Visualising Dynamic Memory Allocators

A.M. Cheadle & A.J. Field,
J.W. Ayres, N. Dunn, R.A. Hayden and J. Nystrom-Persson
Imperial College London
{amc4, ajf}@doc.ic.ac.uk

Abstract

We present generic extensions to the GCspy visualisation framework that make it suitable for tracking the way continuous dynamic memory allocators such as **dlmalloc** or incremental and concurrent garbage collectors make use of heap memory. These extensions include sample-driven client-server communication, incremental stream updates and client-controlled stream update frequency. Additional extensions to the current GCspy client are also described. These include hierarchical driver grouping and hierarchical visualisation, zooming, and the ability to define and view relationships between tiles in different spaces. We also introduce a *heuristics engine* that is responsible for flipping GCspy from its decoupled 'observation' mode to a synchronous 'single-step' mode, and describe a backtrace facility that can trace the server-side call sequence that led to the triggering of a specified event, such as the allocation or freeing of a block of memory. This enables aspects of the allocator (fragmentation, block ordering, splitting and coalescing policies, etc.) to be understood in the context of a particular application and potential optimisations to be identified. The effectiveness of the enhanced framework is demonstrated with a complete integration with **dlmalloc**. The framework is evaluated in terms of both performance and its ability to explore contrived modifications to **dlmalloc**'s coalescing policy.

Categories and Subject Descriptors D.3.4 [*Processors*]: Memory management (garbage collection); C.4 [*Performance of Systems*]: Measurement techniques; D.2.5 [*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, Dynamic memory allocation, Garbage collection, Visualisation of objects.

1. Introduction

Dynamic memory allocators are responsible for the efficient allocation of memory from a program's heap — the area of memory from which runtime data structures are allocated. Understanding and tuning the performance characteristics of these allocators is a complex and challenging task. This is usually achieved by extensive instrumentation based profiling of the allocator (and collector, where there is automatic garbage collection), whilst running a range of benchmarks. There are, however, behavioural characteristics that are best understood visually, especially when determining how they vary over time. For example, the object density of areas of the heap, the extent to which contiguous heap areas have become fragmented, and the distribution of free blocks among segregated free lists.

An established tool for visualising heap usage in the context of garbage-collected systems is GCspy [15]. GCspy provides the capability to track the evolution of the heap visually, by mapping blocks in memory to tiles in a graphical user interface.

In principle, GCspy can also be used to visualise more general memory management systems, e.g. custom allocators and general-purpose allocators, indeed any system that comprises partitioned components in a simple hierarchy. However, broadening its application proves to be more difficult in practice. GCspy was designed for visualising the effects of a garbage collector. In this context the heap state is rendered in response to relatively infrequent garbage collection events, such as the start and/or end of a minor/major collection cycle or a mark/sweep phase. When such events occur, the entire state of one or more heap regions is communicated from server to client via `STREAM` commands, each of which reports the attributes of each block of a given region.

Although the framework can, in principle, be used in a more decoupled "asynchronous" mode, there is currently no internal support for it. For example, asynchronous data capture currently has to be implemented explicitly, e.g. by building a data capture thread that runs concurrently with the application. There is also no automatic support for efficiently tracking high-volume, fine-grain modifications to the heap state. The current stream model supports an "all or nothing" mode of server-client communication that has been tailored for the task of visualising relatively coarse-grained garbage collection events where much of the heap changes between events. When used to track high-volume fine-grained events the current communication model can prove to be prohibitively expensive.

There are also other limitations within the framework which become apparent when attempting to generalise its use. For example, it is presently capable of visualising only two levels of hierarchy, capturing the notion of region and block. This is completely reasonable in the context of a garbage collector as it neatly captures the basic heap memory structures as administered by most collectors. In general, however, other aspects of memory management may involve richer data structures and/or logical connections that form more general hierarchies. The same is true of other visualisation problems for which GCspy might in principle be well suited. When these hierarchies exist, it can be desirable to view different aspects of them at different times and at different resolutions. This avoids 'visual information overload', helps to target visualisation

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.

effort, and can also improve performance by reducing communication and rendering costs.

Despite its current limitations, GCspy provides an excellent basis for the realisation of a general-purpose visualisation tool. It provides a minimally-intrusive client-server framework with many of the visualisation and communication abstractions required.

In this paper we present enhancements to the GCspy framework aimed at extending its scope to include a wide range of memory management systems. This requires significant extensions to the existing server-side architecture that enable it to track, and report, fine-grain high-frequency memory management events efficiently, whilst relieving the software burden associated with instrumentation. In adapting and applying the new framework, we have also identified and implemented a number of useful enhancements to the client-side visualisation, some of which have important implications for the server architecture. Two significant new features are the ability to define *triggers*, that are designed to fire in response to specific events generated by the application, and the ability to report *backtraces* that record the sequence of function calls that led to the firing of a particular trigger. These new facilities equip the framework with additional performance debugging capabilities that can help to correlate observed behaviour with call sites in the application.

Using GCspy as a starting point avoids 'reinventing the wheel' and creates a single integrated tool, rather than another similar, yet disparate tool. Our hope is that this work will help to promote the adoption of GCspy as the de facto memory management tool.

This paper serves to document the enhanced framework, and to evaluate various aspects of its performance and usability. A description of how the new framework can be used to describe and visualise the key structures maintained by Doug Lea's dlmalloc is also presented. In order to evaluate qualitative aspects of the enhanced framework, we also document an experiment aimed at exploring aspects of dlmalloc's coalescing policy. This is not intended as a full-scale investigation, but rather to illustrate potentially useful aspects of the new framework.

The paper makes the following contributions:

- We present enhancements to the GCspy framework to facilitate the instrumentation of systems with high-frequency events, such as custom, and general-purpose memory allocators, and concurrent garbage collectors (Section 3).
- We present a number of general enhancements to the framework that renders it suitable for visualising structures that form rich physical and/or logical hierarchies (Section 4).
- We extend the capacity of the framework as a performance debugger with backtrace and event-driven monitoring capabilities (Section 4.4).
- We present performance results evaluating the enhanced GCspy in part using dlmalloc benchmarks designed to stress the framework (Section 6). Details of the dlmalloc visualisation are presented in Section 5.
- We explore the utility of the new framework by showing how it can be used to understand qualitatively the effect of a contrived modification to dlmalloc's coalescing policy (Section 7).

The source code for the enhanced GCspy framework and the instrumented dlmalloc (version 2.8.3) is freely available and can be obtained from [3].

2. Background

Most modern programming languages support some notion of *dynamic* memory management in which memory can be claimed and released when required at run time. Memory is acquired from the *allocator* either by an explicit request for a contiguous block of a specified size (e.g. malloc) or by a request to build a new object of a specified type (e.g. new). Previously allocated memory that is no longer required by the program can be released to the memory manager either implicitly (and automatically) by a *garbage collector* or explicitly by calling a general-purpose *deallocation* function (e.g. free).

2.1 Garbage Collection

A comprehensive review of garbage collection techniques can be found in [2]. Most collectors fall under the heading of "stop the world", in which program execution is halted whilst the collector runs. However, where there are real-time constraints, the execution of the program and collector can be interleaved in such a way as to bound the inherent *pauses* that arise in program execution and ensure a minimum level of useful progress [12, 13, 14]. From the point of view of this paper the distinction is potentially of some importance: stop-the-world collectors induce coarse-grained memory management events relatively infrequently; concurrent collectors induce fine-grained events which occur with much higher frequency. In some contexts it may be desirable to monitor fine-grained collection events as they occur.

