A Dynamic Memory Measuring Tool for C++ Programs

Woo Hyong Lee, J. Morris Chang and Yusuf Hasan
Illinois Institute of Technology, Dept. of Computer Science
Chicago IL USA 60616
E-mail: {leewoo, chang, hasan}@csl.iit.edu

Abstract

Dynamic memory management has been a high cost component in many software systems. A study has shown that memory intensive C programs can consume up to 30% of the program run time in memory allocation and liberation. Especially, in C++ programs, they tend to have object creation and deletion prolifically. C++ memory allocation rate can be as much as ten times higher than the comparable applications written in C. Despite the importance of dynamic memory management in C++, there exist few software tools to study dynamic memory in C++. This paper introduces a tracing tool, called mtrace++, to study the dynamic memory allocation behavior in C++ programs. Mtrace++ is a source code level instrumented tracing tool which produces records of allocation and deallocation information. Mtrace++ identifies originations of allocated memories and life-spans of objects. With limited overheads, mtrace++ can help programmers to solve dynamic memory problems with affordable cost.

1. Introduction

Dynamic memory management (DMM) has been a high cost component in many software systems [1]. A study has shown that memory intensive C programs can consume up to 30% of the program run time in memory allocation and liberation [2, 3]. The object-oriented programming language (OOPL) systems tend to have object creation and deletion prolifically. The empirical data shows that C++ programs perform an order of magnitude more allocation than comparable C programs [4, 5]. In most cases, many small objects are prolifically created and deleted. The causes for that behavior are not clear. No data has been reported regarding the memory allocation pattern in C++ applications at this point. To provide useful information for C++'s dynamic memory management, this paper presents a C++ memory tracing tool (i.e., *mtrace++*) which helps programmers to investigate the behavior of their programs.

Mtrace++ is a software tool which allows to investigate C++ memory allocation behavior at source code level. It provides information of object distribution and allocation paths. The object distribution includes an object's size, frequency and origination which are useful information for programmers. For example, programmers offer a special strategy for highly referenced objects. Programmers can easily understand all indirect callers of allocated objects with the allocation path table. Eventually, with the help of the allocation paths, programmers can easily and correctly maintain objects. Based on the knowledge of overall allocation behavior, new memory management strategy can be developed (e.g., a special version of segregated fit).

We start with a classification of all the possible situations that may invoke the dynamic memory management (DMM) in C++. C++ code is naturally organized by class, common tracing tool can only trace in non-object manner. C++ has it's unique style of memory invocations. Therefore, we need to classify all the possible situations that may invoke the dynamic memory. In our hypotheses, these memory allocation patterns are related to either application or C++ language. For example, DMM invocations from constructors, copy constructors, or assignment operator= overloading are unique to object-oriented programming with C++. Application specific member functions which invoke *new* or *delete* operator explicitly are application related.

Mtrace++ is a very useful tool for programmers to understand the execution behavior of C++ programs. It utilizes the technique of source code instrumentation. C++ source code is modified by inserting statements which contain *beginning* and *ending* marks for each dynamic allocation scenario. *Mtrace++* is a two phase tool: execution phase and analyzing phase. During the execution phase, it modifies original source code and links with instrumented objects into an executing application and collects data as the application executes. During the analyzing phase, it creates tables which contain concise allocation information. After it finishes the analyzing phase, it produces another format of data file which illustrates the allocation tree and the depth of the path.

The remainder of this paper is organized as follows. Section 2 summarizes the design of *mtrace++*. Section 3 details the measurements of *mtrace++*. The last section presents the conclusions of this paper.

2. Design of *Mtrace++*

Mtrace++ is a tool to understand the dynamic memory invocation behavior of C++. It collects information of an object's size, frequency, allocated path and depth of the path. It adapted the technique of source code instrumentation which is used to trace the dynamic memory allocation. It takes C++ source code as input and inserts C++ statements, as marks, into our hypothesized allocation cases. The following sub-section details the allocation cases which are: constructor, copy constructor, overloading assignment, operator=, and type conversion.

Mtrace++ is coded in C++ and inserts *beginning* mark and *ending* mark for each dynamic memory allocation scenario. To get the concise data, *mtrace++* needs two phases. Firstly, source code is modified, compiled and linked into an executable binary file. After running this binary file, it creates data which contains the information of allocated memories. The second phase is the analyzing phase. During the analyzing phase, it takes the data collected by the first phase and presents it into summarized tables.

2.1. Cases of Dynamic Memory Allocation in C++

C++ has five distinctive allocation cases. The first four scenarios are unique to object-oriented programming in C++. They are: constructor, copy constructor, overloading assignment operator=, and type conversion. The fifth case is user-defined member functions that invoke new operators explicitly. In this case, it all depends on the need of application. Typically, the dynamic memory allocation in C++ is invoked through *new* operator [6]. The *new* operator is implemented through *malloc()* function.

