# Visualizing Dynamic Memory Allocations

Moreta, Sergio; Telea, Alexandru

*Published in:*
EPRINTS-BOOK-TITLE

# Visualizing Dynamic Memory Allocations

Sergio Moreta and Alexandru Telea
*Department of Mathematics and Computer Science*
*Technische Universiteit Eindhoven, the Netherlands*
*s.moreta@student.tue.nl, alext@win.tue.nl*

## Abstract

*We present a visualization tool for dynamic memory allocation information obtained from instrumenting the runtime allocator used by C programs. The goal of the presented visualization techniques is to convey insight in the dynamic behavior of the allocator. The purpose is to help the allocator designers understand how the performance and working of the allocator depend on the actual allocation scenarios in order to optimize its functionality by decreasing fragmentation and improving response time. We use an orthogonal dense pixel layout of time versus memory space which can show tens of thousands of allocation events on a single screen. We enhance the basic idea with several new techniques: antialiased metric bars for detecting high and low activity areas; cushion cursors for checking correlations of multiple views; and a view to show correlation between program structure (functions) and memory allocations. The presented techniques are demonstrated on data from a real application.*

## 1 Introduction

Dynamic memory allocators are an important component of the runtime support of virtually all programming languages. Such allocators are responsible for providing memory blocks on the heap upon requests from the application, as offered by the `malloc` C function and `new` C++ operator. When the memory is not needed any longer, the allocated blocks are returned to the runtime, either via explicit calls to `free` (C) or `delete` (C++) or via garbage collection mechanisms, such as in Java. The design of an efficient memory allocator is of crucial importance for the performance of modern software.

Several performance metrics are relevant in this context. First, the allocator should minimize memory *fragmentation*, i.e. the amount of non-contiguous free memory blocks. This reduces the chance that an allocation request will fail when the free memory is split into many small blocks interleaved with allocated blocks. Second, the allocator should minimize the *waste*, i.e. the amount of memory used for internal management which is not made available to applications. Waste can occur e.g. when only fixed-size blocks are allocated, in which case it is also known as *internal* fragmentation. Third, the allocator should provide a good response time for both allocation and free requests for a wide mix of scenarios, *e.g.* concurrent allocating processes, different block sizes, and request frequencies. In practice, the performance of a memory allocator is measured by logging data on such metrics from an instrumented allocator. The data is next analyzed and the allocator strategy and parameters are tuned accordingly.

However, understanding log data to detect sub-optimal performance and *when* and *why* this occurs is difficult. Typical logs can easily contain hundreds of thousands of high-frequency events. Detecting patterns and correlations in such logs is a daunting task, especially if one does not know what to look for exactly. For example, the total allocated memory is a simple to measure metric which can be monitored by a single numeric value. However, understanding fragmentation patterns is much more difficult.

A different use of analyzing dynamic memory allocator patterns is for studying the behavior of a given program. Incorrect or inefficient behavior can be hard to quantify in concrete queries or metrics, but can be spotted by seeing memory usage patterns. Examples are finding incorrect allocation/deallocation sequences which could lead to memory leaks, dangling pointers, or uninitialized memory reads. Such analyses can be conducted by using automated tools such as Purify [10]. Yet, a visual presentation can be easier to follow, and can show *trends* leading to potential problems, whereas automated analysis typically detects only 'hard' errors.

In this paper, we present an approach for the visual analysis of the behavior of dynamic memory allocators. Our log data is a weakly structured dataset containing hundreds of thousands of (de)allocation events. We use a simple, yet effective visualization able to display tens of thousands of events on a single screen and antialiasing to emphasize high

activity areas. We extend and specialize these core techniques, introduced by us in [8], to get more insight from memory allocations, as follows. We use a new antialiasing to render occupancy metric bars to support separating high from low activity areas (Sec. 4.1.3). We use a new layout to correlate memory visualizations along the address and/or time dimensions; besides allocation metrics, we also show information on the allocating functions; finally, we enhance the correlated views with a new interactive technique, called cushion cursors (Sec. 4.1.4). We demonstrate our techniques by answering several concrete questions on allocation data collected from a real application.

This paper is structured as follows. Section 2 overviews efforts in visualizing memory allocation logs and related event data. Section 3 describes our problem in detail. Section 4 presents our core visualization design and the three new techniques: antialiased occupancy bars that separate activity patterns, correlated views in memory and time, and the cushion cursors. Section 5 describes the findings we obtained from a concrete study performed on real log data. Finally, section 7 summarizes our findings and outlines future research directions.

## 2 Related work

Several methods exist for visualizing dynamic memory allocation data. Event logs, created by code instrumenting and profiling tools [1, 6, 16, 10], record record allocation and deallocation events, and various metrics, e.g. memory fragmentation, occupancy, and block size distribution [5]. Several applications have been developed to visualize such logs. Earlier work include Rivet [3, 2], LynxInsure [7], Polka [12], and the more general TANGO animation framework [11].

