jones
Computing Laboratory
University of Kent at Canterbury
Canterbury, CT2 7NF
U.K.
r.e.jones@ukc.ac.uk

ABSTRACT

GCspy is an architectural framework for the collection, transmission, storage and replay of memory management behaviour. It makes new contributions to the understanding of the dynamic memory behaviour of programming languages (and especially object-oriented languages that make heavy demands on the performance of memory managers). GCspy's architecture allows easy incorporation into *any* memory management system: it is not limited to garbage-collected languages. It requires only small changes to the system in which it is incorporated but provides a simple to use yet powerful data-gathering API. GCspy scales to allow very large heaps to be visualised effectively and efficiently. It allows already-running, local or remote systems to be visualised and those systems to run at full speed outside the points at which data is gathered. GCspy's visualisation tool presents this information in a number of novel ways.

Deep understanding of program behaviour is essential to the design of the next generation of garbage collectors and explicit allocators. Until now, no satisfactory tools have been available to assist the implementer in gaining an understanding of heap behaviour. GCspy has been demonstrated to be a practical solution to this dilemma. It has been used to analyse production Java virtual machines running applications of realistic sizes. Its use has revealed important insights into the interaction between application program and JVM and has led to the development of better garbage collectors.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*; C.4 [**Performance of Systems**]: Measurement techniques; D.2.3 [**Software Engineering**]: Testing and Debugging; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces.

General Terms

Algorithms, Measurement, Performance, Languages, Human Factors.

Keywords

Language implementation, Memory management, Garbage collection, Visualisation of objects, Java.

1. INTRODUCTION

Object-oriented programs consume prodigious volumes of memory. Dynamic memory management is thus a critical component of programming language implementations, whether the language is supported by garbage collection [20] or `new`/`delete`-style explicit deallocation [47]. Today's memory managers are sophisticated, often highly-configurable tools whose design has been guided by understanding of patterns of memory usage, object lifetime and reference in typical programs [41, 9, 14].

However, memory managers are often unstable in the sense that small changes in object lifetime, allocation pattern or heap size may cause large changes in performance; this is particularly true of garbage-collected programs. Although it is clearly essential for memory manager implementers to have a clear understanding of object demographics [42] and the effects of design decisions, such insight is difficult to achieve; up to now, neither tools nor general techniques have been available to assist in this task. Consequently, a typical development methodology has been to add code to the memory manager to take snapshots of the heap or to provide summative statistics of garbage collection runs. Detailed profiling incurs a substantial runtime penalty, so reporting is usually removed from production code although it is often only in deployment environments that memory managers are fully stressed; but it is precisely here that analysis is particularly needed. Worse, summative statistics may provide only data too generic to guide design or implementation, yet detailed snapshots generate intractable volumes of data.

1.1 Challenges for Visualisation

There are many challenges in providing visualisation of memory management systems. First, the impact on the host runtime system must be minimised. Clearly, heap measurements must not perturb the allocation behaviour of the memory manager. But the runtime cost of acquiring data should also be minimised and, in particular, the cost to the running program when data is *not* being gathered should be negligible. Second, large volumes of data must be captured and stored or transmitted, with regard for both space and bandwidth. Third, the visualisation system should be independent of the target runtime system or memory manager; solutions that require the user interface to have intimate knowledge of the system being examined are typically hard to extend or maintain. Fourth, portability is desirable; it should be possible not only to construct the visualisation user interface in any language of choice and on any platform (regardless of the platform and implementation language of the target), but also, and at any time, to connect to, and disconnect from, a target running on the local or a remote host. Finally, the user interface of the visualiser must be useful and flexible: it must provide the memory management developer with the information needed in a coherent, tractable and comprehensible form.

1.2 The GCspy Framework

GCspy is the first solution, to the best of our knowledge, that meets all of these challenges. First, it provides an *architectural framework* for the collection, transmission, storage and replay of memory management behaviour. Second, it provides a novel, informative and easy to use *visualisation tool*. The framework is easily incorporated into any existing memory manager. The visualiser can be attached to and detached from a running system at any time to provide on-line analysis. It can provide both a sequence of snapshots of the live heap and graphs that show how the heap evolves over time. Alternatively, GCspy can be used to store and replay traces, with facilities for fast-forwarding, single-stepping and so on.

Outside periods in which data is being gathered, GCspy imposes negligible runtime cost; the cost of data gathering depends on the implementation, but is of no worse magnitude than the cost of the sweep phase of a mark-sweep garbage collector [20]. Its design scales to very large heaps — GCspy is designed for, and has been used to visualise large-scale systems. In a 'stop and collect' world, the cost of gathering data depends in general on the size of the heap, although some useful information may be acquired in constant time from the memory manager (for example, heap occupancy in a compacting collector as, in this case, all used space is contiguous). The cost of storing, transmitting and visualising this data in the GCspy framework is independent of heap size; instead it depends on the visualisation granularity employed.

The framework is reusable and portable. It does not depend on intimate knowledge of the target system and the visualiser can be constructed in any language and run on almost any system; the only constraint is that both target and visualiser systems should support TCP/IP sockets. Although our visualiser implementation is written in Java, we have developed target-sides in both C and Java under several flavours of Unix. Finally, GCspy provides a flexible user interface. Currently, it offers two different ways of viewing information, but more are planned.

1.3 GCspy Users

GCspy was designed to assist memory management *implementers* to develop, debug and profile their system (whether garbage collected or explicitly managed). It has been used to analyse a variety of different memory managers, mostly for object-oriented languages, and has revealed unexpected insights into the interaction between application and JVM.

Memory managers are increasingly complex modules, offering large sets of command-line options (often mutually dependent), many of which have substantial effects on overall performance. Tuning such systems can be a black art. GCspy provides a valuable tool to *application developers* facing this task, as it lets the developer to see the *direct* consequence of option settings rather than merely indirect effects, such as changes in overall execution time.

We have also found GCspy useful as *educators*: its ability to connect to and disconnect from running systems, as well as its replay facilities, support student understanding of modern memory management systems.

1.4 GCspy in Practice

We have incorporated GCspy into three very different JVMs — Sun's RJVM[1] [46], Sun's HotSpot [40] and IBM's Jikes RVM (previously known as Jalapeño) [1, 2] — and into the Boehm-Demers-Weiser conservative garbage collector for C and C++ [5] in the context of the Dylan language [32]. Using GCspy, we have visualised nine different collectors, as well as other components of the systems (for example, the free-lists of in-place deallocating collectors). Further implementations are underway (or planned) for GHC Haskell [26], the Eclipse Constraint Logic Programming system [45] and the `dlmalloc` dynamic memory allocator [23, 24].

GCspy visualisation has provided a number of new insights into the interaction between JVM and application program. Analysis of the distribution of free space in an in-place deallocating collector allowed us to compare alternative allocation policies and choose those which caused the least fragmentation [19]. Visualisation of the Train collector [17] gave several new and important insights into its behaviour. It showed the presence in several applications of a few very long trains (previously, trains were always assumed to be short [31]), it facilitated the tuning of the remembered set implementation (again by evaluating alternative techniques) and it demonstrated the importance of handling highly-referenced objects specially. A more detailed description of the analysis of the Train collector using GCspy can be found elsewhere [28].

1.5 Paper Overview

In Section 2 we describe related work. Sections 3, 4 and 5 explain the model that makes the visualisation of large-scale systems tractable, including the abstractions that make GCspy easily portable, the architecture of our system and the implementation of GCspy servers respectively. Section 6 briefly illustrates the GCspy user interface, including the facilities that provide graphical representations of heap activity over time. Section 7 offers case studies of GCspy in practice and highlights the new insights that GCspy has revealed. We conclude in Section 8 and suggest directions for future work.

More information on GCspy, including colour versions of the screenshots that appear in this paper and various history graphs, which we recommend viewing, can be found at the following URL:

`http://www.dcs.gla.ac.uk/~tony/gcspy.www/`

2. RELATED WORK

Tools to support runtime object analysis can be divided into two categories according to whether they support application or system profiling and debugging. In addition, there is an abundance of static tools to support design and programming, such as modelling tools [15], class browsers [22], refactoring tools [12] and so on (we do not consider these further here). Of the runtime tools, by far the most common are those that focus on the needs of the application programmer [11, 18, 33, 37, 6]. The only tool of which we are aware that directly attends to the needs of the systems implementer is the Java HotSpot Serviceability Agent [30], a tool for examining and debugging the HotSpot JVM that is also capable of performing post-mortem analysis. Nevertheless, all tools that interact with the runtime system face similar issues: minimisation of disruption to the target system, capture, transmission and storage of information, and ease of maintenance.

Unlike GCspy, most other runtime analysis tools of which we are aware (such as the Modula-3 and SELF heap visualisers [7, 43]) are programming-language specific, hard-coded for the layout of that particular heap and offer only a fixed set of facilities. Halstead's

[1] Sun Microsystems Laboratories' Virtual Machine for Research, previously known as ExactVM, EVM or Java 2 SDK (1.2.1_05) Production Release for Solaris.

parallel program visualisation tool [13] is a notable exception. On the other hand, language dependence does allow the visualiser to make the connection between source code and runtime representation of objects. Indeed, this is precisely the motivation for the wide range of commercial tools currently offered [11, 18, 33, 37, 6]. These tools support source code debugging by allowing the programmer to discover the cause of object retention ('space leaks' [21]), either by drilling down to discover where an object was allocated or by navigating a graph to uncover references to incorrectly retained objects. Although some of these tools provide details on the number and volume of live and allocated objects, the generality of the statistics offered does little to assist the systems programmer to explore heap structure or object age, and the tools *cannot* report on memory manager-specific auxiliary data structures such as card tables, remembered sets and so on [16].