The constructor may invoke *new* operators explicitly or implicitly. The copy constructor, in contrast to the constructor, can invoke *new* operators implicitly only. The default copy constructor may only perform memberwise copy while the user-defined copy constructor can perform a deep copy. To overload the assignment operator= which implements a deep-copy like assignment between class objects, dynamic memory allocation may be invoked. Finally, the type conversion is used in C++ to convert a user-defined type to a built-in type. Such conversions may also invoke *new* operators. All these cases are unique to C++ languages.

In contrast to the first four cases, the user defined member functions of our fifth case are related to the nature of the application. Moreover, the dynamic memory invocations are quite straightforward. Most likely, *new* operators are invoked directly in the program.

2.2. Memory Tracing Technique of Source Code Instrumentation

There are three approaches to trace the dynamic memory. The first one is rewriting the compiler to handle this. It is a time-consuming and expensive process. Only few programmers perform tracing in such a fashion. The second method is object code insertion which is done directly on the executable or the linking stage [7]. The third approach is source code instrumentation. It includes parsing, analyzing, and converting the original source code into a new modified source code.

The object insertion method reads object files generated by existing compilers and adds tracing instructions without disturbing the symbol table or program logic. The disadvantage of object code insertion is that it is largely instruction-set dependent, and somewhat operating system dependent.

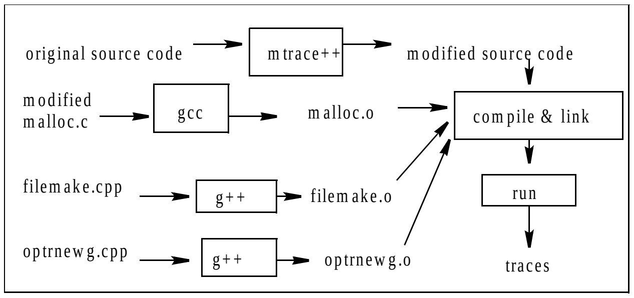
The technique of source code instrumentation is quicker and cheaper than the others. During compilation, it inserts test and analysis functions around every line of source code. The most distinctive advantage compared to the object insertion is the portability. It is quite compatible between different platforms and operating systems. *Mtrace++* utilizes the source code level instrumentation technique to trace dynamic memories of C++ programs.

Mtrace++ takes advantage of source code instrumentation. It inserts two marks at the beginning and the ending of the body of the target member function. The target member functions are constructor, copy constructor, type conversion, assignment operator= overloading and user-defined member function. Each mark contains information of a member function and a class name.

To generate data, basically, three additional object files are needed.: *malloc.o, filemake.o, optrnewg.o*. Between each *beginning* and *ending* mark of invocation scenario, dynamic memory invocation may be occurred. Figure 1 depicts our tracing experiment.

The allocation size and address information comes from the *malloc.o*. This *malloc.o* is a modified *malloc*, originally written by Doug Lea (Lea's malloc version 2.5). Inside the *malloc, malloc(), calloc(), realloc()* and *free()* functions are modified to write result into the data file. Here, we can collect information of number of bytes and returning addresses. *Filemake* module is used to create data file instead of standard output. It allows us to trace interactive or graphic user interface (GUI) programs.

Figure 1 Experimental setup of *mtrace++*



In C++, the operator *new* can be overloaded globally or locally. When we overload the operator *new*, we recognize that the regular operator *new* is implementing a policy of its own, In pure object-oriented programs, all the dynamic memory allocations are performed by *new* operator. The *new* operator is performed by C's malloc library routine. In *mtrace++*, the operator *new* is overloaded both globally and locally. The overloaded global *new* operator is used to distinguish between *new()* and direct call of *malloc()*. When it invokes constructor explicitly, the member data is also allocated in the heap. To collect the invocation of member data, we need to overload the operator *new* locally.

When we run the instrumented application, we get the list of data which contains all the information of dynamic memory invocation. The data format is somewhat simple. *Beginning* mark and *ending* mark represent 'begin' and 'end' of functions accordingly. Each mark contains information of the class name. Between the marks, dynamic memory can be invoked. It starts letter of m *(malloc())*, c *(calloc())*, r *(realloc())*, or f *(free())*.

3. Measurement

Mtrace++ is a run-time C++ dynamic memory tracing tool. During the execution of instrumented binary, it accumulates the traced output information into a data file. *Stat,* which is the analyzing tool for *mtrace++*, creates concise information into tables.

