Visualizing the Java Heap to Detect Memory Problems

Steven P. Reiss
Department of Computer Science, Brown University, Providence, RI. 02912
spr@cs.brown.edu

Abstract

Many of the problems that occur in long-running systems involve the way that the system uses memory. We have developed a framework for extracting and building a model of the heap from a running Java system. Such a model is only useful if the programmer can extract from it the information they need to understand, find, and eventually fix memory-related problems in their system. This paper describes the visualization strategy we use for interactively displaying the model and related information to achieve these goals.

1. Introduction

Memory-related problems are common in many programs. This is especially true for long-running server-style systems where small problems such as memory leaks or excessive storage overhead can mushroom into major problems over the course of a run. In order to avoid or fix such problems, programmers need to understand how their system uses memory.

Our goal is to provide a visualization of memory that can help the programmer achieve this understanding. Because memory utilization in a large system can be very complex and involved, we needed a visualization that would let the programmer focus on potential problems and on the major components rather than showing all the underlying details.

To this end, one first has to understand what types of problems programmers might be interested in and what are the major components of memory utilization.

The latter is relatively easy. Memory usage that is trivial, say that accounts for less that 0.1% of the heap, is probably not particularly relevant. Note there that one must take into account not only the storage used by a particular object, but also all the storage for which that object is responsible. Moreover, details such as the layout of objects in memory, especially with a language such as Java where the user has no control over the layout, is probably not relevant.

The set of potential problems is more interesting. Some of the problems are obvious. Programmers are first interested in memory bugs. In Java, this is reflected in memory leaks, objects that are no longer needed by the program but which are still referred to by some other object.

A second problem involves inefficient use of memory. Java programmers tend to create several levels of objects between the accessor and the actual data. For example, strings are abstract objects where the actual data is contained in a character array. The intervening objects are not free in terms of memory usage, with each incurring some overhead. If the underlying data is relatively small but there are many instances of it, the result can be that a significant chunk of memory is lost in the overhead.

A third problem involves churn. If the number of objects of a given type is large but remains relatively stable, and if these objects are always being created anew, then the program is spending a lot of time doing possibly unnecessary memory allocation and garbage collection.

A fourth problem involves unexpected increases in memory size. While it is common for the size of memory to fluctuate over the course of a run, the programmer might be surprised if memory usage suddenly jumped or if the objects of a particular class suddenly starting consuming more space than expected.

A related problem involves correlating the memory usage with program behavior. Long running server applications typically experience different levels of demand and go through different phases. It is important that the programmer be able to understand memory behavior in these terms.

Our visualization is designed to let the programmer both understand overall memory behavior and address these and other memory-related problems. In the next section we describe the model of memory behavior we use and how it is built. Then the actual visualization is described. The subsequent section describes how the visualization and its framework address memory problems. We finish by describing related work and our planned future work.

2. Background

Most of these memory problems can be understood in terms of a memory ownership graph. In such a graph there is a node for each object and links from an object to all the other objects for which it is responsible [13]. Creating and using such a graph is problematic. First the graph is impractical large when there are large numbers of objects on the heap. One can't make much

sense of a graph with millions of nodes. Second, the notion of ownership is difficult to determine precisely. An object might be referenced by multiple other objects. In this case one must determine which of these objects is its owner, or, if ownership is effectively shared, what fraction of the ownership should be assigned to each reference. A third problem involves cycles in the graph which typically arise from linked data structures. These make it difficult to assess and hence accurately assign ownership without insights from the programmer.

The typical solution is to do graph manipulations to simplify the graph. This involves both finding strongly connected components to eliminate cycles, and then grouping nodes with similar in links and out links. The result is a much simplified directed acyclic graph [8,13,14,20]. This approach is costly both in terms of the amount of time and space required to get the initial memory graph and the time required to process and analyze a very large graph.

Our approach is to ignore individual objects and to assume that most objects of a given class will be used alike and hence can be grouped. This allows us to start with much less information, notably just the number of objects of each class and the number of references of objects from each class to objects of another class, and to work with much smaller graphs.