2.2 General-purpose Allocators

The distinguishing feature of a general-purpose allocator (we drop the word 'deallocator' although the allocator and deallocator go hand-in-hand) is the use of explicit function calls to allocate and deallocate memory, for example malloc/new and free/delete in C/C++ respectively. Explicit deallocation involves releasing the space occupied by an object so that it can be used to satisfy a subsequent allocation request. This typically involves returning the object to a free list/bin which in turn may result in its associated block being coalesced with its adjacent free blocks, if they themselves are free. Details of specific general-purpose allocator/deallocator architectures can be found in [1, 4, 5, 6].

In order to clarify some of the issues involved in allocator design, and to set the scene for the rest of the paper, we now highlight the internal structure of Doug Lea's dlmalloc, which is a popular implementation of `malloc()` and `free()` used in nearly all C programs for dynamic memory allocation. Section 5 describes a visualisation of dlmalloc using our enhanced GCspy framework.

2.2.1 dlmalloc

dlmalloc, and allocators derived directly from it, are included in a number of Linux distributions as the native user-space allocator and are also included in a number of software packages as an implementation of `malloc()`, overriding the standard C library's native malloc. Full details of dlmalloc can be found in [1] and the associated source code.

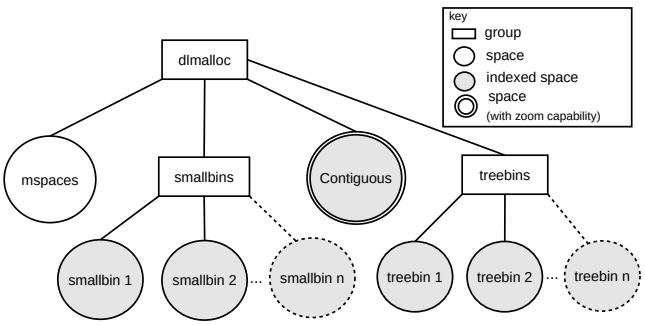

Figure 1. dlmalloc architecture in GCspy group hierarchy form

Allocation within dlmalloc initially proceeds using a bump-pointer allocator with a set of empty free lists. When a chunk of n bytes is requested, it is satisfied by returning the next sequence of n contiguous unallocated bytes, by splitting the top chunk which represents all of the currently available memory. When a free occurs the freed chunk is added to a free list because chunks are not generally freed in the same order as they are allocated. Future allocation requests are serviced first by free list chunks and, failing that, bump-pointer allocation resumes.

dlmalloc segregates its set of free lists into *smallbins*, that contain blocks capable of servicing allocation requests of up to 256 bytes. There are currently 30 smallbins; a maximum of 32 may be used depending on the machine architecture and the minimum permitted size of a chunk. Each smallbin contains same-sized chunks, i.e. implementing a *strict size segregation* policy. Sizes start at 16 bytes, increasing in 8 byte increments up to the 256 byte maximum. A request for a memory block in the range $[8, 256]$ is rounded up to the nearest 8-byte multiple. If the corresponding smallbin is non-empty then a chunk is removed from it and returned. If it is empty the next largest smallbin is consulted likewise. If that too is empty then an attempt to service the allocation is made using the *designated victim*.

The designated victim, dv, is a pointer to the chunk resulting from the last split operation of an oversized (treebin) chunk. Its role is to maintain spatial locality of reference which increases performance by optimising cache misses. If the size of the dv is greater than that of the request, the dv is split and used. If it isn't, then an allocation attempt is made from the smallbin that contains a large enough chunk, which is then split if necessary. If the request cannot be serviced from the set of smallbins *treebin* allocation is attempted.

A treebin is a collection of all chunks larger than 256 bytes organised into bitwise digital trees (*tries* for short) that are keyed by chunk size. There are 32 such tries and these are segregated in power-of-2 ranges, with two equally spaced treebins for each power of two. For each trie, its power-of-2 range is split in half at each node level with the strictly smaller value as the left child. Same sized chunks reside in a FIFO doubly linked-list within the nodes.

If a request cannot be satisfied by a chunk from a treebin, bump-pointer allocation resumes — the top chunk is split in order to satisfy the request. If there is insufficient free memory available in the top chunk, then a request to the operating system is made to expand the user's memory space into the top chunk.

The structure of dlmalloc has been explained in the context of servicing a small allocation request. The logic for servicing a large request is broadly similar, although there are some differences — see [1] and the associated source code for full details.

In addition to implementing the standard malloc interface, corresponding operations are provided for allocation into segregated user-requested regions known as *mspaces*. Not only does this provide regions which effectively have their own local dlmalloc allocator, but if compiled correctly, a program can use these regions for dlmalloc based allocation without overriding the native malloc implementation of the system.

2.3 GCspy

GCspy [15] was developed for visualising heap memory and, more specifically, the effect of the garbage collector on heap layout. It has been used to analyse a number of production garbage collectors and the way in which the collector interacts with an executing application. Specific execution environments that have been studied using GCspy include Sun's Java HotSpot and ResearchVM (previously known as the 'Exact VM') virtual machines. In addition drivers also exist for systems that serve primarily as research platforms, such as Jikes RVM's MMTk memory management toolkit and the .NET Shared Source Common Language Infrastructure, ROTOR.

The GCspy framework adopts a client-server architecture, the memory manager being visualised acts as the *server* and the GCspy visualisation tool as the *client* connecting to it.

The GCspy client is a generic visualiser for incoming server data. At connection time the client receives *bootstrapping data* which describes the information that the connected server will provide. The user interface adapts to this bootstrap information to display the data being transmitted from the server. An important aspect of the design is that an application can be left largely undisturbed when a client is not connected to it.

A *space* represents a component of the memory management system to be visualised. This could be an area of a heap but may also represent a free list or some other relevant structure. A GCspy server can advertise any number of spaces. The server side implementation of a space, the component that allows communication between a memory manager and the visualisation framework, is known as a *driver*.

A space is partitioned into a number of *blocks*. Blocks allow some visual granularity to be adopted for a specific space. For example each block could represent a specific object or a node in a free list. A contiguous view of memory will typically consist of address-ordered blocks representing some arbitrary sized chunk of memory.

At the client, blocks are rendered as small rectangles called *tiles*, typically coloured according to the intensity of some attribute of the block; example attributes include the number of used/free bytes, number of objects, etc, within the block, and are defined by the driver implementer. Each block of a space can be associated with an arbitrary number of attributes.

A *stream* represents the values of some attribute for each block of a space at a point of transmission from the server to the client. Each space can have any number of streams (and thus each block any number of attributes) and different spaces can have different streams.

Streams have an associated textual descriptor which the driver implementer can use to provide extra summary information about a stream that cannot be derived from the values contained within the stream. To allow the visualiser to represent correctly the tiles' values (e.g. by shading them), the set of permissible attribute values is also sent by the server to the client.

The client generates a representation of the tiles in an area of the display window corresponding to a space, with one rectangular box per tile (block) rendered according to a currently selected attribute associated with the stream.

2.3.1 Data Capture

GCspy does not define a data collection method – this is left to the system implementer. When visualising a garbage collector, data is typically gathered by sweeping over each collector component (heap space, region, etc.) at specified garbage collection *events*. Stream data is then assembled from scratch and the entire stream is sent to the client. This constitutes a *synchronous* mode of operation where the application is essentially "paused" whilst the information that is to be sent to the client is assembled. For a "stop-the-world" tracing collector, events are typically associated with each phase of a collection cycle (e.g. *start collection*, *end collection*). In a more complex collector, such as the incremental train collector [16], events are associated with the start and end of young generation and each train collection cycles [17]. It is worth noting that although [17] visualises an incremental collector, each collection cycle appears instantaneous — a cycle is the finest granularity of heap visualisation.

