ELSEVIER

The Journal of Systems and Software 57 (2001) 107–118

The Journal of Systems and Software

www.elsevier.com/locate/jss

A study of the allocation behavior of C++ programs

J. Morris Chang [*], Woo Hyong Lee, Witawas Srisa-an

Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA

Received 21 November 1999; received in revised form 15 April 2000; accepted 28 April 2000

Abstract

The object-oriented programming (OOP) language systems tend to perform object creation and deletion prolifically. An empirical study has shown that C++ programs can have 10 times more memory allocation and deallocation than comparable C programs. However, the allocation behavior of C++ programs is rarely reported. This paper attempts to locate where the dynamic memory allocations are coming from and report an empirical study of the allocation behavior of C++ programs. Firstly, this paper summarizes the hypothesis of situations that invoke the dynamic memory management explicitly and implicitly. They are: constructors, copy constructors, overloading assignment operator =, type conversions and application-specific member functions. Secondly, the development of a source code level tracing tool is reported as a procedure to investigate the hypothesis. Most of the five C++ programs traced are real-world applications. Thirdly, allocation patterns, object size and age distribution are summarized. Among other things, we found that objects tend to have a very short life-span, and most of them are created through constructors and copy constructors. With these findings, we may improve the performance of dynamic memory management through, a profile-based strategy or reusing objects. © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Software tools; Dynamic memory management; Object-oriented programming

1. Introduction

It has become a cliché to say that object-oriented systems are more powerful but slower than software written in most other languages. The latest example is Java which is presently 10–20 times slower than C. The growing popularity of C++ in software development industry implies that more applications will be developed in this language. To study various performance issues (e.g., the behavior of its dynamic memory allocation and deallocation), scientific analysis (e.g., quantitative analysis based on some new tracing techniques) is needed. The results of such research can be used in three ways. Firstly, a programmer's guideline can be derived from the research results. By following this guideline, programmers can avoid using language features that can invoke dynamic memory implicitly. Secondly, architectural support (e.g., instruction set architecture or coprocessor) can be developed based on the results. For example, hardware-assisted memory management (Chang and Gehringer, 1996) can be an effective way to reduce performance bottleneck in memory intensive applications. Thirdly, software tools for memory leak detection can be developed based on the research. Moreover, the performance of dynamic memory management can be improved if the allocation patterns are known.

Dynamic memory management (DMM) has been a high cost component in many software systems (Wilson et al., 1995; Neely, 1996). A study has shown that memory intensive C programs can consume up to 30% of the program run-time in memory allocation and liberation (Zorn and Grunwald, 1992). The object-oriented programming (OOP) language system tends to have object creation and deletion prolifically. In (Detlefs et al., 1994; Calder et al., 1995), the empirical data show that C++ programs perform an order of magnitude more allocation than comparable C programs. However, the causes are not clear. No data has been reported regarding the memory allocation pattern in C++ applications at this point. This illustrates the need for a quantitative analysis on the allocation patterns to outline an efficient programming style.

This paper presents a new approach to investigate memory allocation behavior at source code level. We

[*] Corresponding author. Tel.: +1-312-567-5329; fax: +1-312-567-5067.
E-mail addresses: chang@cs.iit.edu, chang@charlie.iit.edu (J.M. Chang), leewoo@charlie.iit.edu (W.H. Lee), witty@charlie.iit.edu (W. Srisa-an).

start with a classification of all the possible situations that may invoke the dynamic memory management (DMM) in C++. In our hypotheses, these memory allocation patterns are related to either applications or language features of C++. 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. However, novice C++ programmer (and with some experience in C programming) could easily write a C++ program without practicing object-oriented paradigm. In this case, of course, our approach could not be effective in the investigation.

To investigate our hypotheses on the allocation behavior, we have developed a tracing tool to instrument C++ source programs. This tool takes C++ source code as input and inserts C++ statements into the hypothesized allocation patterns. While the instrumented C++ programs are being executed, the traces of dynamic memory allocation will be generated. These traces, then, will be analyzed by another tools to obtain the important statistics (allocation sources, average object size, and average life-span). The C++ programs traced in our experiment are publicly available applications. Most of these programs are real-world applications with high demands on the memory system. The traced programs include portable 3D engine (crystal space), robotics tool box (roboop), Portable Document Format (PDF) files viewer (Xpdf), internet file transfer program (lftp), and C++ benchmark suite (bench++).

A very preliminary result based on two C++ programs can be found in (Chang and Lee, 1998). However, these two programs do not follow the object-oriented programming paradigm and do not represent wide range of applications. While currently there is no standard benchmark for C++ programs, we have studied five C++ programs which represent wide range of applications. Additionally, we also perform a detailed analysis on C++ object's behaviors. There have been many published research that deal with object behavior in languages like LISP, ML, Java, and Smalltalk; however, to the best of our knowledge, there has not been any detailed study on the object's behavior of C++ applications. In this paper, the behaviors such as object life-span, average object size, and size distribution are presented. We also provide the allocation frequency originated from different allocation sources such as constructor, copy constructor, application-specific member functions, overloading assignment, type conversion, and others.

The rest of this paper is organized as follows. Section 2 summarizes the scenarios that invoke the DMM in C++. Section 3 details the source code tracing tool. Section 4 shows the tracing results. Section 5 describes differences between the proposed research and previous research. Section 6 presents the conclusions of this paper.

2. Cases of dynamic memory allocation in C++

In this section, five distinctive allocation cases are described. 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. Typically, the dynamic memory allocation in C++ is invoked through *new* operator (Stroustrup, 1997). 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, type conversion is used in C++ to convert a user-defined type to a vice versa. Such conversion may also invoke new operators. All these four cases are unique to C++ language.

