Avoiding Megaword Memory Leaks On a Cray T90

David J. Gigrich
Structural Analysis Computing
The Boeing Company
david.j.gigrich@boeing.com

Abstract

This paper will address the inherent problems with the use of dynamic memory allocation on a Cray T90. It will illustrate how multi-million word blocks of memory can become lost to the system as the analyst's memory management scheme, using allocatable arrays, competes against both the compiler and system routines for valuable heap-space. Memory leak problems of this nature can occur regardless of language type (FORTRAN 90, FORTRAN 77, or even C) and are not solely restricted to Cray products.

The type of memory leak discussed in this paper can be very subtle, often going unnoticed for years. Slight variances in job size or minor changes in the program's load can cause this problem to suddenly surface. Large increases in a job's memory requirements may result in prolonged execution time or even job failure. The solutions used to avoid this problem will be discussed in detail.

Table of Contents

1. Introduction -- Memory Management

Historically the performance of the entire computer system has been directly dependent on how much memory is available during job executions. Computer systems run applications with vastly differing amounts of memory, from personal computers with mere kilobytes of memory to Cray supercomputers with gigabytes of memory. However, they all have one thing in common, computer memory is a resource used by the system that needs to be managed.

The computer operating system is not the sole manager of computer memory (heap-space), nor has it been for quite some time. Sophisticated computer applications, along with today's compilers, also attempt to manage computer memory to varying degrees to improve job performance and overall job throughput. While most computer applications require little or no internal memory management, there are many applications that can not function without some formal memory management scheme of their own.

Memory management schemes can differ greatly from one computer application to the next. The four main schemes for non-virtual (fixed memory) machines are Single User Contiguous, Fixed Partitions, Dynamic Partitions, and Re-allocatable Dynamic Partitions. The very nature of an application not only dictates which method will work best, but also how it is implemented. Dynamic Partition schemes have proven to be most cost effective methods on the Cray T90 and will be the ones discussed throughout this paper. As computer-simulated models (engines, cars, planes, etc.) become more detailed, the amount of memory required can grow almost exponentially. Refined model definition continues to necessitate the need for increased computer memory and/or better memory management techniques for these applications. Yet conflicts arise when an application's own internal memory management scheme inadvertently competes against the compiler and/or library routines for valuable computer memory space.

Computer memory management conflicts can lead to fragmentation of available memory on the T90 in these types of applications. The fragmentation occurs as interspersed areas of computer memory are de-allocated (freed) during job execution. Thus, resulting in total available memory for the application no longer being contiguous. Typically, this fragmentation goes unnoticed, because the unusable portions of memory are relatively small. However, as problem size increases, an application's available memory can become more fragmented. Without sufficiently large segments of contiguous computer memory, applications may perform very inefficiently, or they may be unable to allocate the space required for continued execution. In this paper, the term "memory leak" will be used to refer to the portions of memory that have become so fragmented (even though still available) they are no longer of any practical use to the application. Computer memory lost in this fashion can not be released back to the operating system until the application it has been assigned to has finished executing. Therefore, it is also not available to any other jobs executing on the computer system.

2. Memory Leaks

There is virtually no way to control every occurrence of this specific type of memory leak during the execution of an application. Compiler and operating system writers have long been well aware of the problem. Nevertheless, it has not been a major concern, because of the large amounts of computer memory available on today's high-performance machines. In addition, the over-management of computer memory can severely degrade an application's performance. Therefore, losing access to a few words of memory or even to several thousands of words of memory is generally not an issue.

2.1 Job Failure

Problems arise when memory becomes fragmented (noncontiguous) in such a way that there are several large unallocated segments (100,000 words or more) of memory available during job execution. Since this memory has already been assigned to the application, as part of the heap area on the Cray T90, there is less room for continued growth in the heap. When the application requests a block of memory greater than the size of each individual free segment, even though still less than the total memory available, the allocation will fail. Once an allocation for memory space is denied, many applications can not complete execution successfully and are forced to terminate.