This synchronous mode of operation is inappropriate when visualising general-purpose allocators where the events (e.g. `malloc` and `free`) occur at substantially higher frequencies than, for example, garbage collection events and result in very small changes to the heap state. For example, if the naive streaming model is used to visualise a `dlmalloc` application, where event rates of the order of 100,000 per second are not unusual, the data capture and communication overheads invariably render the tool unusable.

As an alternative, the implementer is at liberty to perform data capture *asynchronously*, for example using a separate thread to perform data capture concurrently with the application. It is also possible to arrange for communication with the client to be performed periodically, e.g. after specified intervals of time, in order to maintain a satisfactory client update rate whilst reducing overall communication.

This "manual" approach to handling asynchronous data capture has three drawbacks: Firstly, there is no internal support for it, so the development effort, which is substantial, would need to be replicated each time a similar asynchronous mode of operation is required. Secondly, the framework does not support the notion of incremental stream updates: only whole streams can be transmitted to the client. This is potentially expensive when performing periodic client updates as only a small proportion of the stream values may have changed since the previous update. The implementer could choose to partition streams into smaller ones so as to reduce communication costs, but this has to be done manually and may lead to a less-than-ideal client-side "model" of the actual heap layout. Thirdly, external data capture complicates and potentially hinders the interaction of the server, its drivers, and the client when detecting undesirable behaviour and pausing the application within close proximity. This interaction is heavily utilised in equipping GCspy with the *trigger* functionality that allows it to be used as an effective performance debugger.

The first set of enhancements we now describe are designed to overcome these problems and to provide more control over the coupling between server and client. In Section 4 we document a range of additional features that enhance GCspy's visualisation and performance debugging capabilities.

3. Enhancing GCspy I: Sample-based Visualisation

Improving GCspy's ability to visualise dynamic memory allocators, in terms of both efficiency and productivity, involves modifying the server-side architecture and server-client communication models, as we now describe. Enhancements to the client-side visualisation features are considered separately in Section 4.

3.1 Stream Control

To provide efficient support for fine-grained high-frequency event handling within the GCspy framework, we have modified the stream control mechanisms in two important ways.

Firstly, we have added a new command (`INCR_STREAM`) that enables the server to report incremental updates to a stream, rather than specifying the complete set of block (tile) attributes associated with the stream. Secondly, we have added support for *sample-driven* stream control in which stream updates are reported periodically to the client at intervals (the *sampling interval*) controlled by a slider in the client GUI. Together these provide an asynchronous mode of operation that is *internal* to the framework.

In order for the client to be updated correctly, incremental stream updates that occur between client updates need to be buffered at the server. One buffer is required for each stream and server-side buffer management code keeps track of which streams, and which blocks within each stream, have been 'dirtied' since the last client communication. At each client update only dirtied blocks are flushed to the client.

The sampling interval controls two factors simultaneously. Firstly, it influences the visualisation "frame rate" since, in principle, the smaller the interval between successive updates the higher the frame rate that can be achieved. At the same time it also controls the amount of 'decoupling' between client and server. The longer the time between updates the further the application can get ahead of the client, in terms of the current state of the client-side visualisation.

Notice that the sampling rate also influences the volume of communication required to update the client because increasing the time between updates increases the average number of blocks that will have been dirtied by the application in the intervening time. There is thus a performance tradeoff between update communication overheads and frame rate which we evaluate in more detail in Section 6.

3.2 Server Thread Model

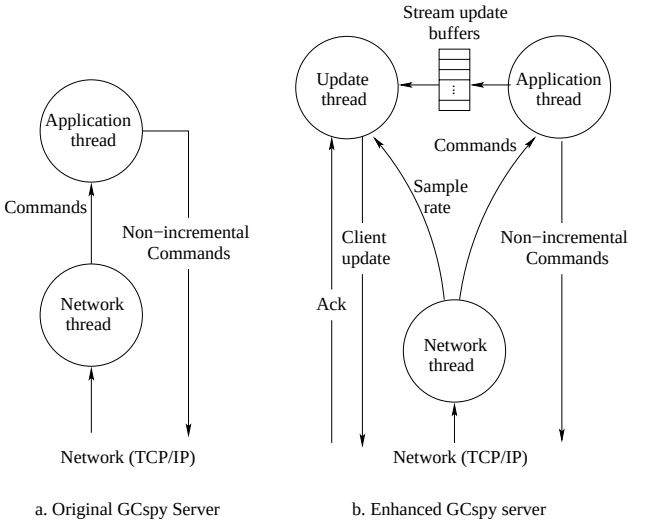

Figure 2. Server Architecture

The extensions described above require a number of changes to the thread model in the existing GCspy server. Currently, the server has just two threads: an application thread that controls program execution and all server-client communication, and a network thread that is responsible for processing commands sent from the client to the server, e.g. connect, disconnect, etc. This is shown in Figure 2(a).

The enhanced server architecture is shown in Figure 2(b). Commands sent from the client to the server are handled by the *Network* thread as before and GCspy's existing (non-incremental) commands are sent from the server to the client as originally prescribed.

3.2.1 Events

In the current version of GCspy an event causes the state of each space's stream to be transmitted to the client. This is no longer the case. The default behaviour is that an event causes the server-side buffers to be updated. These are then flushed to the client, at the specified interval, by an *Update* thread. The *Update* thread keeps a count of the number of occurrences of each such event, which it reports to the client at the update instances. The client is thus able to display event counts that are both accurate and consistent with respect to the current visualisation, even though the counts themselves are only issued from the server periodically.

Note that events generated by the application thread do sometimes need to be reported to the client as they occur. A specific example arises when handling *triggers*, which are the subject of Section 4.4. We will defer discussion of this event reporting mechanism until then as it requires an understanding of the notion of event *attributes*, described later (Section 4.4.1).

3.2.2 Updates

The *Update* thread periodically locks and flushes the server-side buffers to the client (by sending `INCR_STREAM` commands) at a rate specified by the sampling interval. Importantly, the *Update* thread waits to receive an acknowledgement from the client before allowing the application access to the stream buffers, by releasing a buffer lock. This acknowledgement is sent by the client when it has completed the redrawing of the GUI – this ensures that the client-side visualisation at that point reflects accurately the current state of the heap within the application.

Note that a separate semaphore ensures that client updates are not issued until there is at least one item within the stream update buffers. Thus, the actual update rate may also be affected by the application event rate in extreme cases (the *Update* thread may have to wait for the next event). When the sampling interval is zero, each update may report as little as one block attribute to the client.

An important feature of the enhanced server is that synchronous non-incremental event handling is identical to that of the current GCspy. Therefore, the performance of the new framework differs almost insignificantly from that of GCspy when used to visualise garbage collectors in the conventional manner. The new framework can thus straightforwardly replace the current GCspy without suffering a performance hit, even though the additional features may not be required or exploited.

3.3 Space Management

In the current version of GCspy, the number of blocks in a space can be modified at runtime. However, this requires the associated driver to be re-serialised, which results in its entire configuration to be re-sent to the client, even though only the number of tiles has changed. On the client side, the associated space, its data-structures and its widgets are discarded, which creates garbage and regenerates objects unnecessarily.