A recent development is represented by the GCspy framework [4, 9]. GCspy provides facilities for collection, transmission, storage, and replay of memory management behavior. The set of mechanisms provided for getting to the data is impressive, including user-specified triggers, scalable client-server communication, remote monitoring, and allocator genericity. GCspy was used in conjunction to Java's virtual machine memory manager and also the C/C++ runtime, via the `dlmalloc` allocator. However, GCspy's visualization is rather coarse-grained. Time plots of metrics of interest, such as occupancy, are shown. However, time-dependent metrics show only aggregated facts and little structural insight, so finer-grained visualizations are needed. A second view shows the coarse-grained (e.g. 16 K) memory blocks as a grid of tiles colored by block metrics such as occupancy [9]. However useful, this visualization does not show detailed insight into high-frequency events such as generated when monitoring the C dynamic allocator. Also, this view shows a snapshot of the allocated memory pattern, but does not reveal its evolution in time.

## 3 Problem definition

We aim to analyze the behavior of a C runtime allocator running on an embedded platform. The allocator should be able to serve tens of processes with thousands of `malloc` and `free` calls per second. The allocator manages the memory in a *pool*, partitioned into $B$ fixed-size bins, and an unstructured *heap*. Each bin $b_i$ has a fixed number $N_{b_i}$ of free blocks of equal size $dim_0 < dim_i < dim_B$. A `malloc` request of size $s < dim_B$ is served by allocating a full block in the bin $b_i$ whose block size best fits $s$. If $b_i$ is full or $s > dim_B$, memory is allocated on the heap. (De)allocation events are monitored by instrumented C library functions `malloc` and `free` and saved to a log file which is next visualized. The log contains a set of events $S = \{e_i\}$. An event $e_i$ contains the operation type (allocate or free), the address range $[addr_{low}, addr_{high}]$ affected, the time $t$, the calling process ID, and in which bin (or heap) it is served. A typical log contains hundreds of thousands of events.

Important quality metrics include occupancy, waste and fragmentation. *Occupancy* is the amount of used memory in a bin. *Waste* equals the memory lost because of the fixed block sizes. *Fragmentation* manifests itself by having scattered instead of contiguous free blocks. Typical questions we address are:

- How does fragmentation depend on time and pool?
- How does waste depend on time and pool?
- Which are the largest quasi-compact regions allocated?
- Are the (de)allocations served in the right order?
- How does the allocator speed depend on the operation type and parameters?

To answer these questions, we have developed several visualization techniques. These are presented next.

## 4 Core visualization design

Our visualization design is driven by several goals: scalability, limited cluttering, insight into fine-scale behavior, intuitiveness, and ease of use. Our core visualization shows all events in each bin separately, as follows. We use a 2D Cartesian layout which maps event time $t_j$ and memory addresses to the $x$ and $y$ axes respectively. Hence, every event $e_i$ is an axis-aligned rectangle. This layout has several advantages. It is *compact* (hundreds of thousands of elements can fit on a screen) and *dense* (no screen space is wasted). Empty areas convey actual information, i.e. they show free

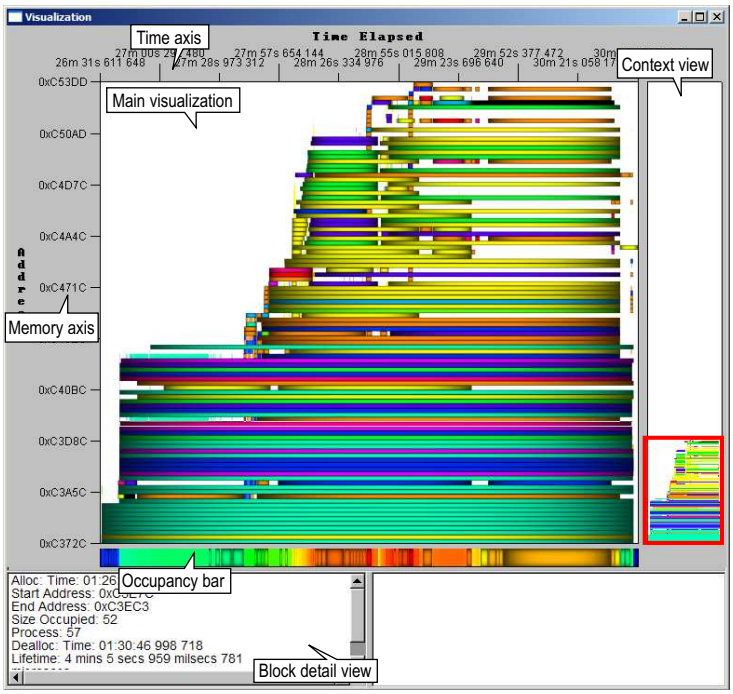


**Figure 1. Interactive tool for visualizing dynamic memory allocations**

memory. Following the $x$ axis, we can see what happens over a given memory range in time. Following the $y$ axis, we see a snapshot of the memory at a given moment. Rectangle sizes show the lifetime and size of blocks. This layout is fast and straightforward to compute. We color every rectangle to show a data attribute $a_j^i$ via a suitable color mapping scheme.