In contrast to the first four cases, DMM calls in user-defined member functions are related to the nature of the applications under development. DMM invocations are done by the programmers. In most cases, *new* operators are invoked directly in the program. The following subsections detail each case with simple examples.

2.1. Constructors

Constructors can be subdivided into categories of explicit invocation or implicit invocation. In explicit case, data members are located in the heap. Our tracing tool (mtrace++) can identify these data members. An example of constructor is described in the following String class:

```
#include <iostream.h>
#include <string.h>
class String {
public:
String(char *ch = "\0")
{  len = strlen(ch)+1;
   name = new char[len];
   strcpy(name,ch);
}
void print() {cout << name << endl; }
private:
char *name;
int len;
};
```

In the above example, only one constructor is provided. The constructor employs the default argument to implement a default constructor (which requires no arguments). This constructor will be invoked implicitly in the object creation and initialization. A main function is included below to show the invocation of the constructor.

```
main()
{
                                        //statements
                                        NO:
  char *s1 = "Illinois";                // 1
  char s4[] = "Technology";             // 2
  char ch[3] = { 'o','f','\0'};         // 3
  String str1(s1);                      // 4
  String *str2;                         // 5
  String str3;                          // 6
  String str4;                          // 7
  str2 = new String("Institute");       // 8
  str3 = String(ch);                    // 9
  str4 = s4;                            // 10

  str1.print();
  str2->print();
  str3.print();
  str4.print();
}
```

To instantiate the class String, the String constructor is invoked implicitly in statements 4, 6 and 7 and invoked explicitly in statement 8. Statement 8 uses *new* to create and initialize a String object. It invokes the String(char *) constructor. It is worth noting that dynamic memory allocation is invoked twice in statement 8; one from the constructor and one from the allocation for the private data members. The constructor can also be used in type conversion. In statement 9, constructor is invoked explicitly to convert a string into a class object. For the similar type conversion, however, the constructor is invoked implicitly in statement 10. The type conversion that employs the constructor is capable of converting a built-in type (e.g., char *) to a user-defined type (e.g., class). The conversion from user-defined type to built-in type which may involve operator new will be discussed in a later section.

It is worth noting that there are three things happening in statement 9: (a) invoking constructor to create a class object (i.e., a temporary object), (b) performing the memberwise assignment from newly created object to the object *str3* and (c) destructing the just created object. Please note that the data member of the temporary object here does not invoke dynamic memory allocation, instead, the data member is placed on the stack as a compiler generated automatic variable. However, the constructor in (a) may invoke dynamic memory allocation. Whenever the constructor is invoked, the operator new will be invoked. Thus, the object is allocated dynamically in the heap region.

2.2. Copy constructors

The copy constructor, in contrast to the constructor, can be invoked implicitly only. The default (i.e. compiler generated) copy constructor can perform memberwise copy. This is insufficient for the class which has a pointer as a private data member (e.g., String class has data member *name* in char *), since the default copy constructor will copy only the pointer but not the entire string. The memberwise copy also refers to *shallow copy*. A user-defined copy constructor must be defined before a *deep copy* is performed. Apparently, a deep copy constructor will invoke dynamic memory allocation. The copy constructor may be invoked in the following ways:

- passing a class object to a function through call-by-value
- returning a class object from a function through return-by-value
- initializing a class object

```
class String{
public: *
  // (default) constructor
  String(char *ch = "\0"){
      len = strlen(ch)+1;
      cout << "cons\n";
      name = new char[len];
      strcpy(name,ch);
    }
  // copy constructor
  String(const String & s) {
      len = s.len;
      cout << "copy\n";
      name = new char[len];
      strcpy(name,s.name);
    }
 void print(){cout << name <<
endl; }
private:
  char *name;
  int len;
};
void call_by_value(String local-
String)
{
  localString.print();
}
String return_by_value(void)
{
  String tmp("Hello\n");
  return(tmp);
}
main()
{
                                  //statement #
    void call_by_value(String);        // 1
    String return_by_value(void);      // 2
    String str1("Illinois");           // 3
    String str2 = str1;                // 4
    call_by_value(str1);               // 5
    (return_by_value( )).print( );     // 6
}
```

The copy constructor defined in class String is invoked in statement 4 for the initialization. [1] The same copy constructor is invoked in statement 5 to created a local copy of the class object to be used in function *call_by_value*(). The function *return_by_value*() invoked in statement 6 will invoke the copy constructor as the class object is returned from the function.

2.3. Overloading assignment operator =

The default (i.e., compiler generated) assignment operator will perform memberwise assignment. To achieve a "deep-copy" like assignment between class objects, overloading the assignment operator is a must. Thus, the member function operator = () that can perform a deep-copy *may* invoke dynamic memory allocation. In the String class presented before, an assignment operator can be overloaded through the following member function:

```
    // overloading assignment operator
    String & operator = (const String & s) {
        if(len < s.len) {
          delete name;
          name = new char[s.len];
          len = s.len; }
        strcpy(name,s.name);
        return(*this);
    }
```

Statement that invoked operator = ( ) member function is presented in the following main function.

```
  main()
  {                                  // statement #
    String str1("Illinois");         //1
    String str2("Tech");             //2
    str2 = str1;                     //3; assignment
    str2.print( );                   //4
  }
```

The statement 3 (i.e., **str2 = str1**) invokes the user-defined assignment operator, then dynamic memory allocation may be invoked. It is clear that this can be extended to any function (global or member) that is used to overload C++ operators. Moreover, a programmer may make any function (member, friend or global) to include operator new.

