Visualizing the Java Heap

Demonstration Proposal

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. We propose to demonstrate the tool in action, showing how it works dynamically on running processes and how it is designed to address a variety of specific memory issues.

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 usage 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. 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. Other problems involve the inefficient use of memory, for example the overhead involved in using multiple levels of objects. A third set of problems involves churn where large numbers of objects are continually created. Another class of problems involves unexpected changes in memory size. All these problems need to be correlated with program behavior.

The most useful memory visualizations to date show memory ownership [4]. An object owns another if it is responsible for the storage of the other. A memory ownership graph tells the user what classes are actually responsible for allocated memory and not just where memory is being used. Several tools have produced off-line ownership views, typically analyzing a complete heap dump, collapsing similar nodes, and then displaying the result as a graph [2,3,5]. Jhat and Eclipse's TPTP show the raw access information textually.

Our visualization is designed to display an annotated memory ownership graph that lets the programmer both understand overall memory behavior and address these and other memory-related problems, and to do it all while the program is running.

2. Tool Overview

Most current tools for Java memory visualization that go beyond simple object counts depend on producing a dump of the current state of memory into a file and then processing that file off-line. This operation can be quite expensive both in terms of run time and the amount of disk space required to store the result.

Our tool uses Java's JVMTI facilities to collect minimal information about the heap. In particular, we collect the number and size of objects for each class and counts of the number of times an object of one class references an object of a second class. The result is that our tool uses less overhead, the size of the data is significantly smaller, the data can be collected on the fly, and the data doesn't have to be stored on disk. This lets us provide on-line memory analysis where the user can see what is happening as the program runs.

In order to reconstruct a memory model from the limited information we collect, we use a variety of techniques. We split collection classes by their referring type, yielding separate lists, sets, maps, etc. for different program uses. We also create pseudo classes to represent large arrays and strings. Then we convert the reference count information into a directed acyclic graph using an algorithm similar to *gprof* [1], adding new nodes to represent cycles which effectively represent linked data structures. Next we allocate ownership. Where items have multiple incoming links we use statistical information from multiple heap dumps to do a least squares approximation using quadratic programming to provide an accurate allocation.

The result is a relatively compact graph that represents heap utilization in terms of ownership. This graph is still too large and often too complex to be displayed

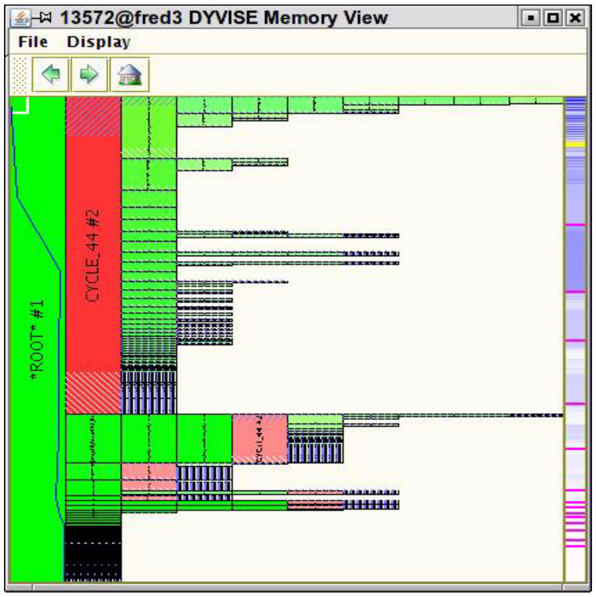

FIGURE 1. Sample Memory Display

directly. Based on our previous work on visualizing *gprof* performance information [6,7], we display the result as shown in Figure 1.

In this display we have converted the directed acyclic ownership graph into a tree. This is done by duplicating child nodes where they have more than one parent. For example, the figure shows multiple instances of CYCLE_44 (the red nodes) and its successors. Where we duplicate nodes, we allocate a fraction of the subtree's size to each of the parents.

Each block in the display corresponds to a class or a cycle. The graph is displayed as a tree with the root at the left and the children of a node to its right. The vertical space occupied by a node corresponds to the total space used by that class and all the memory that it owns. Nodes are color coded from green to red to show which classes are actually using the most memory themselves, ignoring what they refer to. Color saturation is used to show where nodes have been duplicated. The children of a node are sorted so that the larger children are above the smaller ones. A gap to the right of a node indicates the storage used just by this node. The display is restricted so that only classes which account for more that 0.1% of memory space are displayed.

Additional information is included in the nodes. The blue hashing at the top of a node shows the number of new objects of this class that were created since the previous heap dump. The white hashing at the bottom of a node indicates how much the space used by this class has increased since the previous dump. The graph shown in the root node shows the history of memory use by the class over thee courses of the run. This information can identify churn and point to leaks.

The display is also interactive. As the user moves the mouse around within it, tool tips provide detailed information about each class including the number of objects of that class, the amount of local memory used, and the amount of memory used by this node and its children. Users can click on any node to concentrate on the tree rooted by that class, or can request a data window showing additional information about the memory used by the class.

Finally, the bar on the right of the display shows the history of memory usage over the course of the run as reported by the Java management facilities. Here dark blue indicates more memory being used while light colors indicate less. A typical light-dark pattern shows the garbage collector in action. The magenta lines in the bar show when heap dumps were taken. The yellow line shows the current time the user is looking at. The user can use this bar to navigate in time over the heap utilization in the run.

3. Demonstration Overview

We propose to demonstrate the tool by showing it dynamically analyzing the memory of several different software systems. We will show how memory utilization can be quickly explored and understood and how several of the memory-specific problems we cited can be directly addressed through the visualization.

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

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

4. Acknowledgements

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

5. References

1. 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).

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

3. Nick Mitchell and Gary Sevitsky, "LeakBot: an automated and lightweight tool for diagnosing memory leaks in large Java applications," *ECOOP 2003*, pp. 351-377 (2003).

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

5. 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).

6. Steven P. Reiss, "The Desert environment," *ACM TOSEM* Vol. **8**(4) pp. 297-342 (October 1999).

7. Steven P. Reiss, "Bee/Hive: a software visualization backend," *IEEE Workshop on Software Visualization*, (May 2001).