Our experience with dlmalloc (Section 5) shows that tile addition and removal are very frequent operations in dynamic memory allocators. In particular, the number of nodes in a free list may change at every allocation/deallocation.

To remove the need to re-serialise the drivers on each space modification, we have implemented commands for adding (`ADD_TILES`) and removing (`REMOVE_TILES`) a specified number of tiles from a space. For sample-driven streams these commands are issued to the *Update* thread (see above) and have the effect of modifying the structure of the stream update buffers (Figure 2). When these buffers are flushed during a client update the change in structure must also be related to the client in order that the associated stream can be rendered correctly in the GUI.

4. Enhancing GCspy II: Client Functionality

4.1 Driver Grouping

When modelling an allocator it is intuitive to map each free list to its own individual GCspy space and also highly desirable to group related spaces. We have therefore added the capability to group drivers/spaces hierarchically, so that the visualisation reflects accurately the conceptual components of a system. For example, free lists can be grouped into smallbin and treebin sets. We allow spaces to be collapsed so that uninteresting ones maybe hidden from display, thus eliminating 'visual overload' and increasing *both* client and server throughput. A vertical scrollbar has also been added to address the problem of rendering large numbers of spaces simultaneously. This, together with the ability to collapse drivers provides an element of user control over the volume of data that is in view at any time.

Note that when a space is collapsed, updates to the associated stream are not reported to the client. The server-side stream buffers described in Section 3.1 naturally support these collapsible drivers as they serve to cache changes to the associated spaces even though those changes are not being reported to the client. To facilitate this we have implemented a new stream command, `COLLAPSE`, which the client uses to notify the server when a space has been collapsed or uncollapsed. A space is reported as being collapsed either if it is itself collapsed, or if any of its parent groups has been collapsed. A space is reported as being uncollapsed if it is uncollapsed and all of its parent groups are uncollapsed.

4.2 Ranges

We have added a feature that allows relationships between spaces to be expressed through tile *ranges*, which define the start and end "addresses" of the items contained in the associated block. The addresses are simply identifiers associated with the smallest data units (e.g. bytes, table entries, etc.) of interest. This feature allows the GUI to highlight tiles rendered in different spaces that correspond to the same block of memory. As an example, an allocator may be visualised with a space displaying a contiguous memory view and separate spaces for each free list — the free list display contains memory blocks that also reside in the contiguous memory display.

To support these inter-space relationships we have introduced a *tile range stream* to carry range update information from the server to the client. The tile range stream and GCspy's existing *control stream* are actually implemented as instances of a generic *system stream*, which has also been added to the framework. Such streams (optionally) capitalise on the performance benefits of sample-driven stream transmission, incremental updates and driver collapsing, as for user-defined streams.

4.3 Zooming

A visualised heap maybe on the order of 4GB in size, and with a limited number of tiles that can be displayed on the client, each tile must represent blocks of many (hundreds of) kilobytes. The intensity of the colour of the tile is a general indicator of its population. However, at this grain of resolution it is practically impossible to resolve fine-grained aspects of the heap structure, such as fragmentation. We have therefore incorporated a facility that allows the user to zoom in on specified regions of memory, enabling more detailed analysis of such characteristics to be explored.

To support zooming we maintain the address *range* and *blocking factor* of a space. The blocking factor determines how many addressable data units each block (on the server) and each tile (on the client) represents. The associated driver uses the blocking factor to determine the range of data that needs to be streamed to the client. Thus, although zooming is controlled at the client side, required changes to the data transmission are handled by the server. Successive zoom operations are stacked so that it is possible to zoom-out to the previous zoom level.

4.4 A Heuristics-based Trigger Engine

GCspy was conceived purely as a visualisation tool. However, it is often invaluable to be able to focus visualisation effort in response to particular phenomena observed within the application. For example, the user may be interested in tracking unusually large allocations or in identifying points during execution where fragmentation starts to occur.

To facilitate this we have introduced a *trigger* mechanism, which has the effect of pausing program execution and sending a `FIRE_TRIGGER` command to the client when a specific activity is observed at the server. These commands are processed at the client by invoking code specific to the trigger that has fired.

4.4.1 Event Attributes

The triggering mechanism works by associating *attributes* with each event type. Event attributes are simply named integer values that are registered by the server, in a similar manner to the way events are registered in the current framework. For example, in the case of `dlmalloc` the server-side application may associate the attributes "Location" and "Size" (new block location and size) with a memory allocation event. Information about the various events and event attributes are registered at the server and are reported to the client during initialisation.

Note that in the original GCspy framework the "elapsed time" and the "compensation time" [15] were transmitted explicitly with each event. These are now simply event attributes; an example of how they can be used is given below.

4.4.2 Triggers

Triggers are specified by the user on the client. A trigger comprises five components: an event e, an attribute a, a comparison operation op, an integer threshold t, and an action act. Informally this means: If, at the server, event e occurs and $a\ op\ t$ then pause the application and perform action act on the client. In practice, if both the event and attribute tests succeed the `FIRE_TRIGGER` command is sent to the client and the client-side action is performed as a bi-product of processing that command. Currently, the comparison operations supported are $>, =$ and $<$. Thus, for example, e=`Allocation`, a=`Size`, op=$<$, t=2048 causes a trigger to fire whenever an allocation of less than 2048 bytes occurs. Furthermore, attributes can be associated with parameterised callbacks that deliver the attributes' value at the server. It is therefore possible for the attribute, as specified at the client, to have additional parameters that are used when computing the attribute value. An example is given in the next section.

4.4.3 Actions and Plugins

Plugins are part of the current GCspy framework and we have used this mechanism to define the client-side trigger actions. Indeed actions and plugins are synonymous in the new framework: when you specify a trigger action you are actually identifying which plugin to invoke.

Trigger action plugins are composed of both server- and client-side execution logic that extend the GCspy framework. Server-side, plugins provide the functionality to extract attribute values from the host application, such as the size of an allocation or the contents of arbitrary memory locations. On the client-side, plugins contain the logic that executes as a result of processing the `FIRE_TRIGGER` command.

We have defined three trigger action plugins:

Backtrace Plugin The backtrace plugin opens a window on the client, in which it displays a summary of all events generated by the application since the trigger fired. Associated with each event is a summary of the state of the stack at that execution point. The number of stack frames included in the backtrace is a parameter of the plugin. An example is illustrated in Figure 3 showing six frames per backtrace. We now provide details of how the plugin mechanism works, using the backtrace as an example.