The GCspy visualiser is intended to be connected to a running system to provide a live view of its heap and auxiliary data structures. Alternatively, it can be used to capture and replay a trace. Other tools, such as Jinsight [18], JProbe [33], OptimizeIt [6] or HAT [37] provide only a snapshot of the object graph in the heap in some form or other and some means of browsing this snapshot. For example, HAT reads a JVMPI `hprof` dump [39] in order to provide a clickable, text-based representation of the object graph. The volume of data acquired by heap snapshots is huge: Java `hprof` snapshots may be three times as large as the system profiled [37]. TracingVM[2] [48, 49] generates traces typically reaching several gigabytes. In contrast, GCspy can visualise systems dynamically, providing its user with better responsiveness, and its traces are far more compact (at the cost of lacking the degree of detail provided by TracingVM).

To be effective, tools must offer the user some means to handle such complexity. For the most part, existing tool mechanisms are rudimentary: for example, HAT/`hprof` lists objects by class, e.g. "`407 instances of class [LJava.lang.String;`" [37, 39]. Notable exceptions are IBM's Jinsight [11] for Java and the Haskell heap visualisation tool [29]. Jinsight clusters object instances by 'reference patterns' in the object graph. Like GCspy, it provides filters to eliminate 'uninteresting' events [10]. The Haskell visualiser allows memory costs to be attributed to particular program points or 'cost centres' (these are expressions as Haskell is a pure functional language); it has been used to improve space behaviour (a notoriously difficult issue for lazy functional programs) substantially.

It is common for tools to use architectures similar to GCspy: a small in-process component communicating across a socket to an out-of-process user interface [10, 30, 38]. However, as far as we are aware, only GCspy, 'drive-by analysis' [10], the HotSpot Serviceability Agent [30] and `gdb` [34] allow attachment to, and detachment from, a running system. Like GCspy, the Java Platform Debbugger Architecture (JPDA) [38] uses a stateless, packet-based wire protocol, JDWP [36]. Chilimbi, Jones and Zorn discuss alternative approaches such as MetaTF, XML, or ASN.1 encodings [8]; Halstead's SDF [13] adopts a similar approach to that of MetaTF, whereas GCspy uses a custom binary communication protocol. In contrast, Ngo and Barton debug by 'remote reflection' [25]. The benefit of their approach is that no effort is required in the system being debugged — remote reflection relies on the host operating system for access to the JVM's address space — but reflective approaches rely on the debugger or profiler having an intimate knowledge of the target. The GCspy visualiser, on the other hand, requires a communication thread within the target runtime system.

Many profilers and debuggers impact the host system, either by adding substantial complexity to its implementation or by adding runtime overhead. For example, TracingVM [49] currently runs in interpreter mode only and its execution is typically one to two orders of magnitude slower than state-of-the-art JVMs. It is important to minimise perturbation of the target system and certainly to avoid distorting the data being measured; for example, the heap visualiser for the SELF interactive programming environment [43] imposes an impressively small penalty of only 1% on the system's performance [50]. For this reason, JVMPI implementations are required to be pure native. GCspy demands no such restriction, though some care must be taken in some environments to avoid memory allocation in the heap that is being visualised. There is no technical reason why GCspy could not use an enhanced version of the Java Platform Debugger Architecture's JVMPI and JDWP protocols. However, we chose not to do so because the bulk of the JVMPI/JDWP protocols are irrelevant to GCspy (and vice-versa) and because GCspy is intended to be platform independent: it is applicability is wider than Java VMs.

3. ABSTRACTIONS

A key goal of any visualisation system is to provide effective representations of very large volumes of information; to this, GCspy adds the goal that this should be done in a portable way and made target-independent. The key design issues are: tractable representation of the large volumes of information present in system snapshots; identification of suitable abstractions of both the components of the profiled systems and their attributes; and independence of the system to be visualised. In this section, we explain the abstractions used by the GCspy visualisation model and define our terminology.

3.1 Spaces

The GCspy framework operates over a set of abstractions to which a target system can be mapped. This framework has been demonstrated to be sufficiently general to provide visualisation of a wide variety of memory management systems.

> The **Heap** is the set of data to be visualised. It may comprise more than one component. No further constraints are placed on a heap. Typically, a visualiser will simultaneously present all components of the heap that are being visualised. Typically, a visualiser will display a representation of all components of the heap simultaneously.

The state of the heap changes as the program executes. Periodically, the GCspy framework captures the state of the heap and transmits it to the visualiser.

> **Events** are points in the program's execution at which the state of the target may be collected and sent to the visualiser.

Event examples might include the beginning and end of a garbage collection phase, heap expansion or even a regular 'clock' event. Identification of these events is the responsibility of the runtime system implementer. They should be sufficiently frequent to ensure a steady stream of transmissions to the visualiser, but not so frequent as to overwhelm the execution of the application.

[2] TracingVM is a modified JVM that stores to a file very detailed traces of application-level events, labelled by thread, such as allocations, object reads, object writes and so on. A simulator can reproduce the operation of the application by replaying such a trace. TracingVM treats objects as logical units and its traces are independent of the address each object had been given by the memory system. This allows the simulator to evaluate different collection strategies easily.

In order to attain independence of its target, the GCspy visualiser relies on bootstrap information from the visualised system that describes how the latter should be presented. The GCspy user interface uses this bootstrap data to configure itself before it starts receiving the visualisation data. Bootstrapping describes the target system in terms of *spaces* and *streams*.

> A **Space** is a named abstraction of a component of the system to be visualised. A space may represent a memory area, a free-list, a remembered-set and so on. A target system may comprise one or more spaces, according to its complexity. The framework only requires a partitioning of each space into a sequence of blocks (see below): no further restrictions are placed on the structure of the space.

3.2 Blocks

In order to visualise large volumes of information, the visualiser must determine its 'focal length': what visualisation granularity should be adopted. If the visualisation is done at the object-level, the resulting images will contain very detailed and accurate representations of the heap. In addition, this level of detail allows data to be processed to gather further useful statistics (object reachability, object size distribution, class popularity and so on). This is the approach that necessarily must be taken by profilers whose intent is to assist the debugging of application programs, for example to chase down the causes of memory leaks. However, this approach does not scale as heap sizes and object numbers increase to gigabytes and tens of millions respectively — the time to collect, send and then process the data becomes prohibitively high for a dynamic environment with on-line visualisation and the data volumes become inconveniently large[3] unless action is taken to filter out data [10]. Furthermore, the available screen size limits the amount of data that can be shown at any time. Zooming facilities can improve this, but might impose a further penalty on the visualiser operation.

Alternatively, the resolution of the visualiser can be reduced by subdividing the heap: this is the approach that GCspy takes.

> Each space is divided into a sequence of **Blocks** (for the purpose of visualisation only and not to be confused with memory manager notions): the set of not necessarily equally-sized partitions of the heap. For a heap region, the sequence of blocks is typically an address-ordered representation of the space, but this is not a requirement. The target system must send the sequence of blocks in the same order for each space and each event in the visualisation of a space.
>
> Blocks are a target-side notion; their visualiser-side equivalent is the **Tile**, a small area of screen estate (for example, a rectangle) that can be rendered according to one or more of the attributes of the block that the tile represents (see Figure 4, for example).

Thus, object-specific information is coalesced and it suffices to collect, send (and possibly store) only aggregate information about each block. Although this decreases the detail of the visualisation, the data volumes sent to and processed by the visualiser (and the user) are more manageable. This approach may still require iteration over each object in the heap in order to gather data, but the transmission and processing costs in the visualiser are bound by the number of blocks rather than the number of objects in the heap.

[3]Producing a high-detail memory operation trace from an undisclosed system has been recently reported to have generated a 150GB trace file.

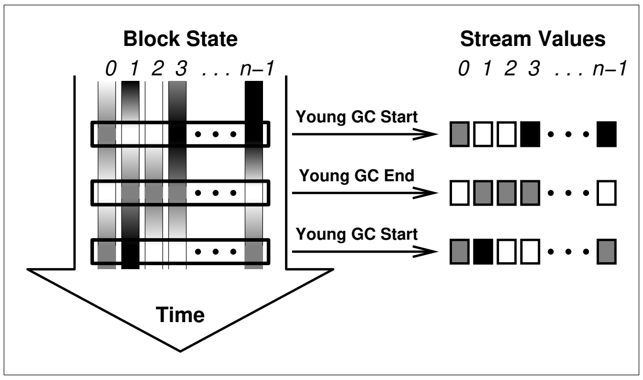


Figure 1: A stream for a single attribute of a single space, highlighting the state of the blocks of this space at a particular event.

Disturbances caused by gathering data can also be bound by collecting it concurrently (see Section 5.4). Furthermore, visualisation granularity (i.e. block size) can be adjusted to obtain the best trade-off between visualisation detail and limitations of screen size. We demonstrate in Section 7 that this approach scales to very large heaps.

3.3 Streams

Each block of a space will typically have several attributes, such as the amount of used space (however the user wishes to define this term), the number of objects it contains, the length of a free-list, or whatever. The framework models the attributes of a space by GCspy *streams*. At each event in a program's execution, the target system transmits the state of each stream of each space to the visualiser. More precisely,

> Each **Stream** of a space has a name, a type, a range of permissible values and certain descriptive textual information; its current state is described by a sequence of integer values, one for each block of the space. GCspy requires that each block of a space has the same set of attributes, although different spaces may have different ones.

Figure 1 represents the gathering of data over time from a single stream. The stream represents the values of a single attribute of a single space of n blocks. As time elapses, the values of the blocks' attributes will change, possibly continuously. However, the target system will typically gather data only at discrete event points in its execution: three of these points are highlighted in the figure. At these events, a sequence of values representing the current state of this attribute for each block of the space is transmitted to the visualiser.