2.4. Type conversions

C++ provides a mechanism to convert a user-defined type to a built-in type (e.g., int, char *). Such a user-defined type conversion can be implemented through a special member function with a form of *operator type( )* {...}. Few rules about these functions: (1) They must be non-static member functions. (2) They should neither have parameters nor have a declared return type. (3) They must return an expression of the designated type.

These conversions occur implicitly in assignment expression, relational expression, arguments to functions (e.g., an overloaded function), and values returned from functions. The conversions can occur explicitly as the cast operator (type) is applied. Let us add two such type conversion functions to our String class. One would convert a class object into *int* while another would convert a class object to *char* *. The latter will invoke operator new.

```
    operator char *( ) {
    char *p = new char[len];
          strcpy(p,name);
          return(p);
    }
    operator int( ){ return(len-1);}
```

It is worth noting that the char * conversion did not simply return the value of the private member. Otherwise, this would violate the integrity of String objects. The following main function will invoke such type conversions in different ways.

```
main( )
{                                    // statement #
  char *sp;                          // 1
  String str1("Illinois");           // 2
  String str2("Tech");               // 3
  int func(String);                  // 4

  cout << (sp = str1) << endl;       // 5
  cout << (char *)str2 << endl;      // 6
  cout << func(str1) << endl;        // 7
  if(str1 > 5)                       // 8
  cout << "More than 5
characters\n";
}

int func(String s)
{
  return(s);
}
```

[1] The *initialization* should be distinguished from the *assignment* which will invoke overload assignment operator (described in Section 2.3).

The statements 5 and 6 invoke conversion, operator char *( ), implicitly and explicitly respectively. The statement 7 invokes conversion, operator int( ), through the value returned from function call. The statement 8 invokes conversion, operator int( ), implicitly. The type conversion that employs constructor to convert from a built-in type (e.g., char *) to a user-defined type (e.g., class object) has been discussed in earlier section (i.e., constructor).

2.5. Application-specific member functions

Member functions that are specific to the application are discussed in this section. Apparently, the way that a member function invokes DMA can never be defined easily. It all depends on the need of application. The following example shows two member functions are part of the class *VPrinter*. They provide different service to the class object.

```
class VPrinter{
private:
        char *_name;
public:
        VPrinter( );
        ~VPrinter( );
};
void VPrinter::open( )
{
        if (!_name) {
        char *name = "printer.ps";
        _name = new char[strlen(name)+1];
        strcpy(_name, name);
        }
...
}
void VPrinter::setup(char *fn)
{
        if (_name) delete [] _name;
        _name = new char[strlen(fn)+1];
        strcpy(_name, fn);
....
}
```

The member functions, *setup* and *open*, invoke *new* operator explicitly. It is easy to see that such DMA invocations are very different from the first four cases mentioned earlier. The fifth case has to do with application only. In this paper, we separate the fifth one from the first four cases.

3. Tracing tool

To get a quantitative analysis of the above-mentioned hypotheses on the allocation behavior, we have developed a tracing tool to instrument C++ programs. This tool takes C++ source code as input and inserts C++ statements, as marks, into the hypothesized allocation patterns. This tool is coded in C++ and inserts beginning and ending marks for each dynamic memory allocation scenarios (i.e., constructor, copy constructor, type conversion, and assignment operator = overloading). These marks contain names of member functions and classes. Thus, the class member functions can be traced as they are invoked. Moreover, the implementation of DMM in C++ (i.e., *new* and *delete* operators) is also modified to the way that each invocation to the DMM will generate traces. In case of interactive or Graphical User Interface programs, Filemake object file is used to create data file instead of standard output. The diagram depicts our tracing experiment (Fig. 1).

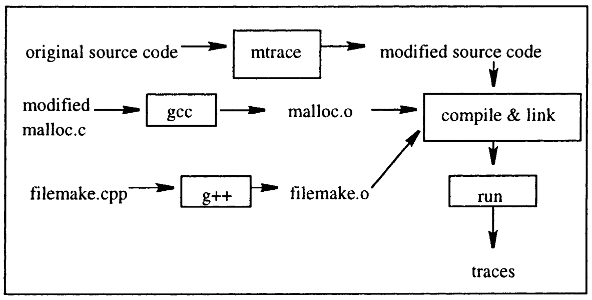

Fig. 1. Experimental setup.

While the instrumented C++ programs are being executed, the traces of dynamic memory allocation will be generated. These traces, then, will be analyzed by another tool to obtain the statistics. The invocation frequency and size of the allocation, for example, are collected through the analyzer. The detailed data are presented in the next section.

4. Results

As we started the project, we fail to find a C++ benchmark suite from the System Performance Evaluation Corporation (http://www.spec.org/osg). Then, we collected many programs which claimed to be C++ programs through the internet. However, we soon found that many of these C++ programs did not utilize OOP technique. Precisely, these programs were more like a C program than a C++ program. Very often, only one or two classes were defined and instantiated in entire program. Thus, these C++ programs were excluded from our experiment.

In this research, we have chosen five C++ programs which are highly memory intensive programs and available publicly. Those programs includes 3D graphic engine (*Crystal Space*), object-oriented robotics tool box (*Roboop*), ftp client (*Lftp*), pdf file viewer (*Xpdf*) and object-oriented benchmark suite (*bench* + +). *Crystal Space*, *Lftp* and *Xpdf* are screen interaction programs, *Roboop* uses predefined user input, and *Bench*++

basically measures process time for each event. These programs invoke dynamic object creation ranging from few hundred thousands to several millions.