One major problem with this simplified approach is that the assumption of similarity, while true for most classes, is not true for the Java collection classes. Java uses a single class to represent an array list, another for a map, another for a linked list, etc. They are used in different ways in different parts of the program. Our approach deals with this problem by creating pseudo classes for each of the collection classes (and the internal classes they use) based on the source reference. Thus, we create what is effectively a new class for a *HashMap* referred to by a *BT_Class* using the name: *HashMap←BT_Class*. Once this is done, our approach tends to yield a graph that is similar to the ones computed by simplifying the complete graph.

A second problem is that not all instances of a class have the same memory footprint. In Java, this only happens for arrays. To deal with this we create new pseudo classes for different sized arrays, where we only consider the order of magnitude of the size. Thus arrays that have between 1,000 and 10,000 elements have *1000 appended to their name; arrays with between 10,000 and 100,000 elements have *10000 appended to their name; and so forth.

We build this model of classes referencing other classes using a JVMTI [12] agent that is part of our controlled dynamic performance analysis tool [23]. This agent periodically uses the JVMTI to walk through the heap counting objects and references. The result is converted into an acyclic graph using a cycle-finding algorithm similar to gprof [6]. This introduces new nodes for cycles, nodes that effectively represent a complex data structure. Finally, we use the relationships in multiple memory samples to statistically allocate objects that have multiple incoming references among their referees using a constrained least-squares fit and quadratic programming.

The result of this analysis is a directed acyclic graph where the nodes correspond to classes, one of the pseudo classes we created, or nodes representing cycles, and the arcs represent memory ownership. The nodes contain information about the class, the number of objects and the size of those objects. The arcs contain information about the number of references and the fractional ownership represented by the arc.

3. Display

We wanted to display this graph in a compact manner so that the programmer could quickly get an overview of the memory behavior. This view had to highlight potential memory problems and provide the programmer with an understanding of memory behavior relevant to types of memory issues discussed earlier.

While we could display the result directly as a graph, the result would be complicated and would not be compact nor easy to understand. The graph is relatively complex, with some nodes, such as that representing the *String* class, being linked to by a relatively large number of nodes, and generally with a total of several hundred or more nodes.

The first step is to convert the directed acyclic graph into a tree which is easier to display. This is done by taking each node that has multiple predecessors and creating copies of that node so that each predecessor has its own copy. While this results in many more nodes, it greatly simplifies the display. Moreover, it allows us to use space efficient representations since arcs can become implicit and do not have to be displayed directly.

The default display has the tree rooted at the left to growing toward the right, as shown in Figure 1. This figure shows the memory usage of a multi-body simulation program using Barnes-Hut. Each block in the diagram contains information as detailed in Figure 2. Vertical size is used to represent the amount of memory owned by the class. The children of a node are displayed to its right starting at the top, and are ordered from top to bottom by the amount of memory they own. This make it easy for the user to see where memory is being used in the application and to follow the ownership relation for a particular class. Moreover, each class has a gap on the right at the bottom of its display that represents the storage used just for the objects of that class. For simple classes like *String*, this shows the overhead inherent in the String object as opposed to its *char* array contents.

Next, we color the nodes based on the fraction of memory that the objects of that class actually occupy