3.4 Summary Information

Ideally, the visualiser should decide upon its representation of data solely from the data streams it receives. However, this information is not always sufficient. For example, it is often useful to provide *summary information*, i.e. a value that summarises an entire stream at a particular event, such as the total number of objects in the space or the total length of all free-lists. These summaries represent a numerical overview of the current state of each space. A subtle point is that the summary information may not always be accurately derivable from the stream data supplied to the visualiser. Consider a stream that represents the number of objects per block. If an object spans two blocks, it may be counted in both of them. Any attempt within the visualiser to calculate the total number of objects risks double-counting. Because of this, the GCspy framework assumes that summary information is gathered by the server and transmitted separately.

3.5 Control Information

It is often desirable to describe the structure of a space further. This may be for presentational or for efficiency reasons. For example, a space may contain a number of application-specific, dynamically moving, internal boundaries (such as boundaries between generations in certain styles of generational collector [3]). Additionally, it may be desirable to distinguish blocks that are unused. For these reasons, a generic and extensible mechanism was introduced to provide such control information to the visualiser. The visualiser will typically use such information to add visual enhancements to the rendered space images in order to improve the appearance (and hence effectiveness) of the visualisations. Such enhancements include marking tiles as unused (to show heap areas that contain reserved but uncommitted address space, or the unused semi-space of a two-space collector), drawing separators between tiles (to specify dynamically-moving boundaries in the space), and so on. Each space provides this information through an additional *control stream*.

3.6 Bootstrapping

Visualisation of any system will be in terms of the abstractions described so far. The system's implementer must decide how many spaces are required and how many streams each space will have. This information forms part of the bootstrap information sent to the visualiser. This data names the target, describes each space and identifies the set of possible events. Each space is specified in terms of its name, size (in number of blocks — this can be modified dynamically) and its streams. Each stream is specified by its name, the type and range of its data values and certain other information relating to its presentation, particularly to how its textual form should be shown (for example, textual information such as units of measurement to improve readability). After the visualiser has been configured, it suffices for the memory manager to send sequences of values, specifying only the space and stream to which they correspond.

3.7 Summary of the Model

To summarise, the requirements of a system for it to be suitable for visualisation by GCspy are that:

(a) the system comprises one or more components;
(b) each component is represented by one or more spaces;
(c) there is a finite partition of each space into blocks;
(d) all blocks of the same space have the same set of attributes; and
(e) each attribute of each block can be represented by an integer within a fixed range.

Table 1 shows simplified examples of how different garbage collectors may be mapped to the GCspy abstractions.

Collector	Spaces	Streams
Simple semi-space collector	The whole heap	Used space Objects ...
Simple mark-sweep collector	The whole heap	Used space Objects Objects marked ...
	Free lists	Length Size ...
Generational collector	Young generation	Used space Objects ...
	Old generation	Used space Objects Objects marked Promoted objects Card table state ...
Train collector (mature object space)	Young generation	Used space Objects ...
	Trains	Length Remset info ...
	Cars	Used space Objects Type Train ID ...
	Physical regions	Used space Type Train ID ...

Table 1: Examples of how different garbage collectors might be mapped to the GCspy abstractions.

4. GCspy ARCHITECTURE

It is essential that the coupling between the GCspy framework and the target system be minimal. In particular, GCspy can make no assumption about the implementation language of the target system. For this reason, solutions based on reflection [25] are ruled out, despite their desirable properties of minimal interference with the running system. Likewise, the target system side of GCspy should have no knowledge of the use to which attribute data is to be put: for example, it should be immaterial whether the output is to be consumed by a visualiser (or other process) or be saved to disk, on the same host or a remote one.

4.1 Client-Server Model

For this reason, a client-server model was adopted for GCspy. The target system that is being visualised acts as the *server* (i.e. a GCspy server needs to be incorporated into it) to which the *client* (the visualiser or the trace storing tool) connects. The communication between the client and the server is performed over standard TCP/IP sockets. Sockets were chosen over extending existing mechanisms (such as JDWP [36]) because they are reasonably generic, portable and not tied to a particular language, runtime system, operating system or machine architecture. Even though the client-server approach considerably complicated the design and implementation of GCspy, it has several important advantages.

First, the visualiser and the target system are launched as two separate processes. This minimises the additional code to be incorporated into the server process and the runtime interference by the visualiser into the server. Although the GCspy server includes a thread to handle communication between the target system and

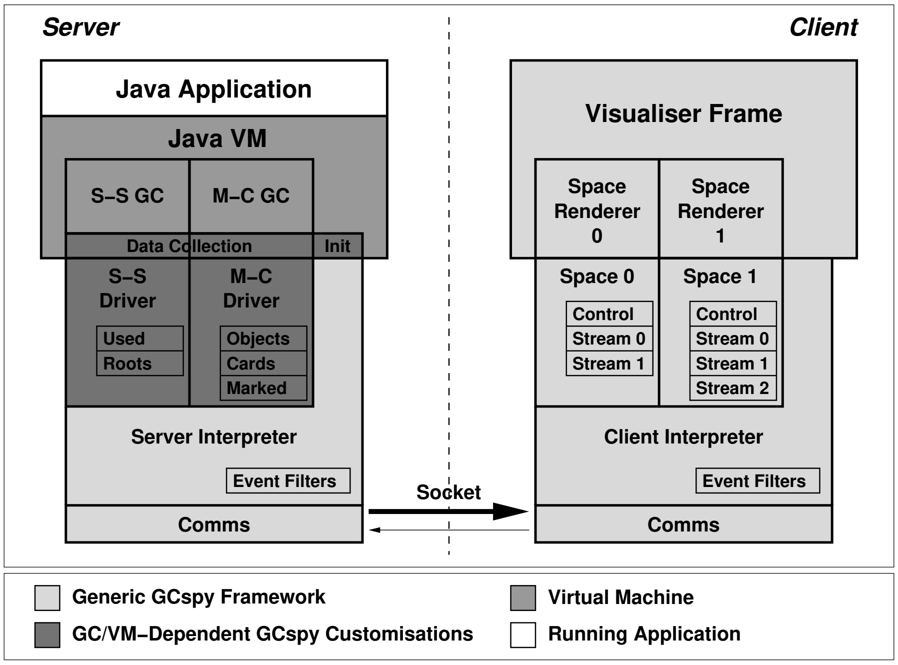


Figure 2: The GCspy client-server architecture showing a Java VM managed by a generational collector. The younger generation uses a semi-space copying collector (S-S GC) and the older a mark-compact collector (M-C GC).

the visualiser, its impact in terms of thread scheduling is small (especially if a second CPU is available for this purpose). For the most part, transmissions to the client only take place at 'safe points' (for example, at the start or at the end of a garbage collection run) to send visualisation data gathered by the memory manager. At these points, threads other than memory manager threads are typically stopped. For example, transmission points in RJVM (see Section 5.6) correspond to the stop-the-world phases of one of its collectors: all data collection and transmission takes place while all mutator threads have been stopped. Figure 1 showed the operation of such a mechanism. Apart from increasing the stop-the-world pause time, a transmission does not affect any running application threads (scheduling, priorities and so forth). Second, by making the visualiser a separate process, it can be launched on a different machine to further reduce the impact on the application. However, the GCspy framework does not restrict transmission to safe points. We describe in Section 5.4 how a concurrent collector might be visualised with only weak synchronisation with the JVM. As a result, the visualisation framework is as non-intrusive as possible and has the potential to produce results that are not skewed by the presence of the visualiser.

A client-server architecture also allows the visualiser to be connected to the target system only when it is necessary, leaving it undisturbed otherwise. This has important benefits. The user can launch their application as usual and only connect the visualiser to it when a problem occurs in order to attempt to discover its cause, or connect to it intermittently to monitor its behaviour over time. Thus, GCspy can be deployed outside the laboratory to diagnose those performance problems that commonly manifest themselves only under real loads. For example, a JVM vendor providing support to a highly-valued client may visualise an application running remotely on the client's machine.

Finally, a further advantage of this approach is that the visualiser and the target system do not have to be written in the same language. This improves the portability of the framework and allows it to be easily incorporated into a variety of systems. Only the server-side code needs to be ported to a particular system, whereas the visualising client is generic and can be used unchanged in any situation. On the server side, the harder undertaking of writing the server code itself needs to be done only once for a particular JVM or other target system. Writing data gathering code for each garbage collection algorithm within a memory manager, for example, is almost trivial (see Appendix C.2).

4.2 Storing / Replaying Traces

It is desirable to be able to store GCspy transmissions in order to replay them and analyse them at a later time. This is useful for demonstrations, for sharing of information with remotely-located colleagues and where it is undesirable to allow a remote system to connect to the server. An example of the last scenario is that companies are not always willing to release their software to researchers on confidentiality grounds. Our hope is that they will be able to release the trace files instead to improve the community's understanding of how realistic applications behave.

GCspy's client-server architecture facilitates storage and replay. A special-purpose client connects to a server and simply stores in a trace file the attribute data the server sends. Similarly, a special-purpose server reads this trace file and transmits the stored data to a connected visualiser. Both these utilities are written in Java, are general-purpose and use Java's gzip API to compress traces. The trace-storing client configures itself according to the bootstrap information sent from the server and writes this information to the trace file, so that the trace-replaying server can adapt itself for each such trace. The visualiser cannot distinguish whether it is connected to a GCspy-enhanced target system or the trace-replaying server. Because of its coarse-grain visualisation and because it needs to store only data streams and summaries, rather than the events of the collection itself, the trace files collected by GCspy are compact and compress well (see Table 3).

4.3 Wire Protocol

Communication between client and server is through a custom binary protocol. GCspy's target- and visualiser-independence requirements preclude use of language-specific protocols such as JDWP [36]. For performance reasons, we chose not to use a text-based protocol (such as XML or MetaTF [8]). The GCspy protocol does maintain some state between the client and the server (the state of the event filters, the configuration of each space and so on).