Table 1 illustrates characteristics of the traced programs. In four out of five applications, the average object sizes are less than 50 bytes. Crystal Space, on the other hand is a portable 3D engines. Thus, the average object size is larger. Nonetheless, Crystal Space still yields average object size of 306 bytes. Fig. 2 depicts the size distribution of the traced programs. In all applications, small objects (of sizes less than 2 Kbytes) are created very frequently. It is worth noting that Bench++ is not included in the histogram. Bench++ only invokes objects of three sizes, 24, 31, and 61 Kbytes. Twenty-four bytes objects are invoked only twice. Sixty-one kilobytes object is invoked only once. The remaining invocations are of 31 bytes.

A brief summary of the traced programs is given in Table 2. The line of code, number of class and the ftp site for each program are listed. The percentage of execution time used for object creation and liberation for each program is also summarized in Table 2. They are ranging from 14% to 50%. These numbers are collected through *gprof* version-2.9.1, a call graph execution profiler from GNU (ftp://ftp.gnu.org). The detailed results of each program are presented in the next subsections.

4.1. Crystal space

Crystal Space is a portable 3D engine written in C++. It supports colored lights, mipmapping, portals, mirrors, alpha transparency, reflecting surfaces, 3D sprite, 32-bit display support and etc. (http://crystal.linux-games.com) Crystal Space is currently a large open source projects. Three input files are provided with the source code to setup the background environments.

These three inputs are applied to *Crystal Space*. For each input, there are about 71–100K objects created. About 62% of the objects are created through copy constructors while 37% are created from constructors. This indicates that 99% of the objects are created from the four hypothesized cases.

Table 4 depicts detailed object information created in Crystal Space. It is worth noting that we use Dynamic Memory Allocation (DMA) to express the number of allocation invoked. This term, DMA is different than DMM because DMM expresses both allocation and deallocation. The first column indicates the size ranges that objects are invoked. The size-distribution indicates the percentage of objects within this size range as compared to the total number of objects. For each object, the aging process is based on the number of allocations that occur within the life-span of an object. The average

Table 1
Characteristic of traced programs

Name	Allocation calls	Deallocation calls	Avg. allocation size (bytes)	Memory requested (bytes)
Bench++	1,092,207	1,092,204	31.05	33,919,728
Crystal space	121,480	137,027	306.19	37,195,962
Lftp	110,866	110,762	21.40	2,372,534
Roboop (input 1)	889,473	889,472	30.33	26,984,794
Xpdf	195,831	191,818	45.85	8,978,851

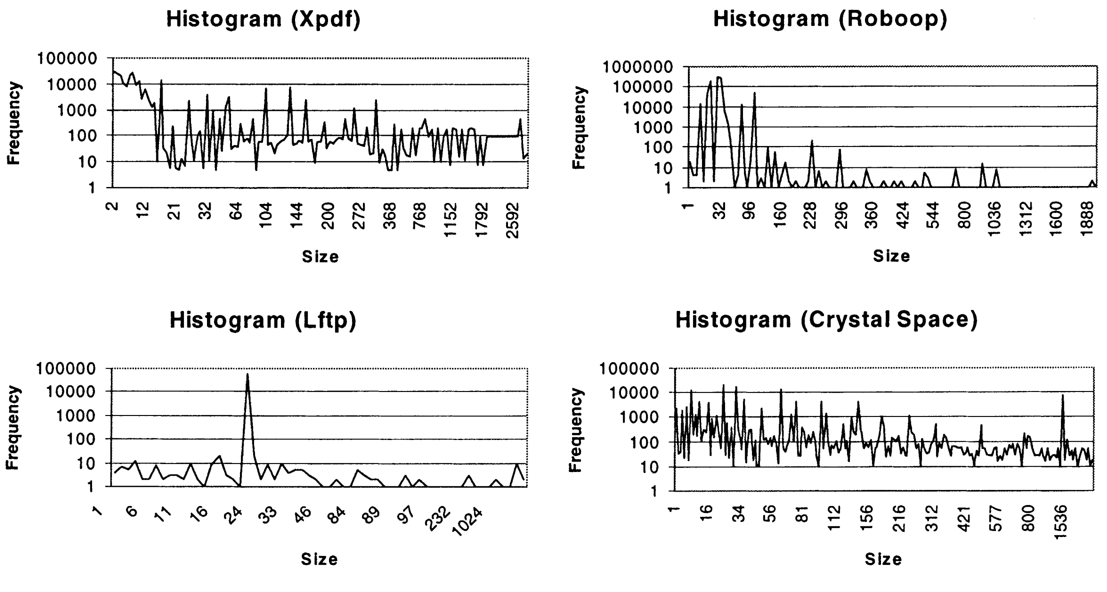

Fig. 2. Size distribution of traced application.

Table 2
Summary of traced programs

Program name	Line of code	Number of class	Ftp site	% of overall execution time used for object creation and liberation
Crystal space	39,279	154	http://crystal.linuxgames.com	18
Roboop	19,434	111	http://www.ind2.polymtl.ca/ROBOOP	23
Lftp	54,654	59	ftp://pub/Linux/system/network/file-transfer	21
Xpdf	25,866	74	ftp://ftp.aimnet.com/pub/users/derekn/xpdf	14
Bench++	17,724	118	http://www.research.att.com/~orost/bench_plus_plus.html	50

life-span is then calculated based on total age of objects within each range divided by the number of objects within each range. Majority of objects (89%) in Crystal Space are less than 256 bytes. Objects with short life, according to Table 4, are within the range of 32–63 bytes. Since objects are evenly distributed from 16 to 127 bytes (about 20% per range). The most heavily invoked sizes (24, 32, and 64 bytes) are individually included in Table 4. The average life-spans of those sizes are also shown.

