Measuring Dynamic Memory Invocations in Object-oriented Programs

Morris Chang, Woo Hyong Lee and Yusuf Hasan
Dept. of Computer Science
Illinois Institute of Technology
Chicago IL. 60616
e-mail: {chang | leewoo | hasayus} @charlie.iit.edu

Abstract

Dynamic memory management has been a high cost component in many software system. Studies have shown that memory intensive C programs can consume up to 30% of the program runtime in memory allocation and liberation. The OOP language system tends to perform object creation and deletion prolifically. An empirical study shown that C++ programs can have ten 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 dynamic memory invocations in 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 the procedure to investigate the hypothesis. Thirdly, results include behavioral patterns of memory allocations. With these patterns, we may increase the reusability of the resources. For example, a profile-based strategy can be used to improve the performance of dynamic memory management. The C++ programs that were traced include Java compiler, CORBA compliant and visual framework.

Key words: 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 right now between 10 to 20 times slower than C. The growing popularity of C++ in the industry shows that more applications will be developed in this language. To study the performance issues (e.g. the behavior of its dynamic memory allocation and deallocation), we will need scientific analysis (e.g. quantitative analysis based on some new tracing techniques.). The results of such research can be used in three ways. Firstly, a programmer's guideline can be derived from the research results. At least, then, we know why (or what kind of programming styles in) the C++ programs tend to be slow. Secondly, architectural support (e.g instruction set architecture or co-processor) can be developed based on the research results. For instance, hardware-assisted memory management [ChG 96] can be an effective way to facilitate the issue of being memory intensive. Thirdly, software tools for memory leak detection can be developed based on the research results. Moreover, the performance of dynamic memory management can be improved as if the allocation patterns are known.

Dynamic memory management (DMM) has been a high cost component in many software systems [WJN 95, Nee 96]. A study has shown that memory intensive C programs can consume up to 30% of the program run time in memory allocation and liberation [ZoG 92]. The object-oriented programming (OOP) language system tends to have object creation and deletion prolifically. In [DDZ 94, CGZ95], the empirical data shows 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 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 application or C++ language. For exam-

0-7803-5258-0/99 $10.00 © 1999 IEEE

ple, 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 trace 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 statistics. The C++ programs we traced in our experiment are publicly available software tools. These tools include Java compiler and debugger, Common Object Request Broker Architecture (CORBA) and Venus.

The remainder 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. The last section 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 [Str 97]. 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 built-in type. Such conversion may also invoke new operators. All these four cases are unique to C++ language.

In contrast to the first four cases, the user defined member functions of our fifth case are related to the nature of the application under development. Moreover, the DMM invocations are quite straightforward. Mostlikely, new operators are invoked directly in the program. The following subsections details each case with simple examples.

2.1 Constructors

The constructor which uses *new* operators can be invoked explicitly or implicitly. Considering a String class described as following:

```
#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 statements 8. The 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 the 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 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 localString)
{
  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
```

1. The *initialization* should be distinguished from the *assignment* which will invoke overloaded assignment operator (described in next section).

```
    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 or friend) 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 nonstatic 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's 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);
}
```