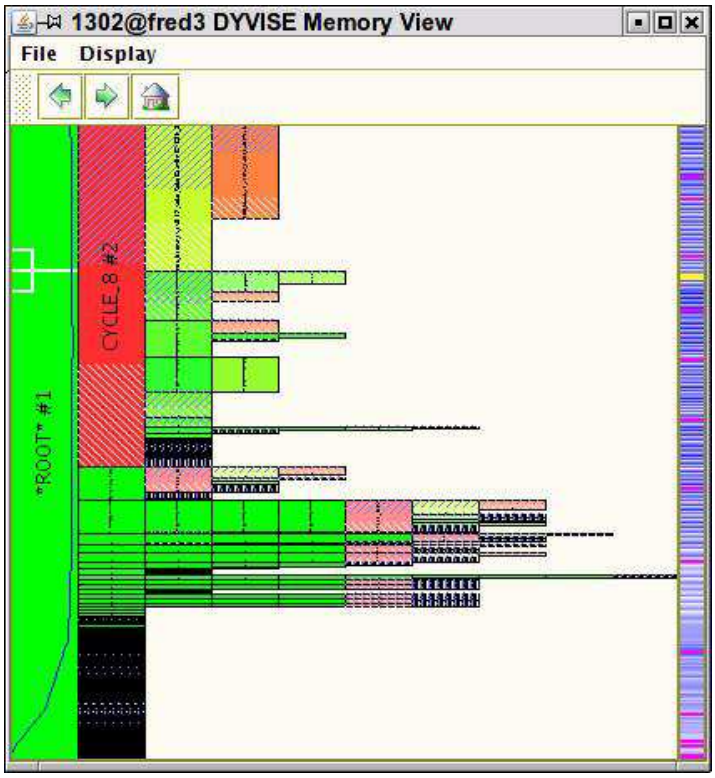

FIGURE 1. Sample memory visualization.

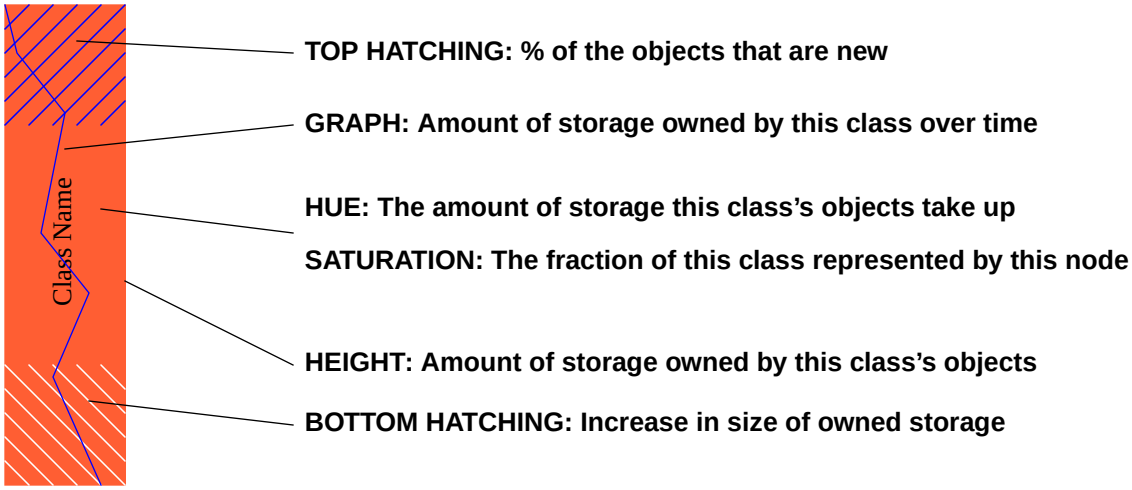

FIGURE 2. Graphical Encodings for a Node

disregarding ownership. Here we use a scale that goes from green to red where red represents the class with the largest local memory usage. We also vary the intensity of the color of a node based on the fraction of the memory that it represents. This means that nodes that are replicated due to the conversion from a graph to a tree are shown in a lighter shade, with the actual shade being dependent on the amount of duplication.

Each of the nodes of the graph is also cross-hatched at the top and bottom to show additional infor-

mation. The blue lines from the top of the bar represent the fraction of the objects represented by the node that are new, that is that were created between the prior heap sample and the current heap sample. The white lines at the bottom of the bar indicate the increase in the memory owned by this class from the prior heap sample. Classes that continually grow will have both white and blue lines. Classes that are undergoing churn, will have blues areas but little white.