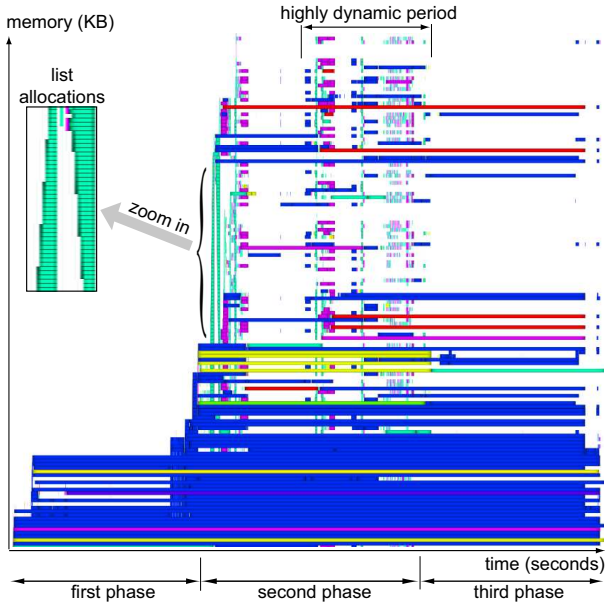


**Figure 2. Visualizing allocations in one bin**

Figure 2 illustrates the basic idea for a memory allocation log dataset containing 119932 allocations spanning a period of 4 minutes done by 54 concurrent processes. Color shows the allocating process ID[1]. This image shows several facts: The "blue" process allocates the most memory. Since the $y$ axis maps to the address space, the long rectangles at the image bottom show that the "blue" process allocates memory early and frees it as last. After start, almost no extra memory is allocated in the first third of the monitored period. Next, the "green" process rapidly allocates many equal-sized blocks, all at one moment, and frees them quickly after, as shown by the thin vertical green stripes. We discovered that this pattern of same-lifetime blocks is typical for *container* objects such as lists. These lists use about a third of the free memory ($y$ axis), so they are quite important. The second third of the period shows a high frequency allocation-freeing pattern which almost fills up the entire memory at some points. In the last third, there are few allocations. All memory is freed in the end.

Figure 1 shows an actual snapshot of our visualization tool. The main view shows the memory dynamics in the currently selected bin. The view can be zoomed and scrolled along the vertical (memory) axis, which is useful when visualizing very large memory spaces (megabytes) or bins with very small block sizes (few kilobytes). To the right of the main view, a *context view* acts like a scrollbar: The complete memory range is visualized, and the user can drag a slider (the red frame) to scroll the view to the area of interest. Under the main view, an *occupancy bar* is dis-

[1]We strongly recommend viewing all figures in full color

played. The bar shows, using a blue (low) to red (high) colormap, the evolution in time of the total memory allocated in the bin shown in the main view. Brushing the main view with the mouse shows details for the block under the mouse, such as the block size, exact allocation and deallocation time, owner process, and function that allocates/deallocates the block.

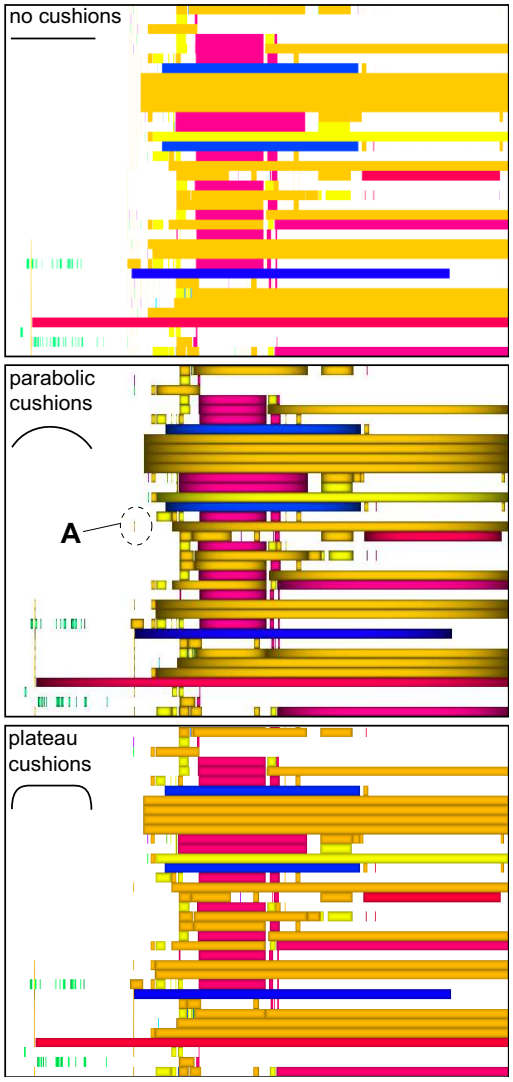


**Figure 3. Block shading. Cushions (mid, low) help showing structure. The cushion profile is shown in the upperleft corner of each view**

## 4.1 Showing Fine Structure

For logs containing hundreds of thousands of events, the basic visualization design discussed before has several problems. We discuss these in turn and present our solutions.