The statement 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 is extracted from *V* program. 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);
....
 }
```

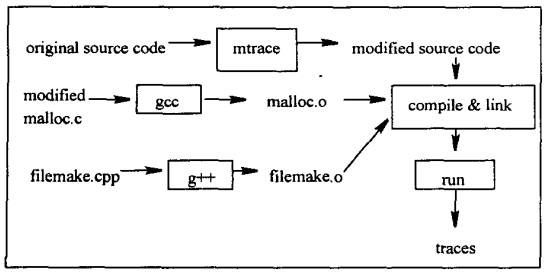
The member functions, *setup* and *open*, invoke *new* operator explicitly. It is easy to see that such DMA invocations are very different from the first four cases mentioned earlier. The fifth case has to do with application only. Thus, we seperate the fifth one from the first four cases.

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 scenario (i.e. constructor, copy constructor, type conversion, and assignment operator= overloading). Each mark contains information of a member function and a class name. 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. The following diagram depicts our tracing experiment.

Figure 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 is presented in the next section.

The C++ programs traced in our experiment are publicly available software applications. These programs include Guavac (Java compiler and debugger), OB (CORBA compliant), V (X windows Framework) and Venus which are C++ library that provides a semi-graphical interface like the Borland's TurboVision framework. These are widely used software. A brief information of these programs is summarized in the following table. They represent several different applications. *Guavac* is a CPU intensive application with no screen interaction. *OB* involves screen interactions but with text only inputs. *Venus* and *V* both are X window applications and have screen interaction through pull down menu.

Table 1: traced programs

program name	line count	ftp site
Guavac	26481	ftp.summit.stanford.edu
OB	78556	ftp.ooc.com/pub/OmniBroker/2.0
Venus	9561	ftp.sunsite.unc.edu /pub/Linux/devel/lang/c++
V	26356	ftp.cs.unm.edu/pub/wampler

4. Results

As we started the project, we have collected many programs which claimed to be C++ programs from the internet. However, we soon found that many of these C++ programs did not utilize object-oriented programming 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. Therefore, these C++ programs were excluded from our experiment.

We have gathered several C++ programs: *Guavac, OB, Venus* and *V*. Guavac is a portable Java compiler which can be freely used and distributed under the terms of the GNU Public License. OB (OmniBroker) is an Object Request Broker (ORB) that is complaint to the CORBA (Common Object Request Broker Architecture). OB allows distributed object oriented computing. OmniBroker provides the source code for non-commercial uses. Venus and V are C++ GUI Framework. These programs represent recent advances in OOP and are coded by professionals. Even though there are only four major programs that were used in this experiment, each one is studied thoroughly in the experiment. More C++ applications are to be collected and investigated in the near future.

For Guavac which has no screen interaction, five hypothesized cases are collected and presented for the entire session. However, for the programs that have screen interactions, data are collected and presented for each interaction session. This allows us to observe the allocation pattern as the application is running. Moreover, we separate fifth case from the rest of four case, so that we can observe the total percentage of the first four cases.

4.1 Guavac

Guavac is a portable compiler for Java. It was written in C++ and can be compiled to Unix machines. It generates correct bytecodes for valid Java program. There are two parts in the guavac version 0.2.5 which were traced in this study. One of them is 'guavac' which is a java compiler and the other is 'guavad' which is used to debug the source code.

Three java programs are used as input to the java compiler. These java programs are 'calc.java', 'clock.java', and 'tictactoe.java'. They are relative short java programs ranging from 150 lines to 370 lines java code. However, the java compiler (guavac), as we expected, responds to all three java programs in a similar way. The tracing results of guavac and guavad are given in next two tables.

The number of the DMA (Dynamic Memory Allocation) invocation corresponding to each hypothesized allocation cases in the guavac program is summarized in Table 2:. The percentage is given in the parenthesis. The Table 3: presents the results from the traces of the guavad program

Table 2: Number (percentage) of the DMA invocation in guavac

	calc.java	clock2.java	tictactoe.java
constructor	1727 (7.01%)	1536 (8.57%)	1367 (8.61%)
copy constructor	4804 (19.5%)	3287 (18.34%)	2924 (18.41%)
overloading =	20 (0.08%)	9 (0.05%)	26 (0.16%)
type conversion	0 (0%)	0 (0%)	0 (0%)
application specific member functions	18068 (73.35%)	13076 (72.97%)	11554 (72.74%)
others	12 (0.05%)	12 (0.07%)	12 (0.08%)
Total	24631 (100%)	17920 (100%)	15883 (100%)

The first four cases which are unique to C++ show similar patterns for each input. Among these four cases, copy constructors are invoked most frequently (about 20% of total DMA). It indicates that many of dynamic memory allocation cases in guavac and guavad are associated with *deep copy*. The fifth case, application specific member functions, is referred to the scenario that memory invocations are initiated in application related member functions. This case represents about 70% of total DMA in both guavac and guavad.

Table 3: Number (percentage) of the DMA invocation in guavad

	calc.java	clock2.java	tictactoe.java
constructor	156 (11.52%)	75 (10.59%)	78 (10.34%)
copy constructor	239 (17.65%)	139 (19.63%)	154 (20.42%)
overloading =	0 (0%)	0 (0%)	0 (0%)
type conversion	0 (0%)	0 (0%)	0 (0%)
application specific member functions	959 (70.83%)	494 (69.77%)	522 (69.23%)
others	0 (0%)	0 (0%)	0 (0%)
Total	1354 (100%)	708 (100%)	754 (100%)