Finally, the blue line in the root element shows the history of the size of ownership for that particular element over the multiple heap dumps. In this case, the amount of memory used is going down as objects collide and merge. The white fork within the node shows the time that the heap being visualized was sampled. A fork is used here to indicate that the graph is an approximation with actual data points only at the times of a heap dump. The width of the fork provides an estimate of the accuracy of the graph. The use of a fork also ensures that the line is visible even when it is at the end of the area.

Note that the combination of color and shape helps the programmer find common patterns that represent duplicated subtrees. Also, the use of color brings the programmer's attention to the classes that occupy the most space and the underlying hierarchies show how that space is being used. We considered other tree representations such as tree maps [25], but decided that the simpler unidirectional representation we use was more intuitive for the programmer.

We restrict the display to the set of relevant nodes. In this case, we discard all those classes whose objects that take up less than a user-settable fraction of memory, currently 0.1%. This tends to eliminate clutter on the display while not hiding problems and the items of interest.

We also provide the user with the option of displaying only additional relevant types of root nodes. Our data collection process separates initial pointers based on the class of static fields, the thread id for stack variables, and whether the initial references is a system or a user reference. The user can enable or disable the display of static class nodes, thread nodes, or a global system node. If the nodes are not displayed, the references are accumulated in the root node, if they are displayed they are shown as immediate children of the root node.

The bar on the right of the display is a time line showing heap memory usage over time as reported by the Java management facilities. Dark blue indicates more memory, light blue less. The pattern of dark and light here shows the garbage collector in action. The magenta lines in the otherwise blue display indicate when the actual heap samples were gathered. The yellow line indicates the currently selected user time. This time bar can be used to go back and investigate the state of the heap earlier in the run. The time line and the point at which the heap is displayed is also synchronized with the displays and time line provided by our performance analysis tool. We considered using a graph here rather than light and dark, but decided that the limited space, the approximate nature of the data, and the potentially large number of data points would be better served with the current display.

The system is designed to display the state of the heap live as the program runs as well as letting the user review the history of the run. When displaying the current state of memory, there is no yellow line in the time bar and the current display is periodically and automatically updated.

The display is designed to be interactive. As the user moves the mouse around the display, tool tips provide more detailed information about the various nodes. This can be seen in the first part of Figure 3. From here the user notes that CYCLE_8 represents about three-quarters of the heap. Looking at the pattern of colors, one can see several instances of this cycle in the overall tree. A permanent window with this and additional information can be obtained by middle-clicking on the node as seen in Figure 4. Here the three graphs show local (red) and total size and counts. The time line also provides tool tips, with the tool tip showing both the actual time and the heap memory usage.

The user can zoom in on a particular node such as CYCLE_8 by clicking on it. This produces the display show in the second part of the figure. The resultant display is the tree induced by looking at that node in the original directed acyclic graph and thus includes all duplicates of the node from the original display. Note that the newly selected root now has its time-usage drawn as a graph. The history buttons at the top let the user go back and forth among these specialized views.

The tree can also be displayed horizontally rather than vertically as in Figure 5. We find that this display does not highlight the usage of memory as effectively as the vertical display. However, the horizontal display makes some of the text easier to read and makes the line showing memory usage over time in the root node easier to interpret. It also can be used to make more effective use of landscape displays.

4. Assessment

Based on our experience, the display has been effective in providing an understanding of how memory is being used. At the same time, it has highlighted many of the common memory-related problems that arise in software development.

The basic display highlights two things. Color coding draws the programmer's attention to those classes that are actually using memory. The ownership relationship then shows what this memory is being used for. For example, in the various figures, it is easy to see that most of the memory space is being used to store 3D vectors (class *SolarVector*) and that this is