Recall from Section 3.2.1 that events generated by the application are not, by default, sent to the client. So, how did the events shown in Figure 3 reach the client? The answer is that the action

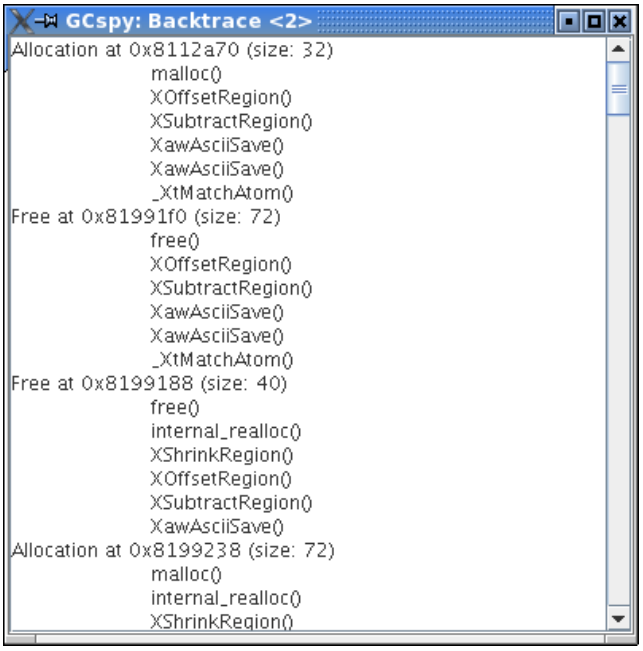

Figure 3. The backtrace plugin

plugin associated with the trigger registers an "interest" in one or more event attributes. The complete list of attributes is known by the server, as explained in Section 4.4.1 above. Typically, the interest list contains those attributes that the plugin needs to perform some client-side activity, for example populating a client window. In the case of the backtrace plugin of Figure 3, the attributes are the location of each allocation/free, the size of each allocation, and the set of six function address strings summarising the call sequence defined by the six frames at the top of the stack.

Before a trigger fires, its associated plugin (trigger action) is disabled. Plugins can however be enabled manually (see below). Although the server is aware of the interest list of each trigger plugin, these interests are essentially disabled at the server. Consequently, events are not sent to the client. As soon as the trigger fires, the associated client-side plugin is enabled, and its plugin window opened, by virtue of receiving the corresponding `FIRE_TRIGGER` command. Opening a plugin window causes its attributes of interest to be enabled at the server[1]. *All* subsequent events are then reported to the client together with the values of the attributes of interest to the plugin. Closing the plugin window has the opposite effect.

As an example, Figure 4 shows the interface through which new triggers are defined. It illustrates four triggers, one of which is set to fire when an allocation of less than 128 bytes occurs; one when an allocation greater than 64KB occurs; one when the memory at heap location 0x5E800000 is modified; and the last when fragmentation thresholds are exceeded (see Section 7). All are enabled, as indicated by the checkboxes. The backtrace plugin illustrated in Figure 3 shows both `Allocation` and `Free` events, even though the backtrace trigger itself was fired by the allocation at location 0x8112a70 (this fired the trigger as less than 128 bytes were allocated).

At the server, the backtrace information is collected using the portable open-source library, libunwind [19]. To limit the performance hit in gathering this information and the size of the data streamed to the client, the symbols and addresses of functions

[1] Actually, the plugin increments a counter for each such attribute and the client automatically reports non-zero counter attributes to the server.

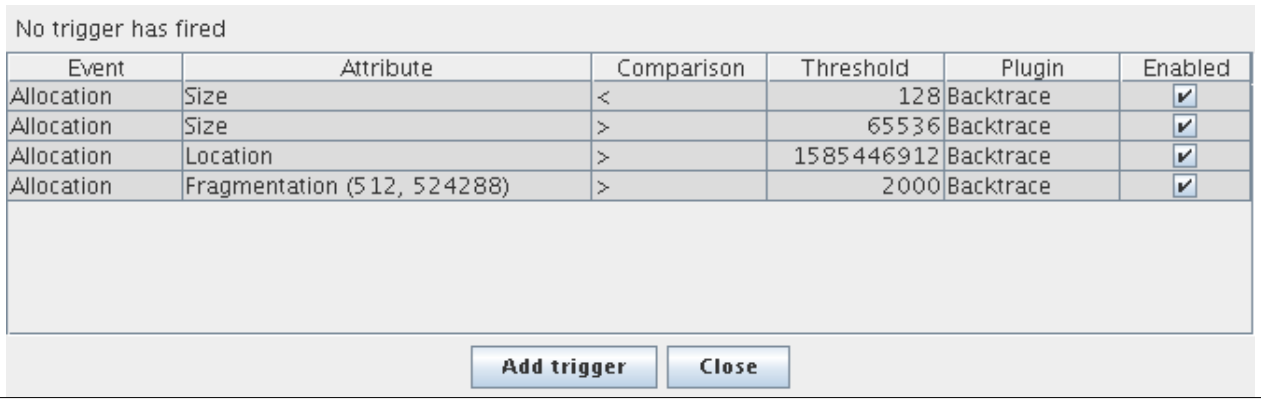

Figure 4. The trigger interface

within the host application, and its shared libraries, are gathered by the server-side plugin at initialisation. The plugin creates a hash table mapping call site addresses to their function names and sends it to the client. Backtrace attributes simply specify the address of the function that generated the associated event and the client uses the hash table to resolve the function name.

Note that when a backtrace window is open a substantial performance penalty is incurred with each event, primarily as a result of the required calls to libunwind. It is possible to open the plugin window independently of any trigger, but this is only recommended when the client is in single-step mode. This is, of course, the case when the plugin is enabled via a trigger as the server is paused when a trigger fires (Section 4.4.2).

Memory Display Plugin In a similar vein to the display of backtrace information, the memory display plugin displays the contents of user-specified memory locations on occurrence of an event. It is worth noting that this event may itself be a watchpoint on the contents of a memory location specified during trigger definition.

Memory Fragmentation Plugin Triggers can be specified that fire when an n byte contiguous piece of memory, is segmented in to x chunks of which the largest chunk is at most y bytes in size. This provides a rudimentary mechanism for the monitoring and diagnosis of excessive fragmentation — Wilson et al. [20] discuss the difficulty in defining a metric that accurately quantifies the amount of fragmentation.

In practice (Section 7) we find that the functionality provided by the trigger-based heuristics engine to be both flexible and effective in achieving our stated goals of performance analysis and problem diagnosis. We define events that "watch" for specific undesired or uncharacteristic behaviour. We use the displayed event count since the program started as a checkpoint and then replay execution to a point several hundred events before the start of this behaviour. We then advance using single-step execution and the backtrace and memory display plugins to assist in more detailed analysis.

4.5 Histories

An important feature of GCspy is the ability to display the evolution of an attribute over time. These *history* graphs are maintained by the client and are updated each time an attribute update is reported to the client. In the original GCspy framework, these were controlled by event commands; in the new framework these updates are reported periodically, at a rate controlled by the client slider.

Note that if the client is disconnected from the server the history trace is interrupted for the duration. We do not log the history within the server as this would require an unbounded amount of memory at the server. Note that, at the time of writing, the same applies when a driver is collapsed. If a history of one of the driver's attributes is being displayed when the driver is collapsed, a similar pause in the history is seen. There is actually no reason why the server could not optionally[2] continue to report attribute values after the driver has been collapsed, but this is currently not supported.

5. Visualising dlmalloc

We have, where necessary, described and motivated many of the features and enhancements we have added, with passing reference to allocator visualisation. In the following section we complete the picture by summarising our integration of GCspy with dlmalloc.

The public set of allocation and deallocation routines of dlmalloc (e.g. `malloc()` and `free()`) have been augmented with GCspy instrumentation. The events are streamed to the client, along with attributes for the size of the (de)allocation request and the address of the block (de)allocated. All streams employ incremental, sample-driven data transmission. The majority of the remaining attributes are defined and managed by the trigger action plugins that have already been discussed.

The GCspy server must initialise (at least under the various versions of Unix) when the *dlmalloc* shared library is loaded into memory. This is achieved using the library's `.init` section, which is invoked by the dynamic linker after all shared libraries have been loaded into memory and before the host application is allowed to run. Prior to this point, it is possible, though unlikely, for the `.init` sections of other libraries to call `malloc()`. It is therefore necessary for our drivers to initially traverse dlmalloc's data structures in order to accurately reflect the true state of the heap.

