The Runtime Structure of Object Ownership

Nick Mitchell

IBM TJ Watson Research Center
19 Skyline Drive, Hawthorne NY 10532
nickm@us.ibm.com

Abstract. Object-oriented programs often require large heaps to run properly or meet performance goals. They use high-overhead collections, bulky data models, and large caches. Discovering this is quite challenging. Manual browsing and flat summaries do not scale to complex graphs with 20 million objects. Context is crucial to understanding responsibility and inefficient object connectivity.

We summarize memory footprint with help from the dominator relation. Each dominator tree captures unique ownership. Edges between trees capture responsibility. We introduce a set of *ownership structures*, and quantify their abundance. We aggregate these structures, and use thresholds to identify important aggregates. We introduce the *ownership graph* to summarize responsibility, and *backbone equivalence* to aggregate patterns within trees. Our implementation quickly generates concise summaries. In two minutes, it generates a 14-node ownership graph from 29 million objects. Backbone equivalence identifies a handful of patterns that account for 80% of a tree's footprint.

1 Introduction

In this paper, we consider the problem excessive memory footprint in object-oriented programs: for certain intervals of time, the live objects exceed available or desired memory bounds. Excessive memory footprint has many root causes. Some data structures impose a high per-element overhead, such as hash sets with explicit chaining, or tree maps. Data models often include duplicate or unnecessary fields, or extend modeling frameworks with a high base-class memory cost. There may be objects that, while no longer needed, remain live [34, 39], such as when the memory for an Eclipse [17] plugin persists beyond its last use. Often, to mask unresolved performance problems, applications aggressively cache data (using inefficient data structures and bulky data models).

To isolate the root causes for large object graph size requires understanding both responsibility and internal content: the program may hold on to objects longer than expected, or may use data structures built up in inefficient ways. We analyze this combination of *ownership structures* by summarizing the state of the heap — at any moment in time within the interval of excessive footprint. In contrast, techniques such as heap [34, 35, 40, 43], space [36], shape [32], lexical [6], or cost-center [37] profiling collect aggregate summaries of allocation sites. Profiling dramatically slows down the program, gives no information about responsibility

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 74–98, 2006.

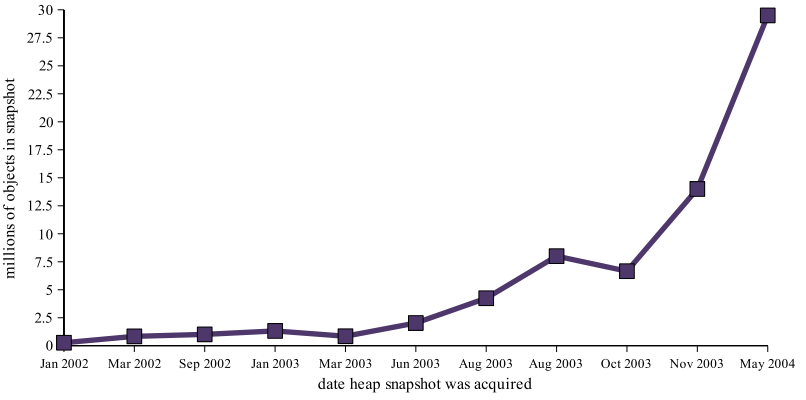

Fig. 1. Growth in the size of Java heaps in recent years

Table 1. A commonly used, but not especially useful, graph summary: aggregate objects by data type, and then apply a threshold to show only the top few

type	objects	bytes
`primitive arrays`	3,657,979	223,858,288
`java/lang/String`	2,500,389	80,012,448
`java/util/HashMap$Entry`	2,307,577	73,842,464
`java/util/HashMap$Entry[]`	220,683	57,726,696
`customer data type`	338,601	48,758,544
`java/lang/Object[]`	506,735	24,721,536

or internal content, and conflates the problem of excessive temporary creation with the problem of excessive memory footprint.

The task of summarizing the state of the heap [12, 20, 9, 29, 19, 32] at any moment in time [3, 31, 18, 25] is one of graph summarization. In this case, the graph's nodes are objects, and the edges are references between them. Summarizing the responsibility and internal content of these graphs is, from our experience with dozens of large-scale object-oriented programs, quite challenging. In part, this is because these object graphs are very large. In Figure 1, we show typical object graph sizes from a variety of large-scale applications. Over the years, this figure shows that the problem has grown worse. Contemporary server applications commonly have tens of millions of live objects at any moment in time.

Furthermore, the way objects are linked together defeats easy summarization. A good summary would identify a small number of features that account for much of the graph's size. Achieving this 80/20 point, especially for large, complex graphs, is challenging. Many commercial memory analysis tools [3, 31, 18] *aggregate* by data type, and then chooses a *threshold* to show those biggest types. This technique produces a table such as Table 1. Typically, generic data types float to the top. Even for the customer-specific types, the table gives us no sense of who is responsible, or how the instances are structured; e.g. are the instance of these types part of a single large collection, or several smaller ones? These same