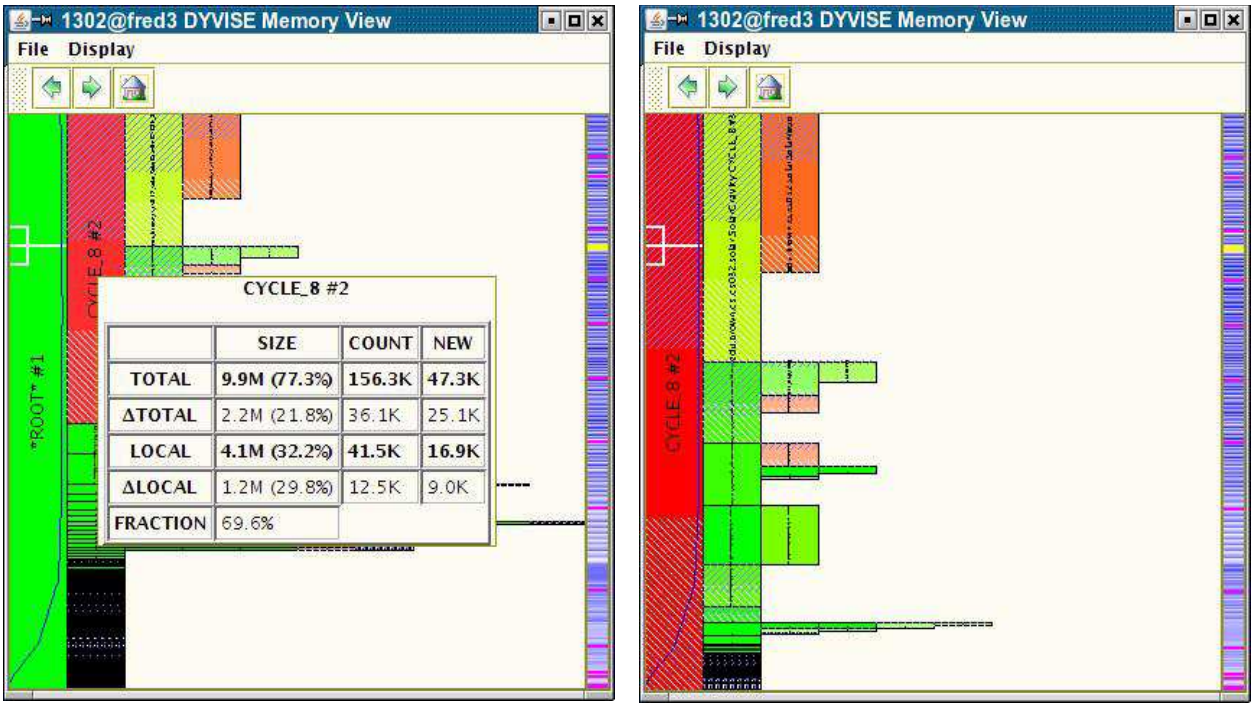

FIGURE 3. Different views of memory utilization.

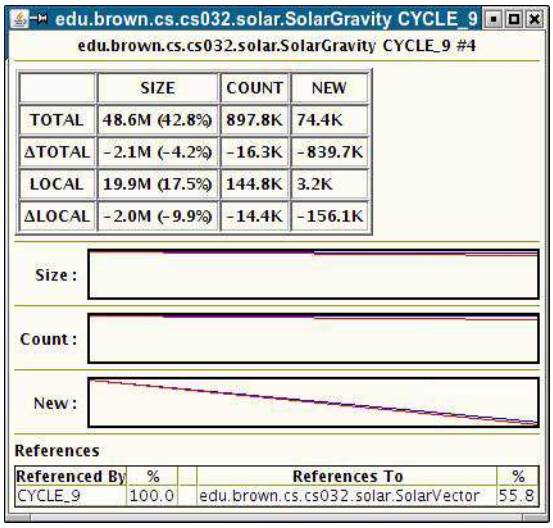

FIGURE 4. Data view for a single node

about evenly split between vectors contained in the objects themselves and vectors used for gravity computations. In more complex situations, such as a code search engine [24] whose memory display is shown in Figure 5, we were able to determine that most of the storage was used for storing Eclipse abstract syntax trees from the cycle node CYCLE_44 which contains the various syntax tree classes. In all the cases we have been able to check, the reported ownership relationship accurately reflects the programmer's intuition of memory allocation.