The GCspy driver/server framework, uses the dynamic allocation routines malloc and free for the allocation of its internal data structures. This is a problem if it is the allocator that provides these functions that is also undergoing visualisation — the allocation events from GCspy will be registered as if they came from the host application. We have therefore modified the server with gcspy_allocator hooks that allow registration of a malloc style allocator that services GCspy's own allocation requests — the dlmalloc integration uses a segregated mspace allocator for this gcspy_allocator.

Figure 1 depicts the hierarchical composition of dlmalloc and the (collapsible) GCspy group and space mappings we apply. Figure 5 shows how this group hierarchy is represented on the client-side user interface. Note that we have configured dlmalloc with only three smallbins and three treebins in order to be able to show the mapping of each of the various entities in a confined space.

[2] A space may be collapsed for reasons of performance — to reduce render times and increase throughput.

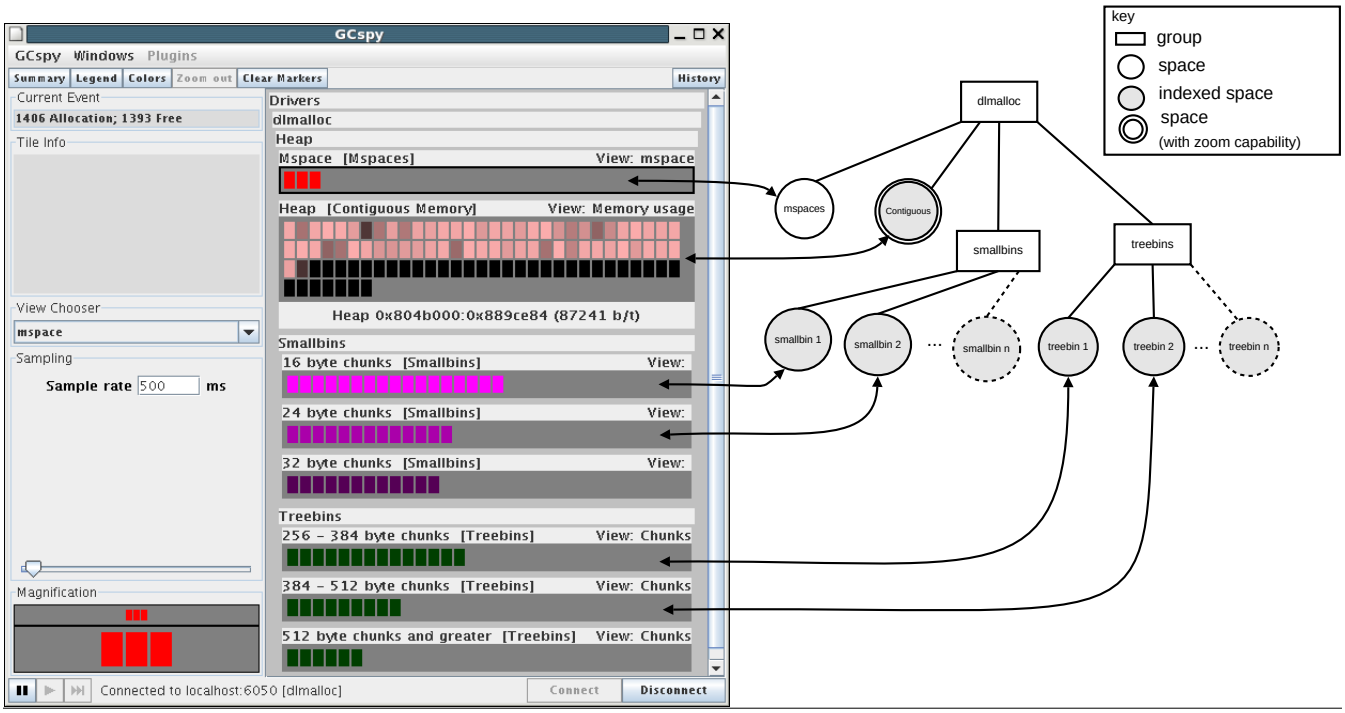

Figure 5. Mapping of dlmalloc's GCspy group hierarchy to client-side user interface widgets

Like previous integrations of GCspy with garbage collected environments, the main space provides a contiguous view of the heap. This facilitates a view of both allocated and free chunks of memory and the zoom capability (Section 4.2) allows us to focus on both densely populated areas and also highly fragmented areas of memory. In addition to this contiguous view, we provide two groups and one further summary space:

Smallbin group A set of spaces, one for each free list, where each (same sized) free chunk on the list is represented by a tile and the order of the list is displayed in the ordering of the tiles. The effect of each smallbin allocation and free event is reflected in the addition and removal of tiles.

Treebin group A set of spaces, one for each trie, where each tile represents a node in the tree and its colour intensity and summary information reflect the size of the block and the length of the free list (of same-sized chunks) at the associated node. The view is that of a flattened tree — the reasons for the choice of this visualisation as opposed to one that shows the complete structure of the tree is explained in Section 8.1. We believe that the most important information is the number of *different* sized blocks in the tree and the number of *same* sized blocks at each node, and this is clearly visualised. The effect of each treebin allocation and free event is reflected in the addition or removal of tiles, or by a change in the tiles' colour intensity.

Mspace summary A single space where each tile represents each individual segregated mspace. Selecting a tile displays the summary information about the mspace. Recall that each mspace effectively has its own local dlmalloc allocator. While it is certainly possible to recursively display contiguous, smallbin, treebin, etc. spaces for each mspace, we have left this unimplemented — mspaces are seldom used in practice, are used for specialised allocation tasks and their characteristics should be better understood in comparison to the heap. Furthermore, we feel that adding what amounts to an individual GCspy server and client to each mspace merely adds to 'visual clutter'. The correct way to perform a detailed mspace visualisation is to substitute the mspace itself for the main dlmalloc heap while the remaining allocations use the native malloc implementation. The allocation or deallocation of an mspace results in the addition or removal of a tile from this space.

Section 4.4.3 has already introduced the additional plugins, and in particular, the memory fragmentation plugin which is arguably one of the most important plugins for the tuning of dynamic memory allocators. In addition, we have added trigger conditions that can be used to "watch" a change in the rate at which events are generated. In the context of dlmalloc, this is most useful in trapping sharp changes in the rate of (de)allocation. The backtrace and memory display plugins can then be employed to help determine where and why the rate has changed so sharply and whether or not it is acceptable behaviour. The history graphing plugin provides graphs of how the spaces change over time, most usefully reflecting the bin lengths throughout execution.

Finally, all the spaces make use of the tile range streams discussed in Section 4.2, in order to relate, via highlights, smallbin and treebin free blocks and mspace regions to the contiguous space and to facilitate zooming.

6. Performance Evaluation

We compared the original and enhanced GCspy performance when visualising the Jikes RVM's MMTk garbage collector, reflected in the experiment reported in [15]. We considered a range of applications from the DaCapo benchmark suite [21]. All applications were run with a 30MB heap with the client and server connected throughout. The client-side visualisation was the same for both GCspy platforms, modulo small differences in the screen layout.