5. SERVER IMPLEMENTATION

It is vital that GCspy intrusion into the target system being visualised be kept to a minimum, both in terms of any code that must be added and in terms of runtime costs. GCspy code added to the system should be small. Systems that embed the visualiser in the target system would be hard to maintain and might intrude substantially on the performance of the visualised system as the user explored the data in the user interface. As far as possible, our goal is to allow the target to run at full speed without perturbation, except when it is required to gather data, for example immediately before or after a garbage collection. In this section, we explain the server-side model and the work required of the memory manager implementer in order to incorporate GCspy into their system.

Each component of the target system to be visualised is modelled by the GCspy framework as a space (see Section 3). Figure 2 shows the GCspy framework incorporated into a JVM that has a generational memory system with its young generation managed by a semi-space collector and its old generation managed by a mark-compact collector [20]. In the figure, two spaces have been created, one for the semi-space collector, the other for the mark-compact collector. The semi-space collector has two streams (*used* space per block and number of references from *roots* to each block) and the mark-compact collector three (number of *objects*, *card* table state and *marked* objects per block).

5.1 Drivers and Interpreters

Communication between the collector and the GCspy infrastructure is performed through a *driver*. The role of the driver is to map information collected by the memory manager to the streams supported by the driver's space. The driver is also responsible for collecting summary and control stream information (see Sections 3.4 and 3.5). As each driver receives sufficient information to generate stream data, it is easily extended to generate the summaries as well. As it understands the structure of its space, it can also provide the necessary control information. Clearly, drivers are collector- and server-specific: whenever the implementer wishes to visualise a new component of the memory manager (such as a new implementation of the remembered sets of a generational collector), a new driver must be written (see Section 5.5). However, the client-side counterparts — the space and space-renderer modules — are generic and not tied to any specific system or driver, as they operate over the stream abstractions.

The server and client *interpreters* are generic modules, responsible for serialising and transmitting data over the socket and for receiving, deserialising and interpreting (hence their name) incoming transmissions. The server-side interpreter is a library that exposes an API to the driver programmer. For the most part, the GCspy interpreter is portable between servers (an exception is the interpreter for Jikes RVM which is written in Java and so must take care to avoid allocation in the Java heap). Communication between the client and the server is asymmetric. The server mostly 'pushes' data to the client when appropriate: initially, the bootstrap configuration information and, subsequently, stream contents. Communications from the client back to the server are mainly user-initiated and infrequent, such as disconnect, control flow commands (play, move forward to the next event and pause) and event filter modification (see Section 5.3).

To transmit the state of the heap, the *server interpreter* serialises each data and control stream to GCspy's custom binary format and sends it to the client. There, the *client interpreter* is responsible for extracting the values and installing them in the appropriate stream of the appropriate space (essentially acting as a multiplexer). Finally, once the data of all available streams have been transmitted, the server sends a final message to notify the client that it can redraw the on-screen representation of the space.

The benefit of this architecture is that it allows the GCspy framework to be ported easily to any system. Such a port involves writing appropriate (collector-specific) drivers, usually deduced easily from existing ones, and adding a small amount of platform-specific code to the server to communicate with the drivers and to initialise the server. Only the server needs to be adapted to a particular system — the client can be used unchanged. The code required is small: Section 5.5 describes an example implementation.

Currently, two implementations of the GCspy server code exist. One is written in C, because a large number of runtime systems are implemented in or can interface with C. The other is written in Java. The latter acts as the reference implementation, is used for the trace-replaying tool and can also be incorporated in JVMs that are written in Java (for example, Jikes RVM [2]). The visualiser is entirely written in Java using Swing; the Java client infrastructure is also used for the trace-storing tool.

5.2 Control Flow

Transmissions are sent from the server to the client when a specific event is reached. To prevent overwhelming the visualiser (and hence the user), the GCspy framework provides facilities to pause and resume the execution of the application. Whenever the user presses the pause button (see Figures 4 and 6), the client sends a pause request to the GCspy server to cause it to pause itself after the next transmission and to stay paused until it has received a resume request. This ensures that, when the system has been paused, all transmissions have completed and the data that has reached the visualiser is consistent. Disconnecting happens in a similar fashion. Again, it is the responsibility of the implementer of the target system, if they want to take advantage of the pause facility, to ensure that no other threads are operating when the GCspy server pauses. In some systems this is easy to achieve; for example, stop-the-world collectors typically halt all mutator threads so GCspy operations can be piggy-backed on that.

5.3 Event Filters

Events deemed uninteresting by the user can be *filtered*. Filtering is implemented by the server (i.e. the server does not transmit these events, as opposed to the client not accepting them). Data gathering for skipped events should be omitted in order to avoid affecting the performance of the running system at uninteresting event points. Four event filters are associated with each event.

- **Enable / Disable** filters allow the user to enable or disable transmissions at specific events. For example, a user might only be interested in old generation collection events and not young generation ones. Disabled events do not cause transmissions to the visualiser and memory managers should not even collect data for them.

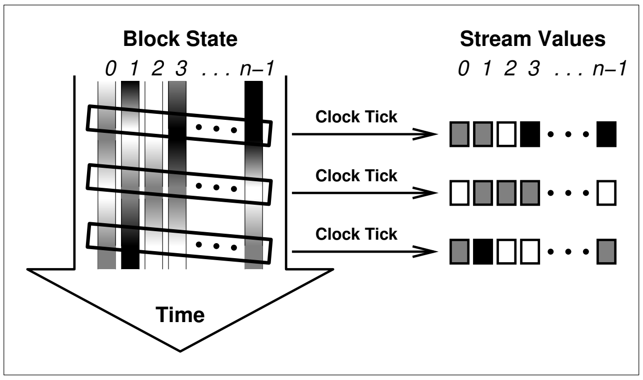


Figure 3: Capturing data from the stream for a single attribute of a single space asynchronously.

- **Delay** forces the server to delay for a given period after transmissions of specific events, allowing the user to 'slow down the film'. We choose to delay at the server side to ensure that client and server are always synchronised.
- **Pause** forces the server to pause after transmission of specific events, saving the user from having to pause the application manually every time such an event is reached (*cf.* debugger break points).
- **Period** allows a regular sampling of events; it causes transmissions of specific events to happen once every *n* events. This is useful if certain events are too frequent and allows the user to reduce their transmission rate.

5.4 Data Collection

The GCspy framework does not define a data collection method. This is the responsibility of the target system implementer. In practice, there are several ways to implement data collection within a garbage collector. Three techniques are enumerated below.

The first way is to piggy-back data collection on garbage collector operations. This ensures that the driver's stream data always contains an up-to-date snapshot of the state of the corresponding space. Here, the collector is extended with code that updates stream data as it operates. This incremental way of collecting data is possibly the fastest. However, it has two disadvantages. First, data-collecting code must be planted throughout the collector, making the maintenance of both more difficult. Second, and more importantly, data collection cannot be turned off when it is not required (for example, when the visualiser is not connected or an event is disabled or skipped) without imposing a performance penalty, as the test whether to gather data or not must be repeated at each data acquisition point (for example, whenever a mark-sweep collector marks an object).

Alternatively, data may be gathered by performing separate and complete sweeps over each component. When an event is reached, stream data is typically recreated from scratch before being transmitted. Although the performance penalty of such sweeps is high, checking whether data should be gathered or not involves just a single test per event. The cost of this test is negligible, allowing the runtime system to execute at practically full speed when the visualiser is not connected or when no events are enabled. Additionally, all the extra code in the collector is concentrated in just one place and does not interfere with the bulk of the collector code. Ports of GCspy to RJVM, HotSpot and Jikes RVM adopted this approach.

The third technique is to collect stream data and transmit it to the visualiser concurrently, without requiring the mutator threads to be halted at all. This will have very little impact on the performance of the target system, especially on multi-CPU machines. Concurrent collectors also benefit as it will not affect their maximum pause time, even when the visualiser is connected. Again, the extra code in the collector is also localised, typically in a separate thread, with a regular 'clock' event providing the points of transmission (see Figure 3). Unfortunately, this approach cannot guarantee that the data sent to the visualiser will be consistent, as it is gathered concurrently with the mutator's operation (compare Figures 1 and 3). However, any single discrepancy caused by this (such as a large free chunk considered to be allocated because it has been temporarily removed from the free-lists to satisfy a small allocation request) will most likely affect a single transmission and not subsequent ones. Users can still obtain useful visualisations of a system, but they must be aware that inconsistencies might be apparent.

5.5 Building and Incorporating Drivers

One of the benefits of the GCspy framework is the ease with which it can be incorporated into an existing memory manager. In this section, we outline how a simple driver can be added to a semi-space collector. Pseudo-code for this example is given in Appendix C. First, at boot time, the JVM must initialise the server interpreter (see Section 5.1) and register the required set of events with it. This infrastructure will be shared by all drivers. As each specific collector is initialised, it must create and initialise its own driver.

The driver code has the following responsibilities. On creation, it creates and initialises its space (or spaces, in the case of a complex component), registers that space with the server interpreter, creates and initialises the space's streams and registers them with the space. To support data gathering, the driver typically provides a method that initialises the data of all the streams to default values[4] and methods to update the stream data according to the state of the collector (a method that specifies the location of an object, a method that specifies the state of a card, and so on).

Suppose that the collector is required to gather data and transmit it before and after each collection. First, it must check whether the server should transmit (i.e. a client is connected) and whether the event is active (i.e. it has not been disabled and it is not being skipped). If this is the case, the collector calls on the driver to initialise its streams (as the stream data is typically recreated for every transmission) and then iterates over the heap, communicating to the driver the location of objects, state of the cards and so forth. Finally, the collector notifies the driver that data gathering has been completed so that the driver can transmit the streams to the client.

In the next two sections, we describe the incorporation of GCspy into two different JVMs and attempt to estimate the effort required to add new streams to existing drivers, to write new drivers or to port GCspy to a new system. We provide GCspy performance results for one of these JVMs in a third section.