The tool has also proven itself useful in addressing memory-related problems. We have used it to identify memory leaks. These show up either as unexpectedly large uses of storage or classes where the storage requirements keep growing over time. For example, in the multi-body simulation of our examples, we at first noted a large chunk of memory that included XML-based classes. The root of these was the XML parser

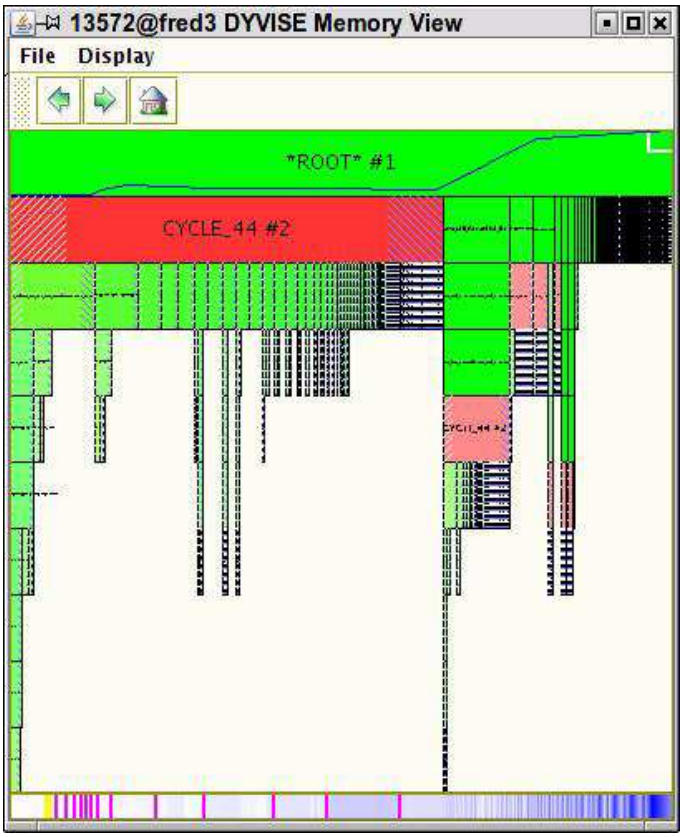

FIGURE 5. The alternative horizontal layout showing memory usage by a code search engine.

object which we keep and share. We did not realize that the parser kept a handle on the last XML that was read and thus was tying up a significant portion of memory. The space-used graph in the root object provides a clue to slower memory leaks; if it is continually increasing and yet the programmer doesn't expect that it should be, then it represents a potential leak.

In another example, our performance analyzer would run fine for a few hours and then would run out of memory. Using the memory display we were able to see that memory usage grew slowly at first and then suddenly jumped, with all the storage being put into arrays of integers owned by the history mechanism. This was unexpected and, by looking at the code, we found a simple bug was causing the system to continually allocate a new array rather than using the just allocated array.

The layout and display also address the other memory-related problems that we cited. Overhead involved in using multiple objects shows up as empty space to the right of an item. For example, clicking on the String class in the example of Figure 3, one quickly finds that Strings take up about 40% more than their underlying character arrays.

While the basic display does not show where objects are continually reallocated or the changes from one time to another, these are highlighted by cross-hatching. A class where most of the objects have been reallocated since the previous heap dump, will be covered in blue cross-hatching. Similarly, a class that has grown considerably in size since the last dump will have corresponding white cross-hatching. In both cases, the class will stand out and the programmer will note that there is a potential problem.

Finally, the time bar on the right of the display allows the programmer to correlate memory behavior with time. This can be done either through tool tips where moving the mouse over the time bar shows the actual clock time represented at that point. or by the programmer inserting markers into the display to remind themselves of particular program behaviors they want to investigate. The time view is also synchronized with the time displays provided by the underlying performance analysis tool.