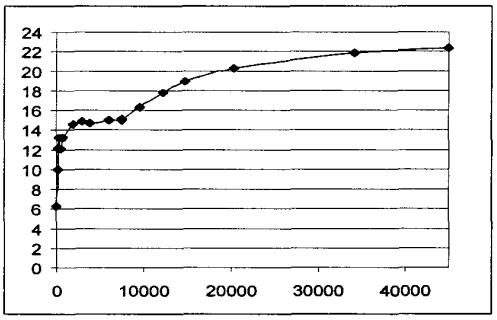
From the above table, the allocation behavior in guavad is quite similar to the one in guavac. However, *guavac* has few allocations are from main function (i.e. in the "others" category). *guavad* has all the memory invocations are conducted inside class member functions. The scenarios of overloading assignment operator= and type conversion are very rare in both programs. Even though, the input java source codes are different sizes, it shows almost same percentages of invocation scenarios for each input. After implementing those two programs, we can be sure that there are some patterns of dynamic allocations during program execution.

4.2 OB

The OB (OmniBroker) is structured to allow integration of a wide variety of object systems. Some highlights of OB are complete CORBA IDL-to-C++ mapping, complete CORBA IDL-to-Java mapping, Dynamic Invocation Interface, Dynamic Skeleton Interface, Interface Repository, Peer-to-Peer communication with nested method invocations, IDL-to HTML translator and DynAny API for Any type handling [Cor 98].

In this section, we analyze the traces from an interactive *client-server* system which is generated by OB. There are sixteen sessions recorded in the experiment. These screen interaction sessions include, for example, new client login, client logout, transmitting messages, or executing some commands. For each interactive session, traces are collected incrementally. The next figure shows the allocation behavior from these sixteen sessions. The x-axis shows the size of accumulated dynamic memory allocation in bytes. The y-axis indicates the total percentage of the DMA invocation that are from the first four hypothesized cases. This way, the DMA invocation that are unique to C++ can be observed.

Figure 2. Allocation pattern in OB.



In the above figure, the total percentage of the DMA invocation based on the first four hypothesized cases increase from 6.25% to 22.3% (from session one to session sixteen). At the first session, there is no client login to the system. After the first session, there are clients logging in and logging out from the system periodically. As these sessions move forward, we can see higher percentage of the C++ related DMA invocation. This trend eventually reaches to a steady state (i.e. in terms of percentage). The x-axis shows the size of the total heap memory allocated for each session.

4.3 Venus & V

Venus and V are consisted of C++ library that provides a semi-graphical interface X windows framework. The library is written to allow users to build programs with a good interface in little time. Venus and V are not commercial products and are distributed under the GNU General Public License.

Similar to the OB, each interactive session is traced for both Venus and V. In the Figure 3 and Figure 4, the total percentage of the DMA invocation based on the first four hypothesized cases (i.e. related to C++) are measured during the entire period of the program execution. The x-axis shows the total number of memory invocations (in terms of times). The y-axis indicates the total percentage

of the DMA invocation that are from the first four hypothesized cases.

Venus traces are collected in 13 sessions and V traces are collected in 5 sessions. Each session includes accumulated data from the previous sessions. The total accumulated bytes during execution for Venus and V are 4800 bytes and 10000 bytes, respectively. From Figure 3 and Figure 4, Venus and V have very different allocation behavior. At the beginning session of Venus execution, all the memory allocations are placed into our C++ related scenarios. However, the percentage decreases from 100% to 58% when the program is finishing. In contrast, V shows a reversed pattern which increases from 37.33% to 68.02%. Both Venus and V are stabilized after several time frames. This demonstrates that each application does have its own DMA pattern. This pattern continues during execution in spite of the changes from the input.

Figure 3 Allocation Pattern in Venus

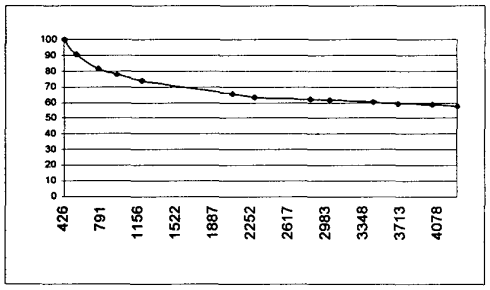
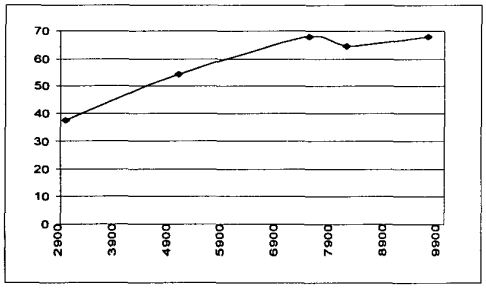


Figure 4 Allocation Pattern in V



5. 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 C++ programs that were traced include Java compiler, CORBA compliant and visual frameworks. These programs are recent developments in OOP and represent applications from no screen interaction to GUI (Graphical User Interface).

The tracing results indicate that many of the DMA are accounted to the object-oriented programming style in C++. Especially, 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.

The total percentage of the DMA invocation based on the first four hypothesized cases varies from application to application ranging from 27% 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 [CHL 98].

6. References

[CGZ 95] 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.

[ChG 96] M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Transactions on Computers*. March, 1996. pp. 357-366

[CHL 98] M. Chang, Y. Hasan and W. Lee, "A profie-based strategy for Dynamic Memory Management", IIT tech. report.

[Cor 98] http://www.ooc.com/ob.html

[Nee 96] Michael Neely, *An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation*, MS Thesis, Department of Computer Science, Univ. of Colorado, Boulder, CO, pp. 22-32, May 1996.

[Str 97] Bjarne StrouStrup, *The C++ Programming Language*, Third Edition, Addison-Wesley, 1997.

[WJN 95] Paul R. Wilson, Mark S. Johnston and Michael Neely, and David Boles, *Dynamic Storage Allocation A Survey and Critical Review*, Technical Report, Department of Computer Science, Univ. of Texas, Austin, TX, pp. 5-32, 1995.

[ZoG 92] 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.

[DDZ94] 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.