From Table 4, the results indicate that sizes, which are invoked frequently tend to have short life-spans (the life-spans of frequently invoked object is much smaller than the average size). The last column also provides the major sources for those allocation calls. For Crystal Space, the majority of calls are made by copy constructor. From Table 3, constructors are responsible for 37% of DMA calls and copy constructors are responsible for 62%. Table 4 indicates that about 40% of objects are called by copy constructors and 13% are by constructors. Since we only report the major allocation source within that range, the remaining objects invoked by constructors and copy constructors may not be reported because they are scattered among other ranges.

Table 3
Percentage of the DMA invocations in CS

	Input 1 (%)	Input 2 (%)	Input 3 (%)
Constructor	37.3	37.4	38.4
Copy constructor	62.6	62.3	61.5
Others	0.1	0.1	0.1
Total invocations	121,168	90,982	71,390

4.2. Roboop

Roboop is a C++ robotics object-oriented programming tool-box suitable for synthesis, and simulation of robotic manipulator models in an environment that provides "*MATLAB*-like" features for the treatment of matrices (http://www.ind2.polymtl.ca/ROBOOP). It is a portable tool that does not require the use of commercial software. A class-named Robot provides the implementation of the kinematics, the dynamics and the linearized dynamics of serial robotic manipulators.

There are two binaries in this package (*demo* and *bench*). *Demo* is an example of Roboop simulation. *Bench* measures time of the Jacobian, Torque, Acceleration. Each binaries are tested with three different inputs. The results are summarized in Table 5. For the *demo* part, the total number of the DMA invocation for each input is ranging from 300K to 880K. For the *bench* part, these numbers are between 186K and 2278K. In both programs, about 70% of DMA invocation are from the four hypothesized cases. It is worth noting that the allocation patterns are very consistent across all the inputs.

Table 4
Object information for crystal space

Size ranges (bytes)	Size distribution (%)	Average life-span[a] (# of allocations)	Major sources
0–7	5.23	79735.44	–
8–15	14.51	29380.59	Copy constructor
16–31 (excluding 24)	5.52	75123.60	–
24	15.01	6429.87	Copy constructor
32–63 (excluding 32)	19.85	38930.41	–
32	12.58	1230.66	Constructor
64–127 (excluding 64)	21.50	54329.24	–
64	10.66	1455.82	Copy constructor
128–255	8.20	96971.84	–
256–511	2.37	71716.01	–
512–1023	1.39	66511.57	–
1024+	6.44	4915.21	–

[a] We choose average instead of median because we find that object tend to have longer life toward the end of the range which would make the median value less accurate than the average.

In Roboop running in demo mode, majority of objects are from 16 to 63 bytes. Table 6 also illustrates that objects in those ranges are also short-lived. These results also indicate that objects of sizes that are frequently invoked also have short life-spans. Table 6 also demonstrates that majority of dynamic memory management calls are made by constructors. It is worth noting that about 60% of objects within 16–31 bytes are created by the constructors. This number corresponds to 36% indicated by Table 5.

4.3. *Lftp*

Lftp is a shell-like command line *ftp* client. It can retry operations and does re-get automatically. It can do several transfers simultaneously in background. You can start a transfer in background and continue browsing the ftp site or another one. All of this can be done in one process. Background jobs will be completed in nohup mode if you exit or close modem connection. There are two binaries in the *Lftp* (*ftpget* and *lftp*). The *Ftpget* allows to run macros. Therefore, it can avoid

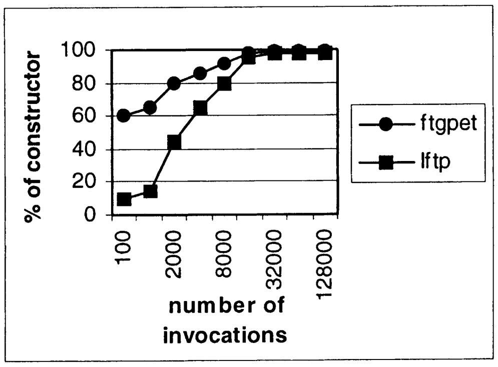

Fig. 3. Allocation pattern in constructor.

Table 5
Percentage of the DMA invocation in Roboop

	Demo			Bench		
	Input 1 (%)	Input 2 (%)	Input 3 (%)	Input 1 (%)	Input 2 (%)	Input 3 (%)
Constructor	36.0	35.9	35.9	32.3	33.3	32.5
(Data Member[a])	(5.7)	(5.6)	(5.6)	(2.2)	(1.9)	(2.4)
Copy constructor	1.5	1.6	1.6	12.9	2.5	0.6
Overloading =	21.8	21.9	21.9	21.5	19.0	24.2
Type conversion	10.6	10.5	10.5	7.5	12.6	9.1
Application-specific member function	30.1	30.1	30.1	25.8	32.6	33.6
Others	0.1	0.1	0.1	0.0	0.0	0.0
Total invocations	889,473	386,972	310,049	186,134	968,134	2,278,607

[a] The allocation made by the data member but has already been included in the total allocation made by constructor.