### 4.1.1 Showing Block Structures

If we color blocks using flat shading, same-color neighbor blocks cannot be distinguished (see Fig. 3 top). Drawing line borders works only for zoomed-in views, where the block sizes are larger than several pixels. We solve this problem by overlaying each block using a so-called *shaded cushion*. This is a luminance texture dark at the border and bright in the center. Shaded cushions have been successfully used to emphasize structure over rectangular layouts in many visualization applications, e.g. for file systems [13], repository logs [15], and business data [14]. We use two types of luminance profiles: parabolic and plateau, shown in Figures 3 mid, low. Parabolic cushions show structure better, but may create too dark images for long lifetime blocks. Plateau cushions give a nice (albeit less contrasting) beveled effect, and are in general better. We presented both cushions to a group of about 20-25 test persons. Approximately one half chose for the parabolic and one half for the plateau cushions, so we kept both options.

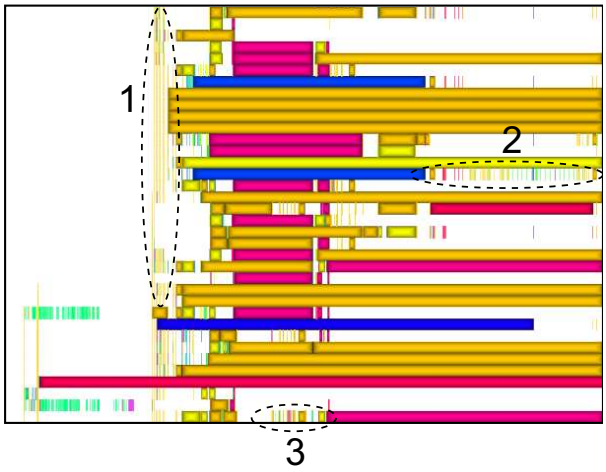


**Figure 4. Using antialiasing, many small-scale events become visible (areas 1,2,3)**

### 4.1.2 Showing Subpixel Structures

Since many (de)allocation events can occur at very high frequencies, the size of an event rectangle can easily become smaller than one pixel. Simply drawing the rectangles produces wrong effects, as a pixel will show a single rectangle, or even no one, depending on round-off errors. For example, in several areas there are memory blocks which seem to 'hang' in the air (e.g. A in Figure 3 mid). This would mean unnecessary fragmentation, since there is free memory below the block (white space) which could be used.

To remove this problem, we must consider how to color a pixel covered by $N$ blocks $e_1, \ldots, e_N$ of which we want to show the attribute values $a_1, \ldots, a_N$. Naive drawing shows

only the color $C$ of the last drawn element $C = c(a_N)$, where $c$ is the attribute-to-color mapping function, or no color, depending on the rounding-off done by the graphics card. A partial answer is to use classical anti-aliasing, i.e. compute $C$ as an average of the colors $c(a_i)$ weighted by the pixel coverage fractions $f_i \leq 1$ of the blocks $e_i$. This answer is not ideal, since very thin *and* isolated blocks, such as the one in Fig. 3 (A) are still hard to see. Seeing such blocks is essential, as it can e.g. indicate the presence or lack of fragmentation.

We solve this visibility problem using an improved antialiasing function:

$$C = \frac{Fc\left(\frac{\sum_{i=1}^{K} f_i^{\alpha} a_i}{F}\right) + Bc_B}{F + B} \qquad (1)$$

where $F = \sum_{i=1}^{N} f_i^{\alpha}$ and $B = \left(1 - \sum_{i=1}^{N} f_i\right)^{\alpha}$ are the pixel fractions covered by block colors and background color $c_B$ respectively and $\alpha > 0$ is a bias factor. The user can tune $\alpha$ interactively. Low $\alpha$ values emphasize areas containing few *and* thin segments. If we compare Fig. 4 rendered with antialiasing to Fig. 3 mid, we clearly see a high number of high-frequency events (areas 1,2,3) which were first invisible.

