Autonomous Garbage Collection: Resolve Memory Leaks In Long Running Server Applications

Brian Willard
willabr@mail.northgrum.com
Electronics and
Systems Integration Division
Northrop Grumman
Melbourne, FL 32902

Ophir Frieder
ophir@csam.iit.edu
Department of Computer Science
and Applied Mathematics
Illinois Institute of Technology
Chicago, IL 60616

Abstract

We demonstrate the benefits of a garbage collection technique that requires neither programmer assistance or rebuilding (compiling or linking) of target applications. Thus, it effectively mitigates performance degeneration due to memory leaks in applications when source code and object code is not available. Our technique is an extension of the garbage collection method known as conservative garbage collection. We refer to this as autonomous garbage collection. Autonomous garbage collection is especially useful for long running server applications in large-scale information processing server environments. Our prototype demonstrates that this garbage collection technique is feasible. Our experimental results show that this technique is more general and easier to use than many of the previous garbage collection proposals targeted at resolving memory leaks in non-cooperative server applications.

07/26/98

Index Terms

dynamic memory allocation, garbage collection, heap storage, information retrieval, memory leaks, object-oriented languages, software performance degeneration

1. Introduction

Memory management defects related to dynamic storage allocation account for some of the most problematic and complex defects in existence today. It is not uncommon for long running information server applications to be plagued with memory leaks. This is especially true for large-scale information processing systems, developed with a programming language that employs dynamic memory allocation under the paradigm that places the responsibility on programmers to explicitly deallocate dynamically allocated storage after it is no longer in use by the program.

Memory leaks are especially problematic in large-scale information processing systems where long running servers are involved. In such systems it is not uncommon for software not totally devoid of memory leak defects to be fielded into operational configuration. Memory leaks accrue over time leading to degeneration of system performance and ultimately system failure.

Information retrieval systems are a prime example of the type of software intensive systems in which memory leaks cause significant havoc. In this digital information age, massive volumes of electronic information are interconnected on the Internet, and steady advancements in information retrieval technology continue to improve the exploitation this massive base of information. The intense, concentrated I/O processing characteristics of these large-scale information retrieval systems has notably impacted traditional management of dynamic storage. The symptoms associated with

weaknesses in traditional dynamic storage management often become much more visible in such systems, causing serious degradation to system performance.

The demands on the memory systems within large-scale information processing systems are massive, and, as information becomes increasingly available in electronic form, these I/O demands continue to escalate. Because of massive I/O demands, we recognize that servers in large scale information processing systems cannot long endure even small memory leaks. We also recognize that a large base of such systems exist and have been fielded, implemented in C/C++. The C/C++ language requires the programmer to manage the dynamic memory allocation/deallocation. Furthermore, in these systems, not unlike other types of software intensive systems developed in C/C++, some level of memory leaks inevitably go undetected and are fielded with the system, despite the application of mature software development methodology by disciplined and highly skilled C/C++ programmers and system test engineers. Thus, we focused on developing a capability to resolve memory leak defects without programmer assistance, requiring neither source code nor object code.

Garbage collectors are effective because they are capable of bounding the amount of storage lost due to memory leaks. As long as the amount of lost storage is bounded, memory and its associated page files can be sized to accommodate the usage of the application. Thus, our research has targeted a solution that bounds memory storage loss due to memory leaks. In addition, we targeted a solution that addresses such problems in fielded systems. Traditionally, all garbage collectors, including conservative garbage collectors, require some level of programmer assistance. We describe an autonomous garbage collection method that meets the research objective: detect and resolve memory

leaks without programmer, compiler or linker assistance. We demonstrate successful operation of the algorithm and illustrate its effective performance. The algorithm performs well in soft real-time execution environments, similar to the execution paradigm in information retrieval systems. In information retrieval systems, it is important, from a usability perspective, to maintain rapid query response time the majority of the time. For example, query response time should be less than 2 seconds 95% of the time. However, in an information retrieval system, if the target query response time, for example less than 2 seconds, is missed, then no catastrophic failure is incurred as is the case in hard real-time systems.

2. Overview Of The Garbage Collector Architecture And Operation

Before describing the internal workings of the autonomous garbage collection algorithm in Section 3, this section describes the architectural framework in which the algorithm is implemented. The garbage collector is implemented in C and executes on DEC's OpenVMS operating system.

2.1 Acquiring Essential Information

To accomplish autonomous garbage collection, the following information must be automatically acquired from within the target application:

- location of heap storage
- location (starting address and size) of all allocated heap segments
- location of the program stacks
- location of writable program memory

With this information a conservative garbage collection [boeh88] technique can be employed. When the garbage collector interrupts the target application, the contents of the registers in use by the application code are saved on the program stack. Thus, the garbage collector knows that all active pointers are either on the program stack or in writable program memory. The garbage collector can then conservatively identify all active pointers by evaluating each word in the program stack and writable program memory and determining whether it has the value of an address within the allocated heap segments. After all words in the program stack and writable program memory are conservatively evaluated, it is safe to declared as garbage and reclaimed allocated heap segments for which not one reference was found.