5. Related Work

Memory analysis and visualization tools have been available for some time. The early analyses typically gave a picture of the heap without regarding the actual structure of object references [4,9,21,22]. More recent visualizations along these lines concentrate on multiple processes or garbage collection [4,19]. These

tools provided information about the types of storage used and who allocated them. *Heapview*, for example, showed a picture of the heap as the program run that could be color coded by data type, allocation site, allocation size, or age [22]. Later tools including *mprof* [27], Sun's *dbx*, *purify* [7], and, for Java, Sun's *hprof*, provide a post-mortem call site analysis of the heap, telling the user the number and source of allocations based on the call stack.

There have also been a number of tools that monitor summary memory utilization of Java programs as they run. These range from the tools distributed with Java today including *jconsole* [3] and *jvisualvm* [11], to Eclipse plug-ins such as *tptp* [5], to standalone open source or commercial systems such as *JMP* [15], Borland's *Optimizeit*, Quest's *JProbe*, or our *dyper* profiler [23]. Most of these tools display only simple visualizations, for example, graphs of total memory used and tables or lists of the space used by class.

A set of more sophisticated memory tools provide a view of the actual objects in the heap. *Jinsight* provides views of the objects and their references [16,17]. Sun's *jhat* provides a web front-end to let the user peruse object space based on a dump. IBM's *HeapAnalyzer* uses textual, tree-based viewers for the same purpose. Ultrarise's *UCrawl* provides a graphical front end for viewing data structures. *Fox* provides a query-based interface for browsing the heap [18].

Another set of tools takes the object graphs produced by the heaps and abstracts them to produce a more efficient display [8,13,14,20]. The main approach here looks for strongly connected components and dominators, effectively finding cycles and their entry points. [10] takes this a step further by deducing data structures and reporting memory usage by structure. This is done, however, in the complete object reference graph. Once it is done, the approaches collapse the graphs by looking for similar subtrees. Our approach tends to produce similar graphs, but we start with reference counts rather than the reference graph.

There have also been a variety of tools that specialize in helping identify memory leaks rather than providing an overview of the heap. Some of these use instrumentation to track writes and maintain data structure age [2]. Others modify the virtual machine to get a fast approximation of object age [1]. Others look at differences between collapsed object graphs [8]. Xu and Rountev look at containers and how they are used to assign probabilities to leaks [26]. Rayside and Mendel use a combination of instrumentation to get lifetimes and a better understanding of ownership based on control along with a collapsed object graph [20].

6. Future Work

We have used the system to look at memory usage in a wide variety of applications. As noted, we have used it to find memory leaks in our performance tool and our particle simulator, and to understand the memory behavior of our search tool. We have also used it to look at the storage utilization of a peer-to-peer network over time, a flow analyzer, a web crawler, and a number of student projects.

While we have found the system relatively easy to use and the visualizations easy to interpret, there are several improvements to the system that we will be working on.

We are continually improving the display front end. Here we would like to provide a separate window showing statistics, a way of selecting and highlighting a node in place to make the identification of duplicate trees simpler, providing an overview when a subtree is displayed showing its parents, a search mechanism to let the user quickly find a specific class, better cycle names, and different color mappings so that a wider range of colors are displayed.

We are also working on making the system easier to use on its own. Currently, the system is implemented as part of our performance monitoring package [23]. We want to provide a separate front end that makes it possible to do memory visualizations without the overhead of the remainder of the package. Also, we plan to integrate the separated tool into the Eclipse framework.

Extensions we are considering include keeping track of the age of objects and adding user-specified time markers. Age information for objects can be computed using the tags much as was done in Bell [1] and could provide additional help in identifying potential memory leaks. User time markers would make it easier for the programmer to relate program events with the memory display.

Our experience with the tool has shown that our approach can use simple memory dump information to produce graphical interactive views that are both accurate and useful for understanding the memory behavior of systems and for identifying potential memory-related problems.

The code for this system is available as part of the DYVISE package at:

ftp://ftp.cs.brown.edu/u/spr/dyvise.tar.gz

7. Acknowledgements

This work is supported by the National Science Foundation through grant CCR0613162.