Table 6
Object information for Roboop (Demo input 1)

Size range (bytes)	Size distribution (%)	Average life-span (# of allocations)	Major sources
0–7	0.00	5130.14	–
8–15	1.50	90.44	–
16–31	61.61	39.64	Constructor
32–63	29.75	34.77	overloading =
64–127	7.07	191.26	–
128–255	0.00	3067.02	–
256–511	0.01	10965.72	–
512–1023	0.01	6067.5	–
1024+	0.01	30830.92	–

Table 7
Percentage of the DMA invocation in Lftp

	Ftpget			Lftp		
	Input 1 (%)	Input 2 (%)	Input 3 (%)	Input 1 (%)	Input 2 (%)	Input 3 (%)
Constructor	99.8	99.7	99.8	83.9	96.0	97.6
(Data member)	(99.7)	(99.6)	(99.7)	(83.8)	(95.9)	(97.5)
Application-specific member function	0.1	0.2	0.1	13.0	3.2	1.8
Others	0.1	0.1	0.1	3.1	0.8	0.6
Total invocation	177,412	213,186	235,591	110,866	286,388	265,351

Table 8
Object information for Lftp

Size range (bytes)	Size distribution (%)	Average life-span (# of allocations)	Major sources
0–7	0.07	15.24	–
8–15	0.05	2634.13	–
16–31	99.74	7.12	Constructor
32–63	0.05	2681.6	–
64–127	0.05	4471.6	–
128–255	0.00	0	–
256–511	0.01	N/F	–
512–1023	0	0	–
1024+	0.026	0.55	–

Table 9
Percentage of the DMA invocations in Xpdf

	Input 1 (%)	Input 2 (%)	Input 3 (%)
Constructor	87.8	85.2	95.2
Application-specific member function	12.2	14.8	4.8
Total invocation	103,971	195,831	466,977

Table 10
Object information for Xpdf

Size range (bytes)	Size distribution (%)	Average life-span (# of allocations)	Major sources
0–7	38.12	1795.38	Constructor
8–15	33.32	1908.41	Constructor
16–31	7.53	2570.04	Constructor
32–63	8.33	3107.14	–
64–127	3.08	4671.94	–
128–255	6.55	3383.68	–
256–511	1.71	1998.14	–
512–1023	0.68	3707.39	–
1024+	0.70	4845.72	–

user-interaction in transferring files. The *Lftp* is similar to ftp.

From Table 7, the memory allocation of these two programs are primary from constructors. The total number of DMA invocation for each program is ranging from 110K to 286K. Fig. 3 shows the percentage of DMA invocation made by constructors. At the initial stage, the portion of constructor is relatively small. However, during the final stage of execution, most of invocations are made by constructors.

In Lftp, over 99% of objects are ranging from 16 to 31 bytes. Table 8 indicates that object within that range are short-lived. The majority of allocation calls are made by constructors.

4.4. Xpdf

Xpdf is a viewer for Portable Document Format (PDF) files. *Xpdf* runs under the X Window System on UNIX, VMS, and OS/2. *Xpdf* is designed to be small and efficient. It does not use the Motif or X libraries. It only uses standard X fonts. Again, three inputs are applied to *Xpdf*. The total number of DMA invocation is ranging from 100K to 466K. About 85–95% of DMA invocations are from the four hypothesized cases. Constructor is the major source for the object creation (Table 9).

About 80% of objects in *Xpdf* are within 0–31 bytes. Objects in these ranges tend to have shorter life. The results also confirm the correlation between frequently allocated size and object life-span. As stated earlier, objects of sizes that are frequently invoked tend to have shorter average life-span. Again, the majority of allocation calls are made by constructors (Table 10).

4.5. Bench++

The Bench++ suite is designed to measure the performance of the code generated by C++ compilers (http://www.research.att.com/~orst/bench_plus_plush.html) [2]. It measures run-time performances for three test groups:

[2] http://www.ooc.com/ob.html.

Table 11
Percentage of the DMA invocation in bench++

	Dhrystone	Packed array	Unpacked array
Constructor	0.01%	41.7%	41.7%
Copy constructor	99.9%	58.3%	58.3%
Total invocation	4,369,006	26,214,002	26,214,002

Traditional Benchmarks (Dhrystone, Whetstone, Henessy), Applications, and Language Features. Among benchmark test cases, we traced three binaries: Dhrystone, unpacked bit array, and packed bit array. The Dhrystone benchmark was originally written in Ada, and was based on statement frequencies found in an examination of hundreds of high-level language programs. It gives a first performance indication which is more meaningful than MIPS instruction sets. Packed bit array and unpacked bit array tests measure the effect of minimizing data space by those array.

In the trace data of dhrystone, almost all of the DMA invocation are from copy constructors. In the case of array tests, the DMA allocation are from constructors and copy constructors which have almost equal portions (41.7% and 58.3%). It is worth noting that these benchmarks created objects as high as 26 million (Table 11).

In Bench++, nearly all objects falls with in the range of 16–31 bytes (2 objects at 24 bytes and millions at 31 bytes). Again, objects of the size that are frequently invoked have short average life-span (2.49 allocations in this case). We also find that the majority of memory allocations are invoked by Section 2.2 (Table 12).

4.6. Relationship between age and invoked-size frequency

The results from five traced files demonstrate a correlation between the frequency of invoked size and object life-span. That is if objects belong to sizes which are frequently invoked, those objects tend to have short life-span. In this section, experiments are performed on the same traced file to see if the reverse of the above correlation is also true. The reversed correlation is that objects with short life-span are invoked more frequently than objects with longer life-span. Fig. 4 depicts the findings.

In all applications, objects with short life-span are invoked more frequently. Again, age distribution of Bench++ are intentionally omitted due to its lack of variety. In Bench++, objects that have life-span of zero (allocate and free without any allocation in between) and one are allocated several million times. On the other hand, object of size 61 Kbytes is allocated only once.

5. Comparison with other research efforts

In 1993, Barrett and Zorn (1993) reported the results of object's lift-span study for five memory intensive C programs. Their results indicates that between 90–100% of objects die young (they die within 32 Kbytes of allocation). Similarly, a study in Standard ML/New Jersey by Stefanovic and Moss (1994) also indicates 90–100% mortality rate. Shaw (1988) performs similar study using four Lisp programs and finds that the mortality rate is between 75–95% within 32 Kbytes of allocation. Additionally, Detlefs et al. (1994) also a perform life-span study for 11 C/C++ programs (nine C programs and two C++ programs). They find that the average object size to be between 15 and 249 bytes.