#### 4.1.3 Visualizing Activity in the Occupancy Bar

The occupancy bar, shown below the tool's main view (Fig. 1), has the same problem as the main view due to high-frequency events. This bar is used to find high and low activity zones, i.e. zones of frequent events, respectively no events. Using a classical color-coded bar (Fig. 5 a) does not show high activity if the occupancy stays relatively constant. An improvement is to draw black bars outlining no-activity zones which are larger than a few pixels (Fig. 5 b). This shows moments when the activity changes *suddenly*, which indicate high strains on the allocator. Yet, this doesn't explicitly show at which side of such a bar high, respectively low activity is. If we add add shaded plateau cushions to the no-activity zones (Fig. 5 c), no-activity zones (cushions) become clearly separated from high-activity ones (flat). An alternative is to draw cushions over all intervals formed by consecutive events $e_i, e_{i+1}$ using antialiasing (Equation 1). This maps the activity to the cushion visual density (Fig. 5 d). Figures 5 e,f show the same idea as in (c,d), this time with parabolic cushions, whose strong contrast makes no-activity areas more salient.

These enhancements of the occupancy bar have been incrementally designed (Fig. 5 a-e) processing user feedback. Although dedicating such minute attention the rendering of the occupancy bar might seem exaggerated, these cushion designs proved very helpful in quickly separating low from high activity areas, the latter being of high interest to the allocator designers.

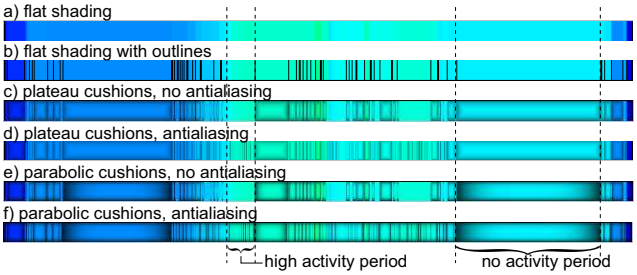


**Figure 5. Occupancy bar rendering: flat shading (a), plateau cushions without/with antialiasing (b,c), and parabolic cushions without/with antialiasing (d,e)**

#### 4.1.4 Correlation of Multiple Views

As already explained, the main view (Fig. 2) shows the allocator activity in a single bin. An important task is to compare different bins to detect possible unbalances, which can be further corrected by adjusting the allocator parameters (per-bin block size, total bin size, or allocation policy). We provide two techniques for correlating multiple bin views. The first technique, called address space correlation, lays out several bin views, scaled to the same size, in a grid layout aligned along the address space ($y$) axis (Fig. 8). This allows correlating the *relative* occupancies of all bins, as discussed in Sec. 5, Task 1. The second technique, called time correlation, uses a similar layout but aligned along the time ($x$) axis (Fig. 9). This allows correlating the activity of all bins, as discussed in Sec. 5, Question 2.

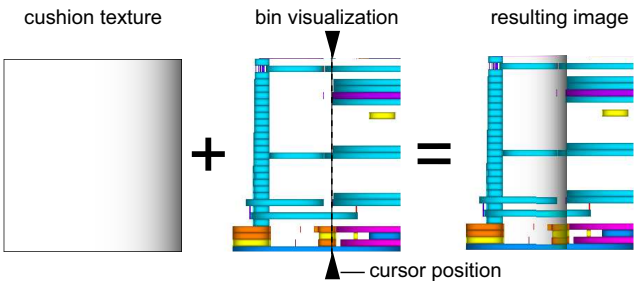


**Figure 7. Cushion cursor construction**

However, just displaying bin views in a grid layout does not allow users to easily see if certain values of interest are indeed correlated. For example, we would like to see whether a high/low activity pattern occurs in the same time in different bins. We support this task by a new technique, called *cushion cursors*. These work as follows.