5.6 Example Implementation I: RJVM

RJVM was the first system in which GCspy was incorporated (and acted as the main test case during GCspy development). It has a two-generation memory system in which each generation can be managed by one of several garbage collectors. This is facilitated by RJVM's *GC Interface* [46], which abstracts the memory management system from the rest of the JVM. In order to be able to visualise different configurations of the memory system with GCspy, six different drivers were developed.

[4] Some drivers may not need to re-initialise streams, for example if they are accumulating values over the course of a program's execution.

	Configuration	_213_javac		reptile	
		sec.	rel.	sec.	rel.
i	No GCspy	29.07	1.00	26.90	1.00
ii	GCspy, uninitialised	29.15	1.00	27.00	1.00
iii	no connection	29.12	1.00	27.13	1.01
iv	connection, no events	29.15	1.00	27.13	1.01
v	connection, level 1	53.22	1.83	75.07	2.79
vi	connection, level 2	55.61	1.91	76.27	2.84
vii	connection, level 3	95.60	3.29	179.62	6.68
viii	connection, level 4	283.48	9.75	554.45	20.61

Level	Description
1	Collect only data that can be acquired in constant time (such as the space used by a compacting collector).
2	As level 1, but also iterate over small data structures (such as the card table).
3	As level 2, but also iterate over objects in the heap (for instance, to count the objects).
4	As level 3, but also iterate over reference fields in objects (for instance, to gather statistics on direction of pointers).

Table 2: Impact of GCspy on total execution time for `_213_javac` and `reptile` benchmarks. Both absolute time and time relative to that without GCspy are shown. Fixed heap sizes of 18Mb for `_213_javac` and 55Mb for `reptile` were used.

(1) A semi-space driver for the two-space copying collector that is the default configuration for the young generation (see Figures 4 and 6). It manages one space with two streams.
(2) A mark-compact driver for the sliding-compacting collector (the default configuration for the old generation), which manages one space with five streams.
(3) A driver for a simple, non-moving, mark-sweep collector, offered as an alternative for RJVM's old generation (see Figures 4 and 6). It manages one space with seven streams.
(4) A driver for the generational, mostly-concurrent, incremental collector [27], offered as an alternative for RJVM's old generation. It manages one space with eight streams.
(5) A driver for the Train collector [17], offered as an alternative for RJVM's old generation, which manages three spaces with a total of forty streams. Its operation is described in more detail elsewhere [28].
(6) A driver for visualising the operation of non-moving collectors' free-lists, used with drivers 3 and 4 (see Figures 4 and 6). It manages one space with two streams.

The events that are defined depend on the collector configuration used. For the two stop-the-world configurations (for instance, driver 1 for the young generation and drivers 2 or 3 for the old generation), there are five events: Young generation collection start, Young generation collection end, Old generation collection start, Marking phase end and Sweeping (or compacting) phase end. The two young generation collection events and four further events are used by driver 4 (for the mostly-concurrent collector) to match the phases of that particular collector: Root checkpoint start, Root checkpoint end, Remark phase start and Remark phase end [27]. Similarly, in the case of driver 5 (for the Train collector), in addition to the two young generation collection events, there are two more: Train collection start and Train collection end.

The architecture of GCspy made the introduction of different events for different collectors straightforward. In our experience and assuming that the programmer is very familiar with the driver and the framework, it takes about thirty minutes to introduce a single new stream to an existing driver, including incorporating the data-gathering code. Deriving the first version of driver 4 from driver 3, and writing driver 6 from scratch, took just under two hours each (including incorporation into RJVM).

5.7 Example Implementation II: HotSpot

GCspy has also been ported to Sun Microsystems' HotSpot JVM (the version used is close to the 1.4 public release). HotSpot also uses a generational framework and the port involved three drivers: one for the young generation, managed by a semi-space collector, one for the old generation, managed by a mark-compact collector, and one for the permanent space (this is used for classes, methods, etc.), also managed by a mark-compact collector. The drivers have six, thirteen and ten streams respectively and were adopted from drivers 1 and 2 of RJVM. A young generation collection causes two events and an old generation collection three (as for the RJVM implementation). The port took about a week, but most of this time was spent on understanding HotSpot's generational framework, rather than incorporating GCspy.

5.8 Performance

We now provide results for GCspy performance. In all cases, the figures are for the version of Sun's HotSpot JVM described above. All benchmarks were run under Solaris 7 on a Sun Ultra80 workstation with 4×450Mhz UltraSparc II CPUs and 2GB of memory.

As it is important that the server experiences no GCspy-imposed performance penalty when the visualiser is not connected, data collection in the HotSpot JVM was done by complete heap sweeps (as described in Section 5.4). The benchmarks used to measure performance were `_213_javac` from the widely used SPECjvm98 suite [35] and `reptile`, the kernel of an Escher drawing package translated from Haskell into Java bytecodes [44]. Table 2 gives performance results for several configurations. In the table, row (i) shows results for the control set, that is, the HotSpot JVM without GCspy compiled in; row (ii) had GCspy compiled into the JVM but it was not initialised, its communication thread was not launched and no data collection was performed. In the remaining rows, GCspy was compiled in and booted. Clients connected to the server in rows (iv)–(viii) but not in row (iii). In row (iv), all events were disabled so only bootstrap data was transmitted. The last four rows show the effects of increasingly expensive data collection levels (these are described in Table 2).

These timing results confirm our claims that the incorporation of GCspy does not impose a noticeable performance penalty unless GCspy is initialised and gathering data. If the thread is started but no transmission occurs, the overhead is minimal or non-existent (we do not have a satisfactory explanation for the difference between `_213_javac` and `reptile` in this case). When the visualiser is connected, for light data gathering the JVM slows down by a factor of two to three, and for heavy data gathering by over an order of magnitude, which was expected. The GCspy overhead is greater for `reptile` than for `_213_javac` because the former allocates mainly very small objects; hence iterating over each megabyte of heap for `reptile` is more costly than for to `_213_javac`, which has fewer and larger objects. The user can improve performance, if

Traces	Events	Collections		Heap Sizes (Mb)			Tiles	Trace Size
		young	old	young	old	perm	(8Kb blocks)	(Kb)
_202_jess	304	146	4	2.1	2.2	2.0	805	189.23
_205_raytrace	161	76	3	2.1	4.7	1.5	1068	333.17
_209_db	89	37	5	2.1	13.2	1.5	2148	198.74
_213_javac	241	101	13	2.1	13.3	2.0	2230	973.65
_228_jack	204	87	10	2.1	1.4	1.8	672	143.00

Table 3: GCspy trace file sizes for SPECjvm98 benchmarks using level 3 (see Table 2) data gathering. Heap sizes refer to the young generation, old generation and permanent space; tile numbers and trace sizes include all three spaces.

they wish, by using event filtering (see Section 5.3). Running the visualiser locally or remotely does not seem to have a large impact on performance.

If desired, GCspy can store rather than display event transmissions. Table 3 shows the size of the trace files obtained from the SPECjvm98 test suite. The heap sizes reported are the maximum size each space of the heap reached while running each benchmark. The visualisation granularity was set to 8Kb (i.e. each tile represented an 8Kb memory block).

6. THE USER INTERFACE

The GCspy visualiser runs in a separate process from the target being observed, communicating via a standard TCP/IP socket. The visualiser is entirely generic and relies on bootstrap information from the server for initialisation. This bootstrap information provides the name of the server to which the visualiser is connected (Figure 4, area ❽), the names and sizes of each space (Figure 4, areas ❶), the names of the events defined on JVM (Figure 4, area ❸) and so on. The state of the server can either be displayed one event at a time (Figures 4 and 6) or as a *history* — an evolution of the state over time (Figure 7). We discuss each view below.

6.1 The Main Window

Figures 4 and 6 illustrate the main window of the GCspy user interface when connected to the JVM described in Section 5.6. The window is split into several areas, each of which is numbered in Figure 4. These are described below.

- ❶ **Spaces.** Most of the GCspy main window is taken up by the areas where spaces are rendered. There is one such area per space. Even though all spaces are visualised at all times, the one selected by the user to be *active* is denoted by a dark frame around it. The active space controls areas ❹, ❺ and ❻. In Figures 4 and 6, each tile in each space corresponds to a 128Kb block in the heap. Notice the effects of the control stream in the Young Generation space: unused tiles (corresponding to the unused semi-space) appear pale grey and a separator identifies the boundary between the two semi-spaces.
- ❷ **Space Tool Bars.** Each space has a separate tool bar from where windows associated with it can be launched. One such window is the legend window (in Figures 4 and 6) which shows the representation of a low, a mid-way and a high value. Notice that care is taken to represent zero values differently (a light frame rather than a solid tile) from very small values — it is useful to distinguish these. Unused tiles are also explicitly identified. The summary window (Figures 5(a) and (b)) contains the summary information for the entire space.
- ❸ **Current Event.** This displays the name of the event that caused the transmission currently being visualised.
- ❹ **Tile Information.** When a tile of the active space is selected by the user (denoted by a white frame around it), this area displays the values for that tile in all available streams (essentially the attributes of the corresponding block). The presentation information associated with each stream, described in Section 3.1, allows the visualiser to provide a more appropriate representation of its values, for example, as a percentage for the used space attribute, as an enumeration for the card state attribute and so forth. Note that permanent information that does not change from transmission to transmission (namely the address range of the block represented by the tile and its size) is transmitted to the visualiser just once, at connection-time.
- ❺ **View Chooser.** This menu allows the user to choose which stream will be used to visualise the active space. In Figure 4 for example, the used space attribute has been selected. Currently, there is a one-to-one correspondence between streams and views.
- ❻ **Magnification.** This shows a subsequence of the tiles of the active space, centered on the active tile and rendered with smaller tiles, and a further subsequence, again centered on the active tile and rendered with larger tiles. The intention is to reveal artefacts over a series of tiles that might not be apparent in the main space area due to line breaks. In future work, we plan to experiment with combining attributes in a single tile and suggest that the larger tile size may add clarity.
- ❼ **Menu Bar.** The menu bar allows windows relevant to all spaces to be launched, such as the event counter window, listing the number of occurrences of each event defined by the server (see Figure 5(d)), and the event filters window, controlling event filtering (see Figure 5(c)).
- ❽ **Main Footer.** This is split into three areas. The left area includes the control flow buttons: (from left to right) pause, resume and step one (the latter is only activated when the visualiser is paused; it moves the execution to, and pauses it at, the next event). The middle area indicates the server to which the visualiser is connected and its state (i.e. paused or about to pause). Finally, the right-hand area allows disconnection from the server and reconnection to a new one.