2.2 Problem Realization

The failure of a job due to insufficient memory does not necessarily mean that the application has a serious memory leak. Typically the problem size has grown such that a step increase in the job's memory limit is all that is required. Changes to the compiler, library routines, or modifications to the program all directly contribute to the memory space requirements of the application. However, if previously executed jobs now require a significantly larger amount of memory than before, there may well be a very serious problem.

Between 1994 and 1999, computer memory requirements for one application had increased steadily from 15MW (megawords) to 18MW. However, having a Cray T90 with 512MW of memory, the problem was considered not yet worth pursing. Following the November 1999 release of this application most jobs still ran successfully under 18MW, yet some required 40MW and two appeared to require over 90MW. The wall-clock times (turnaround) for these types of jobs were 3 hours, 5 hours, and 36 hours respectively. There was little change in actual CPU time, but competition for memory resources had severely degraded not only this application's individual job performance, but also the throughput of the entire Cray T90 system. The problem had now reached a level that made it imperative to have it resolved quickly.

2.3 Identification

The application had been converted from FORTRAN 77 to FORTRAN 90 in 1998 and made extensive use of allocatable arrays. As the first step toward identifying the sudden increase in the application's memory requirements, it was verified that there were no significant changes in the application since its previous release six months earlier. To determine exactly where the allocations failed, the 40MW and 90MW jobs were executed with hard limits of 18MW. This resulted in memory allocation failures in two different routines. Analysis of the routines revealed that the allocations were both for a complex two-dimensional array (2100x2100), each requiring over 8.8 million words of available memory space. Further analysis of the application uncovered a third routine with two large allocations, one of 8.8MW and another of nearly 5MW. It also verified that all other memory allocations were relatively small (well below 1MW), and that each routine did in fact release its allocated memory space prior to exiting. With program length (excluding the heap area) at approximately 1 million words and the allocated arrays being freed appropriately, the memory leak had been confirmed.

3. Understanding the Nature of the Problem

It is vital that the code developer possesses an understanding of what takes place at the system level as a particular application is being executed. Many simple FORTRAN or C language statements that are taken for granted require memory to be allocated in order for certain operations to be performed. Likewise, other statements result in some of the allocated memory from these operations being de-allocated and made available for later use in the program. During program execution, memory allocation can be controlled in several ways; the Operating System (O/S), code generated by the compiler, utility routines, and the application itself.

3.1 Operating System Memory Management

Operating system memory management involves the use of memory management hardware/software to manage the resources of the storage hierarchy and the allocation of these resources to the various activities running on the system. It also deals with protection and security, which helps to maintain the integrity of the operating system against deliberate or accidental damage. The operating system, during program execution, requires memory space for the application, program file allocations, execution of system commands, and so forth.

3.2 Compiler

The code generated by the compiler will often allocate memory space from the heap for a variety of reasons at execution time. Subroutine or function calls result in local variables being placed on or taken off the stack (dynamic variables) as the tree of routines is traversed. In addition, array operations (language syntax) or the passing of partial arrays via an argument list will result in temporary arrays being placed on the stack.

3.3 Utility Routines

A good application programmer makes use of existing routines whenever possible. These utility routines can be a result of the functionality built into the language, system routines, data center routines, or even part of the organization’s own internal library. The possibility that some routines take advantage of dynamic memory (heap space), via allocatable arrays or pointers must be considered.

3.4 The Application

The very design structure of the application can require the utilization of dynamic memory to solve its problems. This is often required for out-of-core solvers, because the amount of memory available to the application may be insufficient. Either the machine does not possess enough memory, or it is a shared resource making it impractical for any single application to tie up most or all of the machine's memory for extended periods of time.