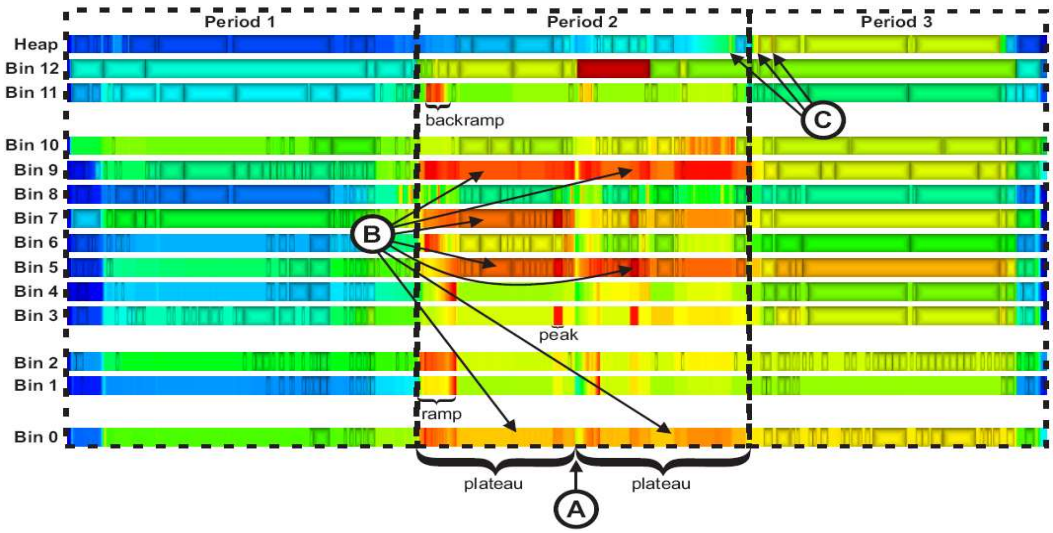


**Figure 6. Visualization of occupancy evolution**

Suppose we want to see if the activity burst in bin 1 beginning at a third of the monitored period, which is visible in both Figures 6 and 8, indeed matches a similar pattern in the other bins. The user can click in bin 1 on the desired pattern in the time correlation layout (Figure 9). A vertical cursor appears at that point, which is drawn over all bin views atop and below. Instead of using a line for the cursor, we draw a shaded cushion texture whose transparency varies horizontally according to a logarithmic profile and blend it atop of the bin views (Fig. 7). More cursors can be placed over a view, whereby several such textures are drawn. Figure 9 shows two groups of three bin views each, with two cushion cursors (i.e. four cushion textures) drawn in each group. Cushion cursors work better than simple line cursors. They are softer, thus disturb the fine details of the image below less than lines. The user can tune their transparency to control softness. Also, they scale better than lines when we draw several of them, nonuniformly spaced, atop of several bin views.

## 5 Applications

In this section, we present some sample analysis tasks and questions supported by our visualization tool.

**Task 1:** *Compare the activity and waste in all bins*
To achieve this, we show all $B = 13$ bins and the heap, correlated in address space, as explained in Section refsec:correlation (Fig. 8). Color shows per-block waste (blue=none, red=maximal). A red bar right of each view shows the free memory in that pool/heap. The black-framed bars under the views show the occupancy evolution in time (blue=all free, red=all full). We see several interesting facts. Bins 1,9 and 12 have the most per-block waste (warm colors) and bins 4 and 5 the least (cold colors). The heap has zero waste (dark blue), which is indeed correct, as the heap doesn't use fixed-size blocks. Statistically, the waste is quite low overall, which is good, except a few allocations in bin 0 (the red horizontal stripes). All in all, the block sizes and best-fit policy perform quite well for the considered test cases. Bins 9,11,12 and 13 are the fullest (shortest vertical red bar). All bins begin with little fragmentation (compact blocks at bottom of all bin views), but end up with a higher one (less compact blocks at top of all bin views).

**Task 2:** *Compare the occupancy all bins*
To get clearer insight in the occupancy evolution, we construct a different visualization which vertically stacks all $B = 13$ occupancy bars so we can compare their evolutions in time (Fig. 6). The 'flat shaded', non-cushioned bar parts rapid allocations followed by deallocations, i.e. short-lived blocks and high allocator activity (Section 4.1.3). During the second third of the monitored period, memory occupancy suddenly increases. Yet, an overall occupancy drop (Fig. 6 A) splits the occupancy patterns of bins 0,5,7 and 9 into two near-constant-occupancy 'plateaus' (Fig. 6 B). In the last third of the monitored period, occupancy decreases. Yet, there are three very short periods where memory occupancy bursts to a maximum in the heap (Fig. 6 C). Our importance-based antialiasing revealed these dangerous moments which would otherwise have passed undetected. Finally, the heap shows a higher block size variation (cushion height) as compared to all bins. This validates again the desired best-fit allocator policy: all blocks which don't fit the bin sizes go into the heap.

**Question 1:** *Are allocations always done from low to high addresses?*
This is apparently the case, if we look at Fig. 8: blocks fill the memory space from bottom to top. However, a closer look at the two list data structures allocated in bin 8 (shown