6.2 History Graphs

The GCspy visualisations described so far, and illustrated in Figures 4 and 6, reveal the state of a system at a single point in time. The evolution of a system over time is also of great interest, but this is hard to observe from a succession of snapshots. To provide a concise view of the history of an attribute of a single space, GCspy can present its evolution as a graph (see Figure 7). Such *history graphs* are two-dimensional grids of very small tiles (large tiles would generate prohibitively large images) in which each row corresponds to a single GCspy event transmitted. Whenever a new event transmission reaches the visualiser, a new row is added to the

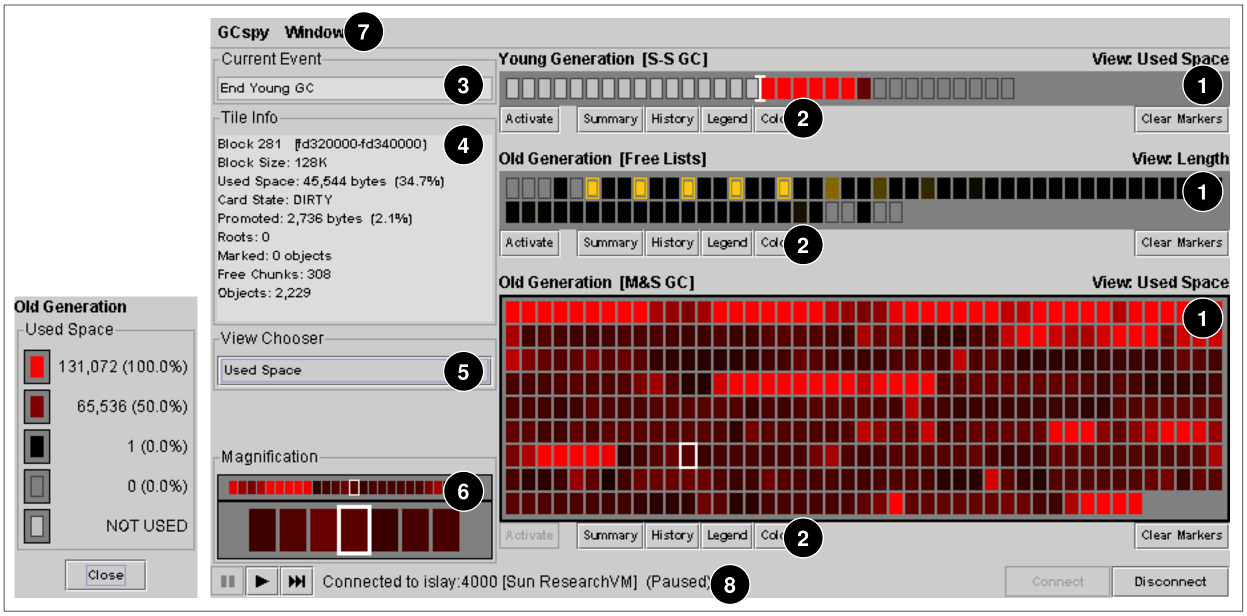
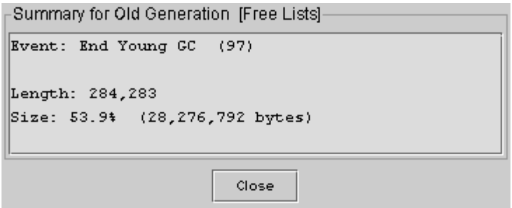
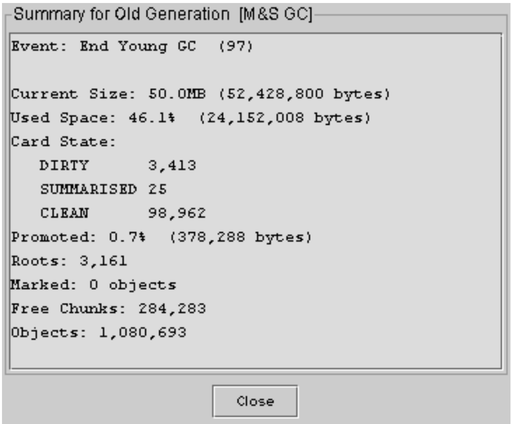


Figure 4: The GCspy user interface main window showing the *Used Space* view of the Old Generation space of RJVM for the `reptile` benchmark. The Mark Sweep space is active, a tile on it is selected, and its legend window is also included on the left.



(a) Summary window for Free List space.



(b) Summary window for Mark Sweep space.

(c) Event filters window.

(d) Event counters window.

Figure 5: GCspy auxiliary windows for RJVM.

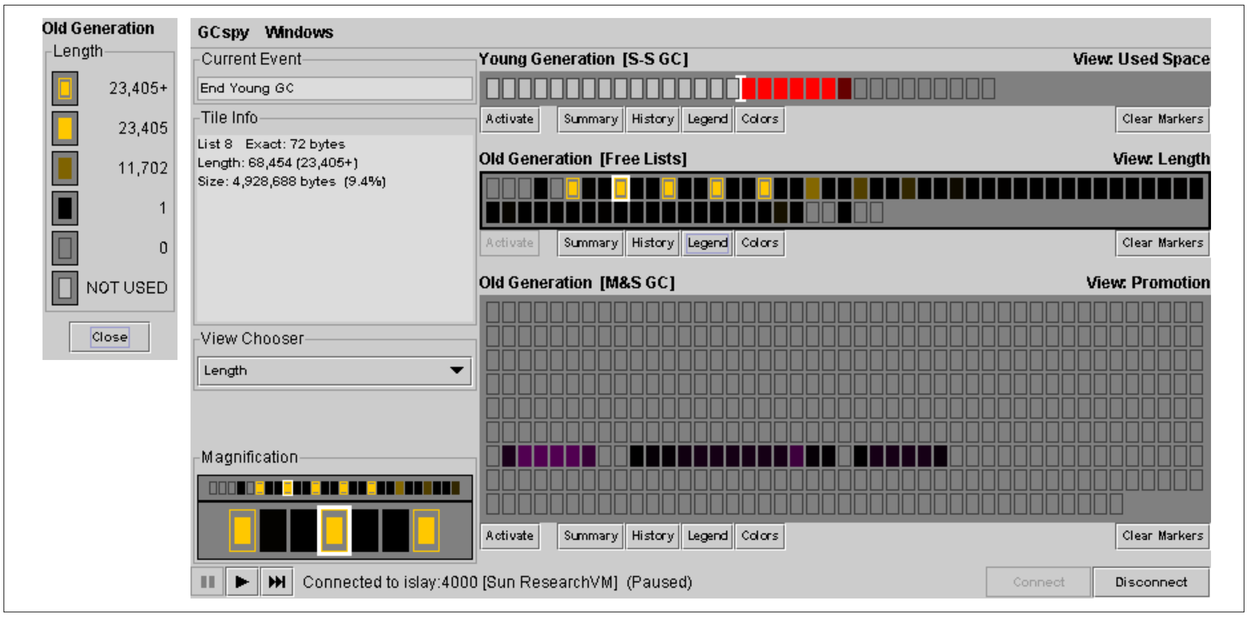
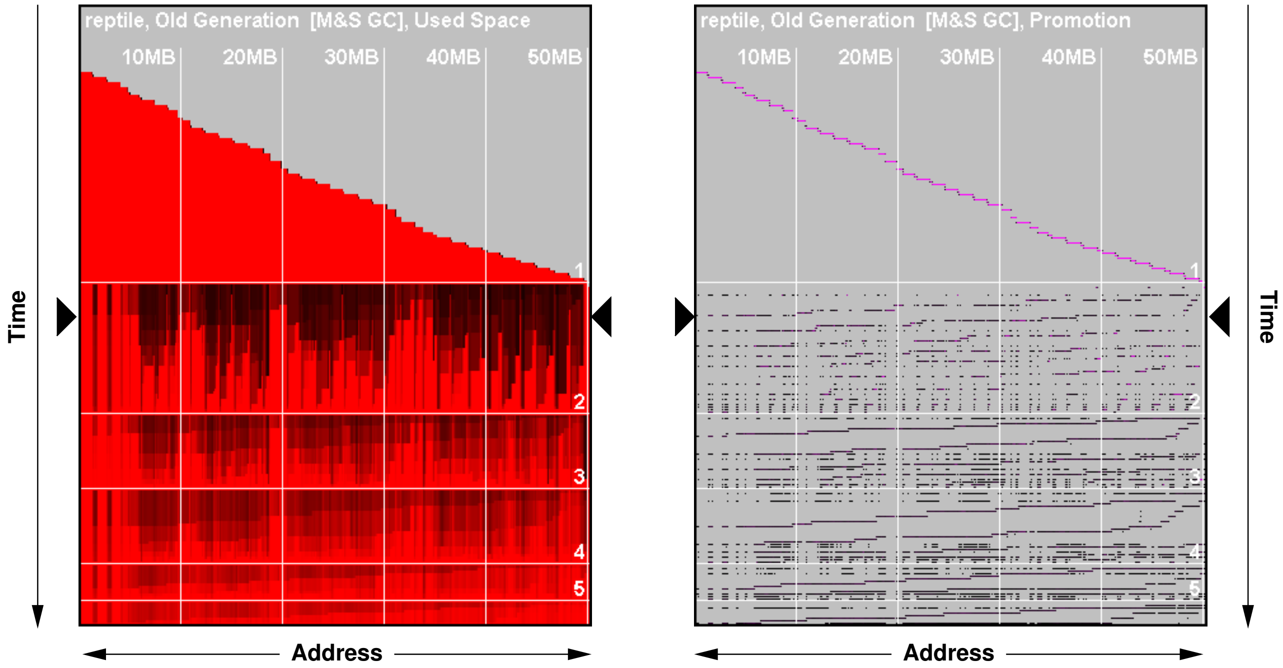


Figure 6: The GCspy user interface main window showing the *Promotion* view of the Old Generation space of RJVM for the `reptile` benchmark. The Free List space is active, a tile on it is selected, and its legend window is also included on the left.