8. References

1. Michael D. Bond and Kathryn S. McKinley, "Bell: bit-encoding online memory leak detection," *ASPLOS'06*, pp. 61-72 (2006).

2. Trishul M. Chilimbi and Matthias Hauswirth, "Low-overhead memory leak detection using adaptive statistical profiling," *ASPLOS '04*, pp. 156-164 (October 2004).

3. Mancy Chung, "Using JConsole to monitor applications," *http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html*, ().

4. Robert F. Erbacher, "Visual debugging of data and operations for concurrent programs," *Proc. SPIE Conf. on Visual Data Explortation and Analysis* Vol. **3017** pp. 120-128 (1997).

5. The Eclipse Foundation, "Eclipse test and performance tools platform project," *http://www.eclipse.org/tptp/index.php*, (August 2007).

6. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, "gprof: A call graph execution profiler," *SIGPLAN Notices* Vol. **17**(6) pp. 120-126 (June 1982).

7. Reed Hastings and Bob Joyce, "Purify: fast detection of memory leaks and access errors," *Proc. Winter Usenix Conf*, (January 1992).

8. Maria Jump and Kathryn S. McKinley, "Cork: dynamic memory leak detection for garbage-collected languages," *POPL'07*, pp. 31-38 (2007).

9. Babak Mahdavi and Karel Driesen, "Heap hot spot visualization in Java," *McGill U. CS Tech report SOCS-01.8*, (May 2001).

10. Nick Michell, Edith Schonberg, and Gary Sevitsky, "Making sense of large heaps," *ECOOP 2009*, (July 2009).

11. Sun Microsystems, "VisualVM 1.1.1," *https://visualvm.dev.java.net/*, ().

12. Sun Microsystems, "JVM Tool Interface," *http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html*, (2004).

13. Nick Mitchell, "The runtime structure of object ownership," *European Conference on Object-Oriented Computing* (*ECOOP*), pp. 74-98 (2006).

14. Nick Mitchell and Gary Sevitsky, "The causes of bloat, the limits of health," *Proc. OOPSLA 2007*, pp. 245-260 (2007).

15. Robert Olofsson, "Java memory profiler user guide," *http://www.khelekore.org/jmp/docs/jmp_users_guide.pdf*, (2006).

16. Wim De Pauw, Doug Kimelman, and John Vlissides, "Visualizing object- oriented software execution," pp. 329-346 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine A. Price,MIT Press (1998).

17. Wim De Pauw and Gary Sevitsky, "Visualizing reference patterns for solving memory leaks in Java," in *Proceedings of the ECOOP '99 European Conference on Object-oriented Programming*, (1999).

18. Alex Potanin, James Noble, and Robert Biddle, "Snapshot query-based debugging," *Proc. 2004 Australian Software Engineering Conference*, pp. 251-259 (2004).

19. Tony Printezis and Richard Jones, "GCspy: an adaptable heap visualization framework," *17th OOPSLA*, pp. 343-358 (2002).

20. Derek Rayside and Lucy Mendel, "Object ownership profiling: a technique for finding and fixing memory leaks," *Proc. ASE '07*, pp. 194-203 (November 2007).

21. Steven P. Reiss, *FIELD*: *A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).

22. Steven P. Reiss, "Visualization for software engineering -- programming environments," in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine Price,MIT Press (1997).

23. Steven P. Reiss, "Controlled dynamic performance analysis," *Proc. 2nd Intl. Workshop on Software and Performance*, (June 2008).

24. Steven P. Reiss, "Semantics-based code search," *ICSE 2009*, (May 2009).

25. Ben Schneiderman, "Tree visualization with tree-maps: a 2-D space-filling approach," *ACM Transactions on Graphics* Vol. **11**(1) pp. 92-99 (January 1992).

26. Guoquing Xu and Atanas Rountev, *ICSE '08*, pp. 151-160 (May 2008).

27. Benjamin Zorn and Paul Hilfinger, "A memory allocation profiler for C and Lisp programs," *Proc. Summer 1988 USENIX Conference*, pp. 223-237 (1998).