In all benchmarks the client machine was a 2.6 GHz Pentium IV with 1GB RAM, a 4-way set associative 512KB level-2 cache with 64-byte lines, and an 8KB level-1 data and 12KB level-1 instruction cache. The system ran Mandrake Linux 10.2 with a 2.6.13 kernel in single-user mode. Similarly, the server machine was a 2.8 GHz

Pentium IV with 1GB RAM, a 4-way set associative 512KB level-2 cache with 64-byte lines, and an 8KB level-1 data and 12KB level-1 instruction cache. The system ran Mandrake Linux 10.2 with a 2.6.13 kernel in single-user mode.

In each case the execution times differed negligibly (less than 2% for each) confirming that the enhanced server architecture has little or no impact on performance when the framework is used to duplicate "traditional" GCspy visualisations.

6.1 Stream Control

We now consider performance aspects of the sample-based visualisation. Recall that the slider in the client GUI enables the user to control the length of the sampling interval between client updates, as described in Section 3.1.

The sampling interval in part influences the "frame rate" of the visualisation. For example, if the slider is set at 100ms the peak frame rate achievable will be 10 frames per second. Of course, there is a cost associated with issuing an update to the client that includes the communication cost associated with flushing the server-side update buffers to the client, the time taken to render the updates on the client and the time taken to acknowledge the server. In practice, therefore, the achieved frame rate will be less than that suggested by the slider, often substantially.

Increasing the sampling interval allows the server-side application to get ahead of the client-side visualisation. The two can be brought into lock-step, at the event level, by setting the sampling interval to zero. In this case the application will pause at each event for the time it takes to update the client. Although the volume of communication will be very small (typically only one tile will be affected) a full cycle of communication, rendering and acknowledgement must be incurred. As the sampling interval is lengthened, the relative cost of communication and rendering decreases. The average communication volume will increase because the longer the time between updates, the more blocks will have been dirtied in the intervening time. However, the trend will not be linear, as several events may affect the same tile. The communication volume will also be affected by the rate at which the application generates events.

In order to evaluate the performance of the enhanced framework we have developed a custom benchmark that allows us to control the event rate, event mixture, etc. straightforwardly by varying a small number of benchmark parameters. The benchmark produces events (calls to `malloc()` and `free()`) in a tight loop, optionally delaying between each event. At each allocation it `mallocs` a random-sized block of memory up to some specified maximum. The benchmark is parameterised by a random number seed that enables the same random sequence of `mallocs` to be reproduced over different runs. The various results presented were generated using the same random seed.

We vary three parameters during the benchmarking exercise: the sampling interval, the length of the delay between successive allocation/deallocation events in the application and the number of tiles in the client-side visualisation. In each case the maximum amount of memory allocated at each (random) allocation was 50KB. Note that with this parameterisation a tight loop (no delay) equates to a sustained average request rate of around 116,000 events/s, when executed independently of GCspy; with a $50\mu s$ delay between requests the rate drops to around 24,000 events/s. We could, of course, achieve higher rates by reducing the average size of each malloc request.

Note that in real applications (we explored selected standard KDE applications) the allocation/deallocation rates vary significantly from the order of a few hundred per second (e.g. around 400/s for `xcalc`) to a few hundred thousand per second (e.g. around 100,000/s for `Konqueror`); typically the event rate peaks at start-up (e.g. to around 300,000/s for `Konqueror`) before settling.

The results for 2000 and 8000 tiles are shown in Table 1. It should be noted that the 8000 tile experiment is essentially the same as that for 2000 tiles except that the tiles have effectively been split into four. The heap size in each case was 512MB. An initial delay was introduced before commencing instrumentation to allow the client to be bootstrapped by the server and for the Java virtual machine, etc. to initialise itself.

Within each table and for each tile set we consider sampling intervals of 100, 200 and 500 ms and we explore inter-event delays of 0, 50 and 100 μs. In each experiment we measured the average number of tiles that were dirtied between updates and the total time taken to communicate those updates to the client. We also measured the effective frame rate achieved, which is computed as the reciprocal of the average time between successive renderings of the client GUI.

The sampling rates represent an upper bound on the frame rate that can be achieved in each case and thus vary from 2 to 10 fps. Of course, these rates could only be achieved in practice if the update costs were zero.

The figures show that, for these parameterisations, the performance of the framework is dominated by the GUI rendering times. Currently, the rendering cost is dominated by the total number of tiles, rather than the number that have been dirtied. For 2000 tiles, the rendering time is approximately 270ms across all parameterisations; for 8000 tiles it is around 2s, although there is some variability–the minimum time was around 1.8s. Thus, the peak frame rate achievable is around 2.7fps ($1000/(270+100)$, assuming zero communication cost) for 2000 tiles and 0.52 for 8000 tiles. In the experiments performed, the peak rates achieved were 2.63 and 0.52 respectively, both with the shortest sampling interval of 100ms, as would be expected.

Although we are achieving acceptable performance in terms of frame rate, the figures show that we could do substantially better if the rendering could be performed incrementally. For example in the experiment with 8000 tiles, a sampling interval of 100ms and a mean of $50\mu s$ between events, the number of tiles that *need* to be updated is 1601 on average – approximately one fifth of the total. If the GUI could be rendered in approximately one fifth of the current time the frame rate could be increased to around 1.9fps from the current value of 0.49 (the mean rendering time in this experiment was around 1.9s). Incremental rendering is thus a priority for future work.

6.2 Driver Collapsing

Recall from Section 4.1 that the effect of collapsing a driver is to eliminate the need to report updates to tiles that are hidden by virtue of being part of a collapsed space. More significantly, perhaps, as we have just seen, is the fact that the components of a collapsed driver do not need to be rendered on the client GUI. As the renderer constitutes the bottleneck in many cases, collapsing drivers (spaces) will invariably serve to boost the frame rate.

7. Enhanced GCspy in Practice

7.1 Deferring dlmalloc's Coalescing Policy

In order to evaluate qualitatively the usability of GCspy for performance analysis we have synthesised an experiment that aims to explore two possible coalescing strategies in dlmalloc. dlmalloc attempts to coalesce all freed blocks as soon as they are deallocated by inspecting the *boundary tags* [22] of the block's predecessor and successor. This "eager" strategy enables coalescing to be performed in constant-time.

Update interval (ms)	Measurement	2000 tiles			8000 tiles		
		Mean inter-event time (μs)			Mean inter-event time (μs)		
		0	**50**	**100**	**0**	**50**	**100**
100	Mean no. of dirty tiles	1347	566	354	4932	1601	978
	Mean communication time (ms)	14.62	5.04	3.33	151.09	39.87	24.01
	Effective frame rate	2.59	2.65	2.63	0.43	0.49	0.52
200	Mean no. of dirty tiles	1585	938	629	6953	2897	1836
	Mean communication time (ms)	22.62	9.14	5.89	200.81	78.03	44.42
	Effective frame rate	2.07	2.08	2.10	0.40	0.42	0.47
500	Mean no. of dirty tiles	1893	1643	1229	7784	5770	3926
	Mean communication time (ms)	38.17	17.74	10.81	292.20	153.45	102.30
	Effective frame rate	1.32	1.28	1.29	0.35	0.42	0.39

Table 1. Enhanced GCspy performance

Another approach is to defer coalescing, instead returning the block to a free list of the appropriate size. It might then be reallocated from that free list, thus saving on the coalescing time, the time to seek a suitable block and most likely split it. The idea is to perform coalescing later, according to some heuristic — here we choose the proportion of contiguous free space. Coalescing is then performed by iterating over all smallbin and treebin structures.

Which strategy is best for a given benchmark? We would expect dlmalloc to get this right, of course. The objective is not so much to prove or disprove this here (a big task!), but rather to show how our framework might be useful in analysing the two competing strategies.