3.1. Analyzing Tool for Mtrace++ (*Stat*)

For the second phase of the tracing, the analyzing tool for *mtrace++*, called *stat*, creates tables which contain the summarized allocation information. During *stat* phase, it also produces the output of the allocation call tree. *Stat* creates tables for every invocation case. Between, *beginning* and *ending* marks, it may have dynamic memory invocation. If the dynamic memory is invoked outside of class member functions, this invocation will not have enclosed *beginning* and *ending* marks. Invocation cases are divided into seven categories (Constructor, Copy Constructor, Overloading Operator=, Type Conversion, Member Data, Application Specific Member Function, and Others). The invocation of 'Member Data' can be included in constructor because the allocation is invoked during constructing an object. 'Others' case is the memory invocation which is implemented from outside of class member functions.

There are two columns in the table for every scenario. The first column indicates the first level of invocation in times and percentage from total allocations. Allocations can be called from direct callers, indirect callers, or through nested functions. The same allocation is called by several different functions for different purposes. The second column shows the lowest level of invocation. Therefore, the second column illustrates where the memory is actually allocated. We will discuss the indirect memory allocation in the section of Dynamic Allocation Call Tree (DACT).

Stat produces analyzed information for each class. With the information, we know which class mostly allocates memory. It is divided into function types as we mentioned above. Knowledge of the memory invocation of each class is necessary in order to determine which class is the most dominant for allocated memory. The following table shows an example of summarized *stat* output:

Figure 2 An example of summarized *stat* output

```
=============================================================
TOTAL MALLOC INVOCATIONS (EXCEPT free()) : 464 (100%)
TOTAL MALLOC INVOCATIONS WITHIN CLASSES FROM TOTAL 464 TIMES
MALLOC INVOCATIONS : 464 (100.00 %)
TOTAL # OF BYTES ALLOCATED: 65603
=============================================================
Analyze by Invocation Scenarios
=============================================================
1. CONSTRUCTOR : 8 times ( 1.72 %)  443(95.47 %)
2. COPY CONSTRUCTOR : 0 times ( 0.00 %)  0( 0.00 %)
3. OVERLOADING OPERATOR : 0 times ( 0.00 %)  0( 0.00 %)
4. TYPE CONVERSION : 0 times ( 0.00 %)  0( 0.00 %)
5. MEMBER DATA : 0 times ( 0.00 %)  6( 1.29 %)
6. APPL SPEC MEMBER FUNCTION : 456 times (98.28 %) 15( 3.23 %)
7. OTHERS : 0 times ( 0.00 %) 0 times ( 0.00 %)
================+============================================
TOTAL : 464 times (100%)
=============================================================
```

3.2. Measuring Invocation by Size

Many C programs tend to have less memory allocation than programs in C++ [5]. C++ programs allocate a significant number of dynamic objects and furthermore, allocate them at a higher rate. Many of those objects can be reused. There are studies of reuse those objects [1]. To reuse objects, we need to know whether the objects with the same size are allocated prolifically. To investigate this, *stat* provides a table of object distribution. This table explicitly describes objects' sizes, frequencies and originations in C++ programs.

The table is columned by its invocation scenario which is the origination of objects. Each invocation scenario has its size and number of invocation in times. The allocation scenario guides programmers to have optimized handling of allocations in class member functions. In the final row of the table, it summaries invocation bytes and times by its type. With the object distribution table, we can get the information of the relationship between size and frequency. In C++, many prolifically allocated small objects can occupy most of memory space [5]. With the knowledge of C++'s object distribution, programmers can design programs in an efficient way of dynamic memory management. For example, programmers can design a special kind of segregated allocator in which only highly referenced objects are processed. This approach will take advantage of high-speed allocation.

3.3. Dynamic Allocation Call Tree (DACT)

To understand the memory allocation behavior of a program, allocated objects must be identified. In C, only the object size is passed to *malloc*; the type of the object being allocated is not known at allocation time. DACT provides allocation information with paths: what sizes of objects are allocated and what types of memory invocations correspond to those allocations. With statistical data, it is hard to know how the allocation is performed. DACT tells you all the paths of indirect callers.

At the same time when *stat* generates statistical information, it creates DACT into a file. This instrumented data file contains the information of all the allocation routes. In many cases, same allocation can be called from different callers. The following data shows an example of DACT list which is collected from Guavac. It is a GNU version of Java compiler compliant written in C++. In this presentation, we can easily understand the paths of memory invocations.

DACT contains information of types, sizes, addresses, and paths of allocated objects. In Table 1, it shows all the chains of classes when memory is allocated. This chain can be quite complex with many entries. If we provide only the last entry of the path, it will be difficult to know where the object is actually allocated. With DACT, programmers can have information about the originations of allocations.