Granted, with a conservative collection approach it is possible for a value to be found that equates to an address of an allocated heap segment, but the word semantically not be a pointer. Such cases are referred to as pointer misidentification. Pointer misidentification is safe as it merely results in false retention of an inactive heap segment. In practice pointer misidentification has a low rate of occurrence.

2.2 VMS Terms

Definitions of several VMS specific terms will be helpful in the explanation of the garbage collection architecture developed in this research project; namely *image*, *executable image, shareable image* and *main image*. These terms refer to program execute concepts that are common to most modern operating system paradigms. The terms are defined in the table below.

Term	Definition
image	an **image** is a file, containing binary code and data, that can be executed by the VMS operating system
executable image	an **executable image** has a unique entry point called a transfer address and can be activated at the command line by issuing the RUN command.
shareable image	a **shareable image** is a collection of procedures that can be called by code in an executable image or other shareable images, and is equivalent to the term dynamic link libraries in other OS environments.
main image	the term **main image** refers to an executable image invoked by a user via the RUN command. Activation of a main image results in activation of any shareable images with which it was linked and in turn any with which they were linked.

Table 4.3.1 Definition of VMS terms

2.3 Obtaining Heap Allocation Information

The implementation chosen is a simplification of, and serves as proof of concept for, the more robust, preferred embodiment of the approach. The preferred embodiment

is realized by creating a shareable image that dynamically (via dynamic binding) intercepts all run-time library invocations, including specifically the dynamic memory management services. This shareable image need not implement the services, but rather only intercept the invocation of such services in order to obtain heap allocation addresses and forward the calls to the stock operating system run-time library routines to perform the actual dynamic memory operation requested. In addition, this shareable image would perform the responsibilities of the gcKernel in our architecture. Figure 2.1 shows this concept.

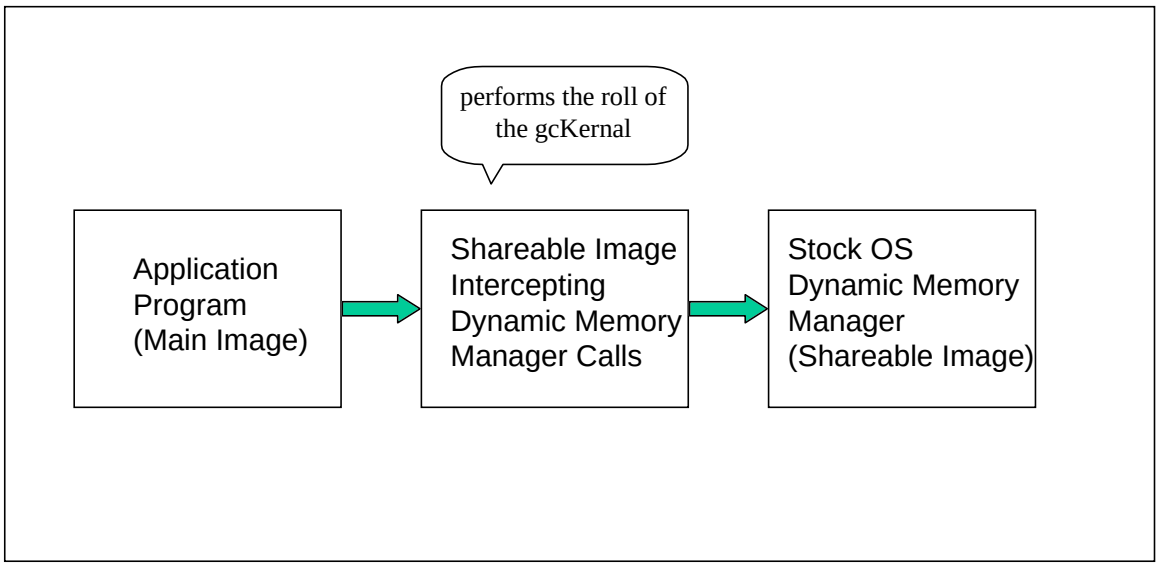

Figure 2.1 Preferred Embodiment of gcKernal

In OpenVMS, the OS provides one run-time library of general purpose routines (including dynamic memory management services) for the entire system that is compatible with, yet separate from, all the language compilers. Thus, the insertion of one shareable image (i.e., dynamic link library), as shown in Figure 2.1, between the

application programs and the run-time library works for programs developed in all languages.

Without incurring the large implementation task of emulating all the interfaces in the general purpose `LIB$RTL` OpenVMS library that would be required to actually accomplish insertion of a shareable image between the application programs and the run-time library as described, the feasibility of the concept was demonstrated in our implementation as follows. The garbage collector (gc) obtains heap allocation information by replacing, and intercepting calls to, DEC's Heap Analyzer[1] shareable image. The memory allocation and deallocation procedures in the OpenVMS run time library (`LIB$RTL`) are instrumented to provide statistics on memory allocation events to the Heap Analyzer. Thus, by replacing the Heap Analyzer with a shareable image (called gcKernel), heap allocation information is easily obtained on any executable image.

Figure 2.2 shows the image interfaces that occur when a main image, written in C++, allocates memory using `new` or `malloc`. The memory allocation service `new` resides in the `DECC$SHR` shareable image, and `new` in turn call