The problem encountered with this particular application was that on rare occasions, after the first 8.8MW of contiguous memory space was de-allocated, small portions of that memory segment were assigned for other purposes. Consequently, the remaining available memory in the segment became insufficient for the next 8.8MW request and a totally new 8.8MW segment would need to be assigned from the heap. Since there were three routines, which utilized an array of this magnitude, a third totally new 8.8MW memory segment might also be required. To further compound the problem, calls for these three routines resided within a loop structure of the main routine.

4. Solutions

Realization that the freed 8.8MW memory segment had been partially re-allocated only for some jobs suggested that the sub-division of this large segment could indeed be prevented. Considerable thought was given to three courses of action: do nothing, perform a simple (quick) fix, and redesign the software.

4.1 Do Nothing - Leave as is

Approximately 60% of the jobs executing the application did not appear to be affected by the sudden increase in memory. However, the majority of users still had several jobs switching back and forth between 18 and 40MW of memory required. These users found themselves unable to select the proper job memory size before submitting their jobs. Typically, the larger a job's memory limits, the greater the wall-clock time before the job begins and finishes execution. In addition, the 90MW jobs were severely degrading the entire Cray system. These factors dictated that some type of correction would be necessary as soon as possible.

4.2 Simple Fix

The simple (quick) fix to the problem was to force the large dynamic array to have its memory space assigned at or near the end of the heap area. This usually ensured that small memory requests could be satisfied by allocating space from the beginning of the heap area, without impacting the space required for the large array. To accomplish this, 2 million words of memory were requested prior to the allocation of the large dynamic array, then freed once the large allocation occurred. This scheme required less than a day's effort and enabled over 90% of the jobs to run successfully under 18MW. Still, nearly 10% of the jobs were not helped by this scheme, because there was no sure method to determine the amount of memory space that had to be available in the heap, excluding the memory space required for the large array.

4.3 Software Redesign (Comprehensive Solution)

To ensure that this type of memory leak problem would **not** continue to plague the Cray system, a comprehensive solution was required. The software was redesigned in such a way that only a single large allocatable array (matrix) would be required, rather than three. This was accomplished by assigning the array's large memory space at the highest level and passing its address throughout the application. This provided three benefits, the matrix in question only needs to be created once, application I/O was reduced, and it eliminate the possibility of this specific type of a memory leak. On the downside, a large segment of memory is locked up for the entire execution of the job.

5. Results

Job performance is measured in a variety of ways; CPU time, memory usage, wall-clock (turn-around) time, and total dollar cost. Job turn-around time is usually the one of most concern to the user community. However, CPU time and memory usage are always important factors in trying to improve an application's wall-clock time and reduce its total execution (dollar) cost. The following tables illustrate two different jobs with and without the corrections to the application to resolve the memory leak:

Job 1

Software	Wall-Clock	CPU Time	Memory (MW)	Dollars ($$$)
Original	10.8 hours	2680 seconds	46.6	$246
Simple Fix	3.1 hours	2924 seconds	17.5	$174
Redesigned	2.4 hours	2949 seconds	13.3	$163

Job 2

Software	Wall-Clock	CPU Time	Memory (MW)	Dollars ($$$)
Original	36.2 hours	2797 seconds	90.5	$379
Simple Fix	3.3 hours	3045 seconds	17.8	$180
Redesigned	2.5 hours	3087 seconds	14.5	$173

6. Conclusions

Better utilization of available computer memory is always preferred to purchasing more. Operating systems and compilers are complex enough without vendors attempting to account for every application's memory management scheme. Therefore, it becomes the responsibility of the software developers to ensure that application memory management schemes do not conflict with those of the system, compiler, or utility routines.

The smarter the compilers and operating systems become in today's sophisticated computing environments, the more difficult our jobs become. Hence, as software developers strive harder to reduce an application's memory requirements, by continually releasing memory back to the system, the more fragmented available memory actually becomes. At a certain point available memory fragments can get so small that they are essentially useless to the system. This is especially true of applications developed long ago, when computer memory was a precious commodity.