Table 1: An example of DACT

type	size	address	path
m	24	135605320	CCompiler
m	24	135605288	CCompiler->CJavaDirectory-> CFilePath_FUNC_NAME_IS_IsFile -> CFilePath_FUNC_NAME_IS_Exist
m	26	135578760	CCompiler->CJavaDirectory
m	24	135605256	CCompiler

DACT can also help programmers to fix the memory leak problems. Memory leak occurs when a piece of dynamically allocated memory can no longer be referenced and freed. The allocation should be freed explicitly by programmers before it is not accessible. However, in many cases, programmers accidently forget to release the memory. With DACT, programmers can easily detect memory leaks and can remove unnecessary memory allocations. The indirect allocation paths have already been proven as a extremely useful information, while the direct caller information allows programmers to understand the allocations hardly and is also less useful [8].

3.4. Depth of Allocation Tree (DAT)

DAT measures maximum hierarchical depth of memory invocation. It is the length from the beginning of the first branch of the indirect callers to the direct caller in which the object is actually allocated. The bigger the number of DAT, the harder the maintenance will be required. It would increase the possibility of memory allocation error.

DAT can be increased due to coupling. There are three different types of coupling: coupling through inheritance, coupling through abstract data type, and coupling through message passing [9]. When a class inherits from another class, directly or indirectly, the two classes are coupled. This coupling will break the encapsulation that OO programming provides. However, this type of coupling does not affect to increase DAT. The reason is that DAT is limited within a class member function. When one class uses another class as an abstract data type, two classes are coupled. One class is coupled with another class through the use of abstract data type because it uses the class declaration as data type. The message passing coupling occurs when one class sends a message to an object of another class, without involving the two classes, through inheritance or abstract data types.

DAT has an important role to identify the complexity of allocations. In many cases, programmers have difficulties in storage management which are caused by high level of allocation depths. In C++, allocation depths can be increased without recognition of programmers due to C++'s nature of class hierarchy. Using DAT, programmers can control depths of allocation paths. This allows programmers minimize complexity to manage dynamic storage easily.

4. Conclusions

Dynamic memory management has been a high cost component in many software systems. Even though it has powerful and convenient features of dynamic memory management in C++, many developers have difficulties in understanding the implication of dynamically allocating memory blocks in C++ programs. This paper presents a memory tracing tool which allows to investigate dynamic memory allocation in C++ programs. To investigate the memory allocation, we need hypothesized situations that invoke the dynamic memory management explicitly and implicitly. Theses situations occurs in: constructors, copy constructors, overloading assignment operator=, type conversions and application specific member functions.

In this paper, *mtrace++* is introduced as a C++ memory tracing tool. *Mtrace++* traces member functions of a class which directly or indirectly are responsible for allocation. *Mtrace++* measures sizes and frequencies of objects, and arranges these objects into the hypothesized scenarios. This allocation information is important to know because it allows to find out the most frequently allocated objects and where the objects come from. After the object are identified, programmers can address a special policy to tune the objects to have a better performance. Furthermore, *mtrace++* traces all indirect paths which can include different classes and different member functions. The allocation path implies the complexity of the program. If allocation paths are too deep, it will be hard to maintain the objects. With the help of allocation call tree, programmers can easily and correctly maintain allocated objects.

Using *mtrace++*, programmers can identify where and why dynamic memories are allocated. This helps programmers to understand the memory allocation behavior of their programs. Furthermore, the allocation behavior can provide ideas to develop strategies for high-performance memory management to solve the overhead of C++'s large number of dynamic memory invocations.

5. References

[1] Paul R. Wilson, Mark S. Johnston and Michael Neely, and David Boles, "Dynamic Storage Allocation A Survey and Critical Review," *Proceeding of International Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.

[2] Benjamin Zorn and Dirk Grunwald, "Empirical Measurements of Six Allocation intensive C Programs," Technical Report CU-CS-604-92, Department of Computer Science, Univ. of Colorado, Boulder, CO, July 1992.

[3] M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Transactions on Computers*, pp. 357-366, March 1996.

[4] David Detlefs, Al Dosser, and Benjamin Zorn. "Memory allocation costs in large C and C++ programs," *Software - Practice and Experience,* pp. 527-542, June 1994.

[5] Brad Calder, Dirk Grunwald, and Benjamin Zorn, "Quantifying Behavioral Differences Between C and C++ Programs," Technical Report CU-CS-698-95, Department of Computer Science, Univ. of Colorado, Boulder, CO, January 1995.

[6] Bjarne StrouStrup, *The C++ Programming Language*, Third Edition, Addison-Wesley, 1997.

[7] Jim Pierce, "IDtrace-A Tracing Tool for i486 Simulation," Technical Report CSE-TR-203-94, Department of Electrical Engineering and Computer Science, Univ. of Michigan, Ann Arbor, MI, 1994.

[8] Benjamin Zorn and Paul Hilfinger, "A Memory Allocation Profiler for C and Lisp Programs," Technical Report (DARPA Contract No: N00039-85-C-0269), Department of Computer Science, Univ. of Colorado, Boulder, CO, 1990.

[9] Wei Li, "Another Metric Suite for Object-Oriented Programming," *Journal of Systems and Software,* 1998.