(a) *Used Space* history graph.

(b) *Promotion* history graph.

Figure 7: History graphs showing the *Used Space* and *Promotion* streams of the Mark Sweep space of RJVM for the `reptile` benchmark. The black triangle markers denote the point at which the main window snapshots of Figures 4 and 6 were taken.

bottom of the grid (essentially, the *y*-axis of the graph represents time, starting from zero at the top).

The information provided by GCspy's set of abstractions (described in Section 3) is sufficient for the client to generate history graphs, ensuring consistency between the graphs and the visualisations in the main window without imposing a further burden on the server implementer. History graphs can be customised in several ways (size of the tiles, colour options, vertical separators that group tiles together, horizontal separators that correspond to events in the server and so on). For example, in Figure 7 the vertical separators indicate 10Mb heap increments and the horizontal lines correspond to old generation collections (the garbage collection number also appears on the graph for easier identification). The visualisation model is an extension to the basic model employed by GCspy, making it easy for the user to understand one having seen the other. We describe how main window and history graph views have been used in practice in the next section.

7. GCspy IN PRACTICE

GCspy has been used to study the behaviour of production JVMs where it has met our claims for scalability by visualising heaps of up to 1Gb (using 128Kb blocks and 8,192 tiles). These case studies have improved researchers' and developers' understanding of their collectors and have revealed a number of new insights.

A number of applications were used to explore the visualisation in both case studies. They were:

- `reptile` (see Section 5.8).
- `GCOld`, which creates and manipulates a large tree structure to evaluate the performance of incremental garbage collectors [27].
- `paraffins`, which calculates the paraffin molecules that may be constructed from a given number of carbon atoms. This is a good stress-test for the Train collector because components are constructed through combinations of existing ones, hence some objects (such as that representing CH_3) are very heavily referenced.

The first system studied was RJVM (see Section 5.6). The RJVM generational collector was configured to use two generations, the younger managed by a semi-space copying collector and the older by an in-place de-allocating mark-sweep collector [20].

Figures 4 and 6 show the *Used Space* and *Promotion* views of the `reptile` benchmark. Looking at the Old Generation space, greater space usage within a block and greater amount of promotion to a block are both indicated by a brighter (lighter in monochrome) shading. We can see from the *Used Space* view that blocks at the start and the end of the old generation are heavily used, presumably because those at the start are long-lived objects and those at the end have been comparatively recently promoted and so have not yet been collected. This conjecture is confirmed by the history graph of this space, Figure 7(a), which shows vertical bands of long-lived objects at the left of the graph and a band of objects at the extreme right at the time of the main window snapshots. Observe that these objects are reclaimed by the second old generation (the horizontal white line, labelled '2' in the figure).

There are two points of interest in Figure 6. First, looking at the Free List space, we can see that there are four very popular size classes (and seven unused classes). Furthermore, popular size classes are equally spaced. The reason for this is as follows. The `reptile` application allocates 24-byte objects heavily (the first popular size class). If contiguous objects are freed, their space is coalesced, leading to long free-lists for common multiples of 24 bytes. Second, promotion to the old generation is concentrated on a few blocks, as the free chunks were added to the free-lists in address order, hence all the free chunks in one block will be used up, before the allocator considers free chunks in another block. Figure 7(b) sheds further insight into the behaviour of the old generation allocator. Until old generation collection 1, promotion allocation is done from a single large free chunk that initially spans the entire generation. The diagonal line sloping down indicates which parts of that large free chunk are used. After collection 2, we can observe distinct lines sloping from the upper right (the end of the old generation) down to the lower left (the start of the generation). By this time the heap is fragmented by this application. Free-lists in the old generation maintain the free chunks, which are obtained by sweeping the space from left to right, in a LIFO manner. Thus, the first chunk to be selected for allocation is the rightmost available one of the best matching free-list (predominantly the 24-byte list). Once this list is empty, the allocator splits chunks from larger sized lists, causing the diagonal lines in the history graph to be repeated. We can also observe the effect of this allocation policy on used space in Figure 7(a) as bands sloping down to the left.

The second case study [28] is of the RJVM Train collector, notorious for the subtleties of its operation and for being difficult to tune. That it was possible to incorporate GCspy into such a complex collector reinforces our claims for the generality and adaptability of the framework. A number of applications, known to be problematic for the Train, were studied and several new insights emerged.

- Long-lived data frequently clusters to form a few, very large trains. Commonly, in both `reptile` and `GCOld`, objects were copied to cars at the end of the same train. As that train became older, GCspy showed that objects continued to be clustered in it. History graphs obtained from `GCOld` showed this pattern particularly clearly. Over time, a single data structure caused an increasingly larger number of objects to be clustered with it, and that train took longer and longer to collect. These patterns are repeated in other applications and show that collection policies dependent on an assumption of short trains [31] are not well-founded.

- Remembered set maintenance in the Train collector may be costly. Visualisation offered an opportunity to compare techniques for tuning remembered sets. One technique to reduce costs is to isolate heavily referenced ('popular') objects in a car of their own. GCspy visualisation of popular objects has confirmed that they are comparatively rare but nevertheless important to handle specially. A technique to increase the capacity of remembered sets is to 'coarsen' them by making each entry represent a larger region of memory. Again, GCspy history graphs (obtained from the `paraffins` benchmark) confirmed that isolating popular objects and coarsening remembered sets distributed entries between remembered sets more evenly (and so helped smooth garbage collection pause times).

History graphs have been particularly instructive but useful information has also been obtained from the GCspy main window.

- Visualisation of the free-lists of the non-relocating collectors of RJVM revealed that every second list was empty. This was caused by the free-lists' assumption of 4-byte object alignment although RJVM had been subsequently modified to align objects at 8-byte boundaries. The presence of empty lists imposed a (small) performance overhead.

- A particular application was observed to be causing very slow young generation collections (sometimes over 300ms).

Visualisation of its card table with GCspy immediately revealed the cause: up to 40,000 cards were being dirtied between each young generation collection, causing the card-scanning code to be a major bottleneck. It turned out that this particular application was a pathological case for card-based generational collectors due to the large number of reference fields being frequently updated.

8. CONCLUSIONS AND FUTURE WORK

This paper has described GCspy, an adaptable heap visualisation framework. The major contributions of this work are that, unlike previous heap visualisation systems, GCspy is not committed to a particular memory manager, it is capable of visualising very large systems and it can attach to and detach from running applications. GCspy is sufficiently generic to be incorporated into different systems with relative ease. This is achieved by its two main abstractions, spaces and streams, in terms of which data can be visualised in a generic manner, and by its client-server architecture: a visualising client connecting to a server embedded in the target system. Only the server-side code needs to be customised for a particular system; this is done through a well-defined driver abstraction. The visualiser remains unchanged but is configured dynamically at connection-time for the particular server to which it has been connected.

The record of heap activity can be stored in a trace file, replayed or shared. Such trace files are very compact when compared to alternative solutions. As well as a sequence of snapshots, GCspy generates history graphs of the behaviour of a particular aspect of a system over a period of time. GCspy can be used for program analysis by runtime system implementers or even by sophisticated application developers. It has proved useful in practice, revealing new insights into the behaviour of several collectors and their interaction with particular applications.

Our claims of generality and ease-of-adoption are demonstrated by the incorporation of GCspy into three very different Java virtual machines, using servers written in C and Java, and the visualisation with appropriate drivers of nine collectors, as well as other components of these systems (such as free-lists). Currently, other ports are complete (the Boehm-Demers-Weiser conservative garbage collector) or underway (the GHC Haskell compiler, the Eclipse Constraint Logic system and the widely-used `dlmalloc` dynamic memory allocator).

In future work, we will concentrate on adding new facilities to the framework. Currently, GCspy visualises data slices orthogonally (i.e. one attribute stream at a time); we intend to explore how to combine streams to show several attributes at once. Further avenues for exploration include rewinding and looping capabilities (useful when trying to discover a problem), tile re-ordering (to allow different logical orderings for the data as alternatives to the usual physical ordering), two-dimensional organisation of a space (in order, for example, to visualise the remembered sets of the Train and Beltway [4] algorithms), and pluggable space renderers that can show alternative views of the data (such as histograms instead of tiles). Additionally, further ways will be sought to improve server performance. These include performing data collection at the server side more incrementally, while still ensuring that the presence of GCspy remains non-disruptive when the visualiser is not connected, and transmitting deltas instead of complete sets of the attribute data in order to decrease the network traffic between the client and the server. Finally, standard tools to perform analysis over the GCspy trace files might also be desirable.

APPENDIX

A. ACKNOWLEDGEMENTS

This work was supported by the EPSRC with grants GR/R57140 and GR/R42252. Tony Printezis is funded through Sun Microsystems' External Research Program. We are grateful to Sun Microsystems for making the source code of their HotSpot JVM product available to us and to IBM for making the Jikes RVM system widely-available under open source. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors. The authors are grateful to the following people for their contributions to this work. Discussions with Kenneth Russell greatly influenced the GCspy architecture, especially its client-server model. Paul Dempster did the port to the Boehm-Demers-Weiser collector. Huw Evans, Alex Garthwaite, Peter Linington, Andy King and Gareth P. McSorley provided useful feedback on drafts of this paper. Andy also provided statistics for earlier versions of this paper. We would also like to thank the anonymous referees for their constructive comments. Finally, Richard Jones is grateful to the University of Glasgow for hosting his visits.

B. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), February 2000.

[2] 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.

[3] A. W. Appel. Simple Generational Garbage Collection and Stack Allocation. *Software — Practice and Experience*, 19(2):171–183, March 1988.

[4] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, pages 153–164, Berlin, June 2002.

[5] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software — Practice and Experience*, pages 807–820, September 1988.

[6] Borland Inc. The OptimizeIt tool. http://www.optimizeit.com [November 12, 2001].

[7] 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.

[8] T. Chilimbi, R. E. Jones, and B. Zorn. Designing a Trace Format for Heap Allocation Events. In T. Hosking, editor, *Proceedings of the 2000 International Symposium on Memory Management*, pages 35–49, Minneapolis, MN, USA, October 2000. ACM Press.

[9] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In R. E. Jones, editor, *ISMM98 Proceedings of the First International Symposium on Memory Management*, pages 37–48, Vancouver, October 1998.

[10] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitski, and H. Srinivasan. Drive-by Analysis of Running Programs. In *Proceedings of the Workshop on Software Visualization, International Conference on Software Engineering*, Toronto, Canada, May 2001.

[11] W. De Pauw and G. Sevitski. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency — Practice and Experience*, 12:1431–1454, 2000.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] R. H. Halstead. Self-describing files + smart modules = parallel program visualisation. In *Theory and Practice of Parallel Programming*, number 907 in Lecture Notes in Computer Science, pages 253–283. Springer-Verlag, 1994.

[14] T. Harris. Dynamic adaptive pre-tenuring. In T. Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN, October 2000.

[15] W. Harrison, H. Ossher, and P. Tarr. Software engineering tools and environments: A roadmap. In *Future of Software Engineering, ICSE 2000*, pages 263–277, Limerick, Eire, 2000.

[16] A. L. Hosking, J. E. B. Moss, and D. Stefanovic. A Comparative Performance Evaluation of Write Barrier Implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992.

[17] 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.

[18] IBM Research. The Jinsight project. //http://www.research.ibm.com/jinsight/ [November 12, 2001].

[19] M. S. Johnstone and P. R. Wilson. The Memory Fragmentation Problem: Solved? In *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 26–36, Vancouver, Canada, October 1998. ACM Press.

[20] 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.

[21] R. E. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[22] 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.

[23] D. Lea. The GNU C++ library. *The C++ Report*, 1993.

[24] D. Lea. A memory allocator. The DLmalloc homepage http://gee.cs.oswego.edu/dl/html/malloc.html [April 4, 2000], 1997.

[25] T. Ngo and J. J. Barton. Debugging by Remote Reflection. In *Proceedings of Euro-Par 2000*, Munich, Germany, August 2000.

[26] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference*, pages 249–257, Keele, March 1993.

[27] 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.

[28] T. Printezis and A. Garthwaite. Visualising the Train Garbage Collector. In D. Detlefs, editor, *Proceedings of the 2002 International Symposium on Memory Managemenr*, pages 50–63, Berlin, Germany, July 2002.

[29] C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[30] 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.

[31] 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.

[32] A. Shalit. *The Dylan Reference Manual*. Addison-Wesley, 1996.

[33] Sitraka Inc. The JProbe Suite. http://www.jprobe.com/ [November 12, 2001].

[34] R. M. Stallman. *Debugging with GDB: The GNU Source-Level Debugger for GDB*. Free Software Foundation, July 2000.

[35] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[36] Sun Microsystems Inc. Java™ Debug Wire Protocol (JDWP). http://java.sun.com/j2se/1.3/docs/guide/jpda/jdwp-spec.html [November 12, 2001].

[37] Sun Microsystems Inc. Java™ Heap Analysis Tool (HAT). http://java.sun.com/people/billf/heap/ [November 12, 2001].

[38] Sun Microsystems Inc. Java™ Platform Debugger Architecture (JPDA). http://java.sun.com/j2se/1.3/docs/guide/jpda/ [November 12, 2001].

[39] Sun Microsystems Inc. Java™ Virtual Machine Profiling Interface (JVMPI). http://java.sun.com/j2se/1.3/docs/guide/jvmpi/ [November 12, 2001].

[40] Sun Microsystems Inc. The Java HotSpot™ Virtual Machine, 2001. Technical White Paper.

[41] D. M. Ungar. Generation Scavenging: a Non-Disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[42] D. M. Ungar and F. Jackson. Tenuring Policies for Generation-Based Storage Reclamation. *ACM SIGPLAN Notices*, 23(11):1–17, 1988.

[43] 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.

[44] D. Wakeling. Compiling Lazy Functional Programs for the Java Virtual Machine. *Journal of Functional Programming*, 9(6):579–603, November 1999.

[45] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College London, August 1997.

[46] D. White and A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1999.

[47] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: a Survey and Critical Review. In *Proceedings of the Second International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[48] M. Wolczko. The Tracing JVM. http://research.sun.com/people/mario/tracing-jvm/ [November 9, 2001].

[49] M. Wolczko. Using a Tracing Java™ Virtual Machine to Gather Data on the Behaviour if Java Programs, March 1999. SML 98-0154.

[50] M. Wolczko. Personal Communication, October 2001.

C. INCORPORATING A SIMPLE DRIVER

Here we outline the implementation of a driver for a simple semi-space collector. This pseudocode description follows very closely our implementation for a collector for Jikes RVM using the UMass GCTk garbage collector toolkit. Construction of the driver was straightforward and used less than 300 lines of code, including comments. The object-oriented features of Java considerably simplified both the construction of the collector and the driver. Note that, to avoid perturbing the data being measured, it is important that no allocation be made in the Java heap. We exploited Jikes RVM's ability to create objects outside the heap but the details are beyond the scope of this paper.

C.1 JVM Changes

To add GCspy to an existing collector it is necessary to modify the collector (i) to initialise the GCspy framework and (ii) communicate its state to GCspy. At boot time, the collector creates a set of events, the server interpreter and a driver for each space.

```
void initialiseGCspy() {
  /* first, initialise the GCspy server,
   * this is shared by all drivers */
  String eventNames[] = { "Semispace GC Start",
                          "Semispace GC End" };
  Events events = new Events(eventNames);
  gcspyServer_ = new ServerInterpreter(
                     "Jikes RVM",
                     events,
                     1 /* max space number */ );

  /* then, initialise each driver */
  gcspyDriver_ = new GCspyDriverSemispace(
                     gcspyServer_,
                     "Semispaces",
                     heapStart_,
                     GRANULARITY,
                     (heapEnd_ - heapStart_)/GRANULARITY );
}
```

Before and after each collection, the collector must gather and transmit data to the visualiser if connected and the event should be transmitted. If required to transmit, the driver is asked to clear the streams before the collector iterates through the heap, passing data to the driver. Finally, the server is told that data gathering is complete.

```
private void gatherGCspyData(
                int semispaceEvent,
                ADDRESS spaceStart,
                ADDRESS spaceEnd) {
  /* first, check whether the event should be transmitted,
   * i.e. whether the client is connected and the event
   * is active (not disabled or skipped) */
  if (gcspyServer_.shouldTransmit(semispaceEvent)) {
    gcspyDriver_.zero(heapEnd_);
    Iterator it = heap_.iterator(spaceStart, spaceEnd);
    while (it.hasNext()) {
      gcspyDriver_.object(it.next());
    }
  }
  /* if the event is active, this will also perform
   * the data transmission --- it also counts the
   * number of events reached to keep the event
   * counters up to date */
  gcspyDriver_.finish(semispaceEvent);
}
```

C.2 Driver Implementation

On creation, the driver sets up the space and its streams. For simplicity, we omit some of the arguments to the constructors.

```
public GCspyDriverSemispace(
                   ServerInterpreter gcspyServer,
                   String name,
                   ADDRESS startAddr,
                   int blocks,
                   int blockSize) {
  startAddr_ = startAddr;
  blockSize_ = blockSize;
  maxTileNum_ = blocks;

  /* first, create a new space */
  space_ = new ServerSpace(
               "Semispace GC", /* space name */
               maxTileNum_,    /* number of tiles */
               1               /* max stream number */ );

  /* then, register the space with the GCspy server */
  spaceID_ = gcspyServer.addServerSpace(space_);

  /* now, create a new stream */
  objectsStream_ = new Stream(
                     "Objects",  /* stream name */
                     maxTileNum_ /* max tile num */ );
  objects_ = objectsStream_.getData();
  objectsSummary_= objectsStream_.getSummary();

  /* and register it with the corresponding space */
  space_.addStream(objectsStream_);

  /* finally, set up the tiles names (this information
   * appears on the visualiser when clicking on a tile) */
  for (int i = 0; i < maxTileNum_; ++i) {
    ADDRESS start = startAddr + (i * blockSize_);
    ADDRESS end = start + blockSize_;
    if (end > endAddr)
      end = endAddr;
    space_.setTileName(i, "["+Integer.toHexString(start)+
                       "-"+Integer.toHexString(end)+")" );
  }
}
```

The rest of the driver's public interface consists of methods to clear the values set in each of its space's streams, to map values supplied by the collector to the appropriate block (this is essentially histogram binning) and to initiate transmission to the visualiser.

```
public void zero (ADDRESS maxAddr) {
  tileNum_ = getTileNum(maxAddr);
  /* resize the space if the number of blocks has changed
   * (i.e. the collector resized the heap) */
  space_.setData(tileNum_);

  /* initialise streams to their default values,
   * passed to the stream during initialisation,
   * and zero the summary information */
  space_.resetData();
}

public void object (ADDRESS addr) {
  /* count the object in the appropriate data slot... */
  ++objects_[getIndex(addr)];
  /* ...and also in the summary value */
  ++objectsSummary_[0];
}
```