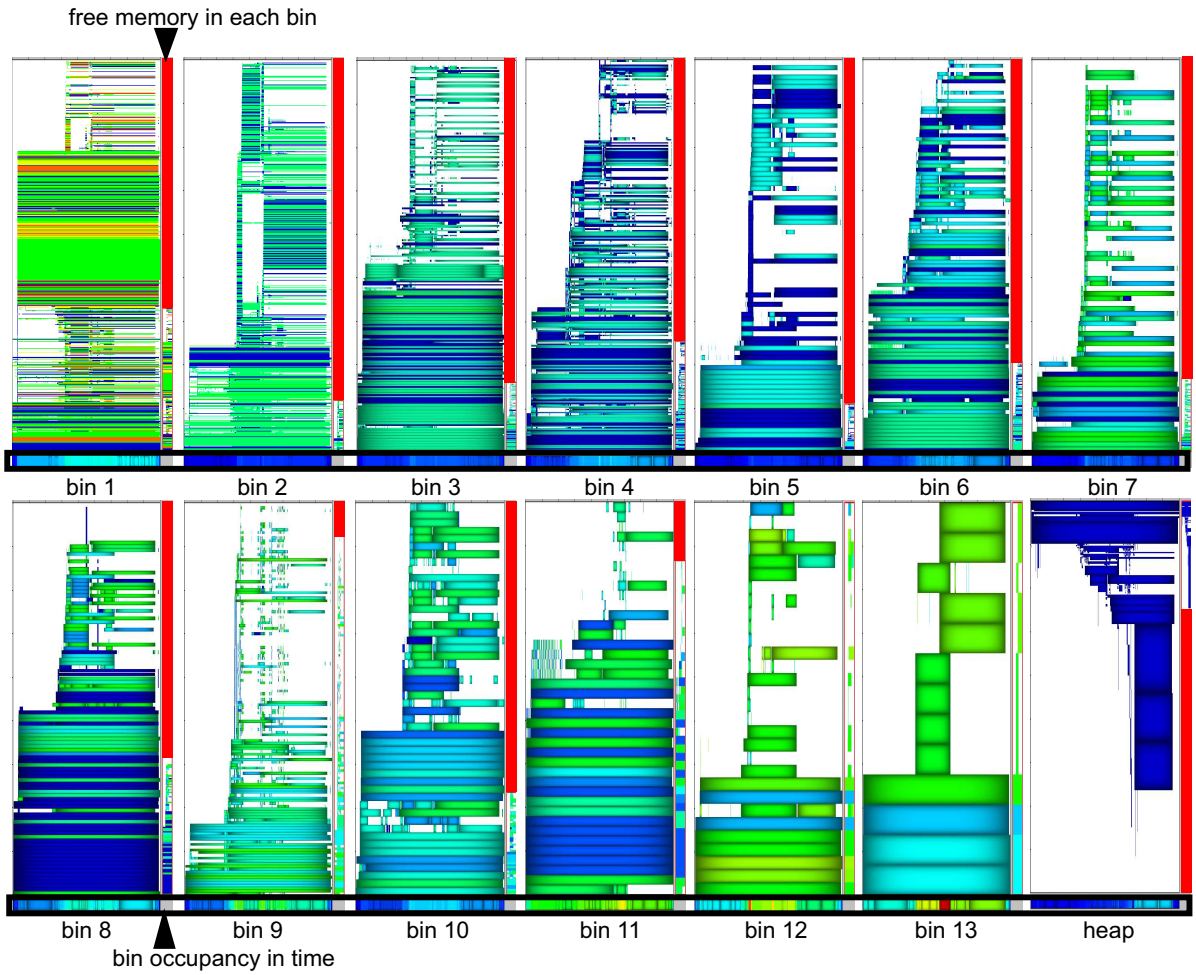


**Figure 8. Bins and heap occupancy visualization. Color shows per-block wasted memory**

in the zoom-in in Fig. 2) tells a different story. The first list (left in the zoom-in) gets allocated from *low* to *high* addresses, as shown by its slightly up-right slanted left side. The second list (right in the zoom-in) gets however allocated from *high* to *low* addresses, as shown by its slightly down-right slanted left side. This finding suggests that the low-to-high allocation invariant is violated here. Finally, we saw that in the heap blocks get allocated from high to low addresses, conversely than for the bins (Fig. 8 lower-right). This is indeed correct for this allocator.

**Question 2:** *Is activity correlated with the memory-allocating functions?*
In our allocator, we log also the addresses of the functions that (de)allocate blocks. We would like to see whether functions correlate with allocation patterns, and how. For this, we map function addresses to a set of unique IDs, and then to colors, and visualize every block colored by its allocating function ID. To answer the question if activity is correlated with allocating functions, we use the time correlation layout (Section 4.1.4) to show different bins. We use the cushion cursors to find out, first of all, if activity patterns in different bins match indeed. Figure 9 shows the result for two groups of three bins each. Clearly, the first high-activity bursts occurring in all bins after the first third of the monitored period is correlated. Color shows us extra information. We see that most blocks being allocated at that precise moment are cyan. This means that a *single* function is responsible for that high-activity burst in *all* bins, i.e. for all memory block sizes. Second, we see that most long lifetime blocks (long horizontal strips) are red, green, and yellow. Hence, these are the functions responsible for persistent data.

Online material about our tool is available for download at: `www.win.tue.nl/~alext/MEMOVIEW`

## 6 Acknowledgements

We are grateful to Christian del Rosso (Nokia Research) for providing us with the case study and with useful feedback on the results of our visualization.