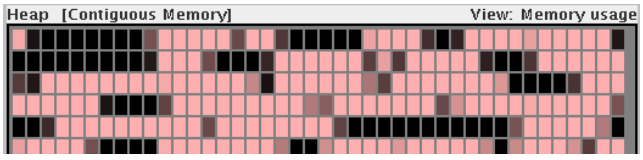

Figure 6. Deferred coalescing results in greater fragmentation

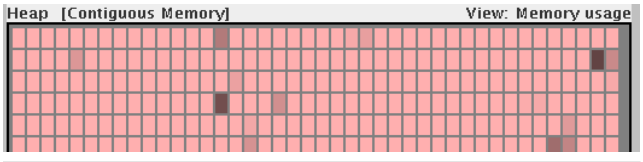

Figure 7. Eager coalescing minimises fragmentation

Figures 6 and 7 show GCspy visualising a benchmark running dlmalloc with and without deferred coalescing.

We use the trigger provided by the memory fragmentation plugin to trigger a transition to single-step mode (displaying backtrace information) whenever a 512KB contiguous block of memory is segmented into strictly more than 2000 chunks of which the largest is at most 512 bytes in size (see Section 4.4.3 to recall the plugin parameters). Figure 4 displays this. We find that the trigger fires soon after execution begins, confirming the presence of at least one highly fragmented block. We record the event count at which the trigger fired. Figure 6 shows a visualisation of this block where we see graphically a high proportion of unused tiles.

We then repeat the experiment with the original dlmalloc (eager coalescing), placing a watchpoint on the previous event count. Figure 7 shows a visualisation of the same block of memory at the same point in the program, as determined by the event count, where we see graphically a much higher proportion of used tiles. Rather unsurprisingly, perhaps, the fragmentation trigger does not fire at all during execution. Deferred coalescing may not always be a bad strategy but for this benchmark it is seen to exhibit a degree of fragmentation that eager coalescing appears to avoid.

8. Conclusions and Future Work

We have demonstrated that the original GCspy framework can be adapted to visualise the behaviour of a general-purpose allocator and have detailed the important enhancements to the GCspy client and server that facilitate this.

The event rates generated by a dlmalloc application can be several orders of magnitude higher than those generated by a garbage collector. Tracking heap updates in anything approaching real time is therefore predicated on the use of incremental sample-driven updates from server to client and on providing client-side visualisation facilities that help reduce communication. We have described an enhanced server architecture which supports such a communication model by locally caching stream updates and reporting them in batches via a difference list, to the client at a user-controllable rate. We have found that this provides satisfactory and "smooth" visualisations with little loss of precision.

Performance evaluation studies have shown that the enhanced framework runs equivalently to the original GCspy when used to visualise a garbage collector in the manner initially intended. Furthermore, using a contrived benchmark, we have explored the effect of sample-driven streams on server-client communication and visualisation frame rate. These experiments have shown that, for a realistic number of tiles, the performance of the framework is limited by the GUI rendering times.

We believe that this work furthers GCspy's promise as a framework for visualising *any* memory management system — a claim made in the original GCspy paper [15] but arguably not fully substantiated at the time. Indeed, other potential applications of the framework hinted at in [3], for example, may now be practical.

8.1 Future Work

At the inception of this work we set out to integrate GCspy with two declarative programming environments: the ECLiPSe Constraint Logic Programming System and GHC, the Glasgow Haskell Compiler. Both these systems have relatively complex allocators and advanced (incremental) garbage collectors. We now believe that GCspy has the necessary functionality required to perform such integrations and, furthermore, to provide useful insights into both memory subsystem and application behaviour.

GCspy could benefit from further enhancement in two specific areas. We struggled for sometime over how to usefully visualise dlmalloc's treebins. Ideally GCspy should visualise each individual trie structure and its internal nodes, and track its structural changes due to each allocation and free event, and rotations result-

ing from tree re-balancing operations. Not only is the visualisation of such tree structures challenging, especially in confined screen real-estate, but the degradation in client-server performance and event throughput is also an issue. Many different types of (hybrid) data-structure may be employed by an allocator in its free list representation. GCspy could benefit from generalised graph visualisation and layout capabilities for the visualisation of such structures. However, this is a complex problem, indeed it has its own dedicated research area and while such projects as AT&T's Graphviz [23] attempt to address many of the issues, it is still not clear the extent to which it is successful. Our experience with Graphviz raises concerns over its scalability and capability in handling online algorithms and incrementally updating graph structures. It is our belief, that *treemaps*, *H-trees* and *bubble trees*, as discussed in [24], provide promising tree visualisation techniques — all satisfactorily perform the visualisation in restricted space. For example, a treemap is contained within a rectangular drawing area. Tree visualisation is performed by cutting the tree at each node depth. The drawing area is then partitioned into rectangles, where the number of rectangles is equal to the number of nodes at that depth, and the area of each rectangle is proportional to the number of children the associated node has. Each rectangle in turn is then recursively partitioned by applying the same method at each individual node. The downside to such approaches is that they are less intuitive than standard two dimensional tree representations — the structure of the tree is not so immediately obvious.

GCspy's heuristics engine is a candidate for significant development. It is obvious that it can benefit from the addition of more complex trigger rules and composition primitives than we have initially provided. Furthermore, over time, it could be backed by a database of past characteristics that have led to, or have been diagnosed as, performance bottlenecks, thus enabling the formation of a primitive expert system.

We have shown here that the critical performance bottleneck in many situations is the client-side renderer. Any improvements in this respect, e.g. by incremental redrawing, would help significantly for very large tile sets. At the same time, however, we remark that driver collapsing goes some way toward alleviating these problems.

Acknowledgments

The authors are grateful to the anonymous reviewers for their comments during the review process.

References

[1] Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[2] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[3] GCspy Team. Enhanced GCspy source code. http://www.experimentalstuff.com/Technologies/GCspy/

[4] Emery D. Berger and Robert D. Blumofe. Hoard: A fast, scalable, and memory-efficient allocator for shared-memory multiprocessors. Technical Report UTCS TR99-22, University of Texas at Austin, November 1999.

[5] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[6] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In OOPSLA [25].

[7] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. AI Memo 569a, MIT, April 1981.

[8] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[9] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[10] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[11] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[12] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[13] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[14] Andrew M. Cheadle, Anthony J. Field, Marlow Simon, Simon L. Peyton-Jones, and Lyndon While. Exploring the barrier to entry — incremental generational garbage collection for Haskell. In Amer Diwan, editor, *ISMM'04 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[15] Tony Printezis and Richard Jones. GCspy: An adaptable heap visualisation framework. In OOPSLA [25], pages 343–358.

[16] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[17] Tony Printezis and Alex Garthwaite. Visualising the Train garbage collector. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 100–105, Berlin, June 2002. ACM Press.

[18] Imperial College GCspy Team. *I Spy with my GCspy*. http://www.doc.ic.ac.uk/~ajf/Research/publications.html

[19] HP Labs. The libunwind project. http://www.hpl.hp.com/research/linux/libunwind/.

[20] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[21] DaCapo Project. The DaCapo benchmark suite (version beta051009). http://osl-www.cs.umass.edu/DaCapo/gcbm.html.

[22] Donald E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms, chapter 2. Addison-Wesley, second edition, 1973.

[23] AT&T Research. Graphviz - graph visualization software. http://www.graphviz.org/.

[24] Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, /2000.

[25] *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA, November 2002. ACM Press.