It is worth noting that very few studies have been concentrated on the comprehensive analysis of object behavior in C++. Detlefs et al. (1994) seem to be the only one who have performed object behavior study for C++, and yet only two programs were studied. In this paper, a comprehensive study of objects' behaviors and characteristics is fully explained. The study includes five sources of memory allocations, the object life-spans, average object size, size distribution, age distribution, and survival rate. Fig. 5 illustrates the survival rate for the traced programs.

While the number of allocations between object creation and liberation server as the aging criteria, it alone does not provide the complete insight on the survival rate. We use milestones, (e.g., 500 Kbytes, 1 Mbytes, 5 Mbytes, and 10 Mbytes) as entry points to measure the survival rate. The milestones represent the total amount of accumulated memory requested by the applications. For example, we find that after 1 Mbytes of allocation in Xpdf, only 22% of object has yet to be freed. While it is true that C++ do not use a garbage collector, the milestone can give us insight on the persistency of objects as related to the memory usage. Because many of the traced applications initially allocate a large chunk of memory, there are several cases that the initial allocations have already exceeded

Table 12
Object information for Bench++ (Dhrystone)

Size range (bytes)	Size distribution (%)	Average life-span (# of allocations)	Sources
16–31	99.99	2.49	Copy constructor
1000+	0.00	1,092,206	–

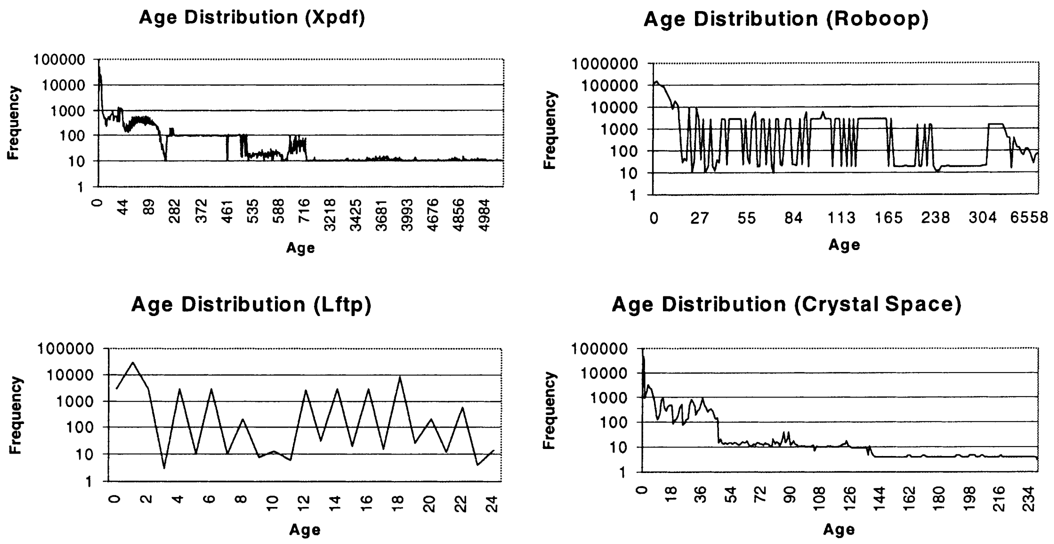

Fig. 4. Age distribution of traced applications.

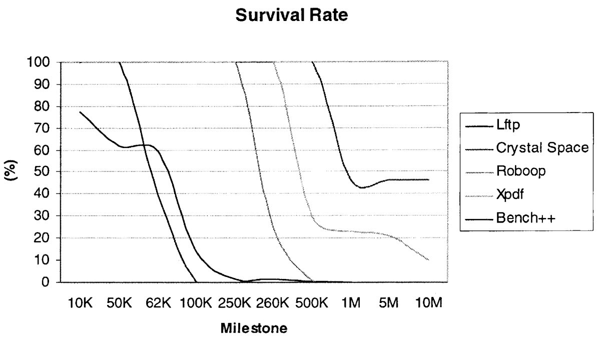

Fig. 5. Survival rate.

smaller milestone (e.g., in Crystal space, the first allocation is already over 500 Kbytes). Thus, the survival rate of such programs is 100% below 500 Kbytes (only Lftp has the initial allocation of less than 32 Kbytes).

6. Conclusions

Dynamic memory management has been a high cost component in many software systems. This paper presents a study about dynamic memory allocation behavior in C++ programs. The hypothesis of situations that invoke the dynamic memory management explicitly and implicitly is presented. They are: constructors, copy constructors, overloading assignment operator =, type conversions and application-specific member functions. A source code level tracing tool is developed to investigate the hypothesis.

The tracing results indicate that many of the DMA are accounted to the object-oriented programming style in C++. Especially, constructors and copy constructors invoke DMA the most. It is worth noting that copy constructors are invoked implicitly in C++ programs. This means that inexperienced programmers may not even be aware of the invocation of copy constructor as they are coding the C++ programs. Moreover, copy constructors may be invoked in several ways such as, passing a class object to a function through call-by-value, returning a class object from a function through return-by-value and even initializing a class object. Awareness of the cost of dynamic memory management and the programming styles that may invoke DMA is crucial to the coding of an efficient C++ program. A software tool that reports the location, the path and the depth of each DMA (Lee et al., 2000a) can assist programmers to manage their C++ programs.