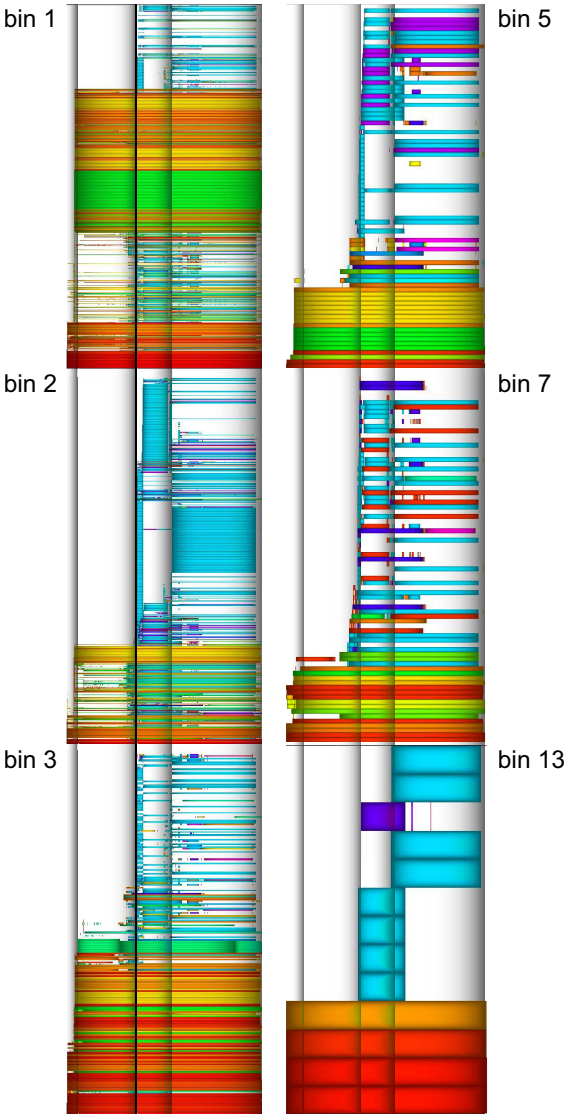


**Figure 9. Correlation of activity and allocating functions using cushion cursors**

## 7 Conclusions

We have presented several new techniques for visualizing dynamic memory allocations. The entire set of techniques revolves around a number of core design principles: orthogonal layouts for simplicity of interpretation; dense pixel displays for scalability; antialiasing for showing subpixel data; and shaded cushions to show correlations and structure in several places (the main memory view, the metric bars, and the interactive user-driven cursors). The combination of these techniques achieves a visualization scalable to hundreds of thousands of elements, as compared to existing approaches [4, 9]. We demonstrate our application by answering several non-trivial questions and analysis tasks on a real memory allocation log dataset. Finally, let us note that the presented techniques are quite generic, so they could be used for other types of time-dependent data than memory allocations.

In the future, we plan to enhance our visualization to also support visual debugging of end-user applications, e.g. by emphasizing memory leaks and relationships between allocating and deallocating code fragments in complex software.

## References

[1] B. Alpern, L. Carter, and T. Selker. Visualizing computer memory architectures. In *Proc. IEEE Visualization*, pages 107–113. IEEE Press, 1990.

[2] R. Bosch. *Using Visualization to Understand the Behavior of Computer Systems*. PhD thesis, Stanford University, 2001.

[3] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: A flexible environment for computer systems visualization. *Computer Graphics*, 34(1), 2000.

[4] A. Cheadle, A. Field, J. Ayres, N. Dunn, R. Hayden, and J. Nystrom-Persson. Visualising dynamic memory allocators. In *Proc. Intl. Symp. on Memory Management*, pages 115–125, 2006.

[5] R. Griswold and R. Townsend. The visualization of dynamic memory management in the icon programming language. In *Tech. Report 89-30*. Dept. of Comp. Science, Univ. of Arizona, Dec. 1989.

[6] C. Jeffery and R. Griswold. A framework for execution monitoring in icon. *Software - Practice and Experience*, 24(11):1025–1049, 1994.

[7] LynuxWorks. The *LynxInsure++* analysis and visualization toolkit, 2006. http://www.lynuxworks.com.

[8] S. Moreta and A. Telea. Multiscale visualization of dynamic software logs. In *Proc. EG/IEEE EuroVis'07*. IEEE Press, to appear, May 2007, 2007.

[9] T. Printezis and R. Jones. *GCspy*: An adaptable heap visualisation framework. In *Proc. OOPSLA*, pages 343–358. ACM Press, 2002.

[10] Rational, Inc. The *Purify* program analysis tool, 2007. http://www.rational.com/purify.

[11] J. Stasko. Animating algorithms with x-tango. *SIGACT News*, 23(2):67–71, 1992.

[12] J. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *J. of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[13] J. van Wijk and H. van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Proc. IEEE InfoVis*, pages 73–78. IEEE Press, 1999.

[14] R. Vliegen, J. J. van Wijk, and E. van der Linden. Visualizing business data with generalized treemaps. *IEEE TVCG (Proc. InfoVis'06)*, 12(5):789–796, 2006.

[15] L. Voinea and A. Telea. Multiscale and multivariate visualizations of software evolution. In *Proc. ACM SoftVis*, pages 47–56, 2006.

[16] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *Proc. ICCS*, pages 440–447, 2004.