The total percentage of the DMA invocation based on the first four hypothesized cases varies from application to application ranging from 70% to 100%. However, there is a consistent allocation pattern for a specific application regardless of its input. With a consistent DMA invocation pattern, a profile-based memory management strategy is feasible and has been implemented successfully (Chang et al., 2000). Additionally, there is also a correlation between allocation frequency and objects' life-span. The experimental results indicates that objects of the same size, which are frequently allocated tend to have short life-span. Based on these insights, many short-lived objects may not need to be put back on the free list after their liberation. Instead, those objects may be immediately reused for upcoming allocation requests with the same size. This leads to the notion of object reuse which may improve the performance of memory allocator (Lee et al., 2000b).

Acknowledgements

This work is partially supported by IIT ERIF grant.

References

Barrett, D., Zorn, B. 1993. Using lifetime predictors to improve memory allocation performance. In: Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation, ACM SIGPLAN Notices, ACM Press, New York, 28(6) 187–196.

Calder, B., Grunwald, D., Zorn, B. 1995. Quantifying behavioral differences between C and C++ programs. Technical Report CU-CS-698–95, Department of Computer Science, University of Colorado, Boulder, CO, January 1995.

Chang, M., Gehringer, E.F., 1996. A high-performance memory allocator for object-oriented systems. IEEE Transactions on Computers, 357–366.

Chang, J.M., Lee, W.H. 1998. A study on memory allocations in C++. In: Proceedings of the 14th International Conference on Advanced Science and Technology (ICAST'98), Naperville, Illinois, 4–5 April, pp. 53–62.

Chang, J.M., Hasan, Y., Lee, W.H., 2000. A high-performance memory allocator for memory intensive applications. In: Proceedings of the Fourth IEEE International Conference on High Performance Computing in Asia-Pacific Region, Beijing, China, 14–17 May.

Detlefs, D., Dosser, A., Zorn, B., 1994. Memory allocation costs in large C and C++ programs. Software – Practice and Experience, 527–542.

Lee, W.H., Chang, J.M., Hasan, Y., 2000a. A dynamic memory measuring tool for C++ programs. In: Proceedings of IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET 2000), Richardson, TX, 24–25 March.

Lee, W.H., Chang, J.M., Hasan, Y. 2000b. Evaluation of a high-performance object reuse dynamic memory allocation policy for C++ programs. In: Proceedings of the Fourth IEEE International Conference on High Performance Computing in Asia-Pacific Region, Beijing, China, 14–17 May.

Neely, M. 1996. An analysis of the effects of memory allocation policy on storage fragmentation, MS Thesis, Department of Computer Science, University of Texas, Austin, TX, pp. 22–32.

Stefanovic, D., Moss, J.E., 1994. Characterization of object behavior in Standard ML of New Jersey. In: Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming. ACM Press, New York.

Shaw, R. 1988. Empirical analysis of a Lisp system, Ph.D. thesis, Stanford University, Technical Report CSL-TR-88–351.

Stroustrup, B., 1997. The C++ Programming Language, 3rd ed. Addison-Wesley, Reading, MA.

Wilson, P.R., Johnston, M.S., Neely, M., Boles, D. 1995. Dynamic storage allocation a survey and critical review, Technical Report, Department of Computer Science, University of Texas, Austin, TX, pp. 5–32.

Zorn, B., Grunwald, D. 1992. Empirical measurements of six allocation intensive C programs. Technical Report CU-CS-604-92, Department of Computer Science, University of Colorado, Boulder, CO.

Ji-en Morris Chang received the B.S. degree in electrical engineering from Tatung Institute of Technology, Taiwan, the M.S. degree in electrical engineering and the Ph.D. degree in computer engineering from North Carolina State University in 1983, 1986 and 1993, respectively. His industrial experience includes positions at Texas Instruments, Taiwan (1983–84), Microelectronics Center of North Carolina, Research Triangle Park, North Carolina (1986–88) and AT&T Bell Laboratories, Allentown, Pennsylvania (1988–90). From 1993 to 1995, he was an Assistant Professor in the Department of Electrical Engineering at Rochester Institute of Technology, Rochester, NY. In 1995, he joined the Department of Computer Science at Illinois Institute of Technology, Chicago, IL, where he is currently an Assistant Professor. In 1999, Dr. Chang received the University Excellence in Teaching Award at IIT. His research interests include computer architecture, object-oriented programming languages, memory management, hardware description languages, and internet architecture. Dr. Chang has been serving on the technical committee of the IEEE International ASIC Conference since 1994. He also served as the Secretary and Treasurer in 1995 and the Vendor Liaison Chair in 1996 for the International ASIC Conference.

Woo Hyong Lee is a Ph.D. candidate in Computer Science Department, Illinois Institute of Technology. He received the bachelor degree in mechanical engineering from Sungkyunkwan University, South Korea and master degree in computer science from Western Illinois University in 1991 and 1995, respectively. Currently, he serves to the Department of Computer Science, Illinois Institute of Technology as a UNIX system administrator. His current research interests are the performance issues of object-oriented programming languages. Studies include source code level tracing of C++ dynamic memory and high-performance dynamic memory management algorithms.

Witawas Srisa-an received the B.S. degree in Science and Technology in Context and M.S. degree in Computer Science from Illinois Institute of Technology. In 1999, he received the Dean's Scolarship to pursue his Ph.D. study and joined the Computer Systems Laboratory at IIT under the advisory of Dr. Morris Chang. He is presently a Ph.D. candidate in the Computer Science Department, Illinois institute of Technology. He is currently an instructor in the Computer Science Department at IIT. He has taught computer organization, advanced computer architecture, and client/server application development. His research interests include computer architecture, object-oriented programming, dynamic memory management and garbage collection, hardware support for garbage collection, Java, and C++ programming languages.