Unreachable Procedures in Object-Oriented Programming

AMITABH SRIVASTAVA
Digital Equipment Western Research Laboratory

Unreachable procedures are procedures that can never be invoked. Their existence may adversely affect the performance of a program. Unfortunately, their detection requires the entire program to be present. Using a link-time code modification system, we analyze large, linked, program modules of C ++, C, and Fortran. We find that C ++ programs using object-oriented programming style contain a large fraction of unreachable procedure code. In contrast, C and Fortran programs have a low and essentially constant fraction of unreachable code. In this article, we present our analysis of C ++, C, and Fortran programs, and we discuss how object-oriented programming style generates unreachable procedures.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*Abstract data types*; D.3.4 [**Programming Languages**]: Processors—*Code generation*; *code compilers*, *optimization*

General Terms: Languages

Additional Key Words and Phrases: Link-time code modification system

1. INTRODUCTION

Unreachable procedures unnecessarily bloat an executable, making it require more disk space and decreasing its locality, which may affect its cache and paging behavior. However, programmers rarely write procedures with the intention of never using them. One does not expect to find many such procedures in C [Kernighan and Ritchie 1988] and Fortran [ANSI 1966] programs, but if the programming style emphasizes modeling objects and defining behavior rather than writing procedures when needed, then the program may contain unreachable procedures, since all the behavior patterns may not be used. Section 2 of this article discusses how object-oriented style can generate unreachable procedures.

Object-oriented programming systems like Flavors [Weinreb et al. 1983], LOOPS [Bobrow and Stefik 1983], SCOOPS [Srivastava 1985; Texas Instruments 1985] are interactive systems built around Lisp [Steel 1984] and Scheme [Rees and Clinger 1986] with dynamic inheritance. A change made to classes in these systems is propagated throughout the inheritance structure;

Author's address: DECWRL, 250 University Avenue, Palo Alto, CA 94301

thus, at any time methods and functions can be added to the system. In this environment, the notion of unreachable procedures does not make sense.

This article is concerned instead with languages like C ++ [Ellis and Stroustrup 1990; Stroustrup 1991], C, and Fortran, which are usually *statically linked.* A program build usually ends with a *link phase* in which separately compiled files are combined together into a single executable. During the link phase all procedures in an object module are included if any one of them is referenced. To minimize unreachable procedures, library designers have traditionally split files into many smaller files each containing few procedures. Splitting a file is not always possible, since it destroys the organizational structure of programs and is not a suitable solution for programs written in object-oriented style. Section 5 discusses these issues.

Dead-code elimination [Aho et al. 1988] is a standard compile-time optimization that eliminates useless code in the program: code that is never reached or code that computes a value that is never used in the program. Unreachable procedures are also dead code. Unfortunately, their detection requires the entire program to be present. Compilers cannot determine if a global procedure is unreachable, since most compilers process a single file at a time, and global procedures have scope larger than a file. Unreachable procedures must be marked at link-time when the whole program is available.

We chose C ++, C, and Fortran because they are widely used statically linked languages. C ++ provides common object-oriented features, and C ++ programs using the object-oriented programming style are available. Many C and Fortran programs that do *not* use the object-oriented programming style are available in public domain. Since C ++ evolved from C, they share the same linking and loading mechanisms, and their comparison is of interest.

We discuss how object-oriented programming style can generate unreachable procedures and describe the method for their detection. We present the results of our analysis of C ++, C, and Fortran programs by our link-time code modification system OM [Srivastava and Wall 1993]. We also argue that library splitting is not a desirable solution.

2. UNREACHABLE PROCEDURES—WHY WRITE THEM?

Three important sources of unreachable procedures are object-oriented programming style, library structure, and debugging methodology.

2.1 Object-Oriented Programming

In object-oriented programming the structure of a program parallels that of the system being modeled. The emphasis is on the properties and behavior of objects rather than internal implementation details. Some aspects of this style can easily produce unreachable procedures.

Class design. The system being modeled consists of various entities that interact with one another. A class in the program represents an entity in the system. A class definition specifies the behavior of its objects, that is, how they interact with other objects and how they can be queried and modified.

The object-oriented style focuses on an object's properties and behavior. This makes programs easier to understand, modify, and maintain. The class designer keeps the internal details local to the class and provides interface routines for the rest of the system. Thus, *all* possible interfaces and manipulations for the class are defined. However, other objects may use only a *few* of the defined interfaces. The unused interface routines are unreachable procedures.

Consider an example of a class definition modeling a queue. The internal data structure is kept local to the class while external interfaces addQueue, getQueue, deleteQueue, and printQueue are defined. If the internal representation is changed, only the class QUEUE needs to be modified; the change should not be visible to the rest of the program. If the program uses only getQueue and putQueue, then deleteQueue and printQueue will be unreachable.

```
class QUEUE{
private:
  int *queueArr;
public:
  int getQueue( ); // get next element from queue
  void putQueue(int); // add element to queue
  void deleteQueue( ); // delete element from queue
  void printQueue( ); // print elements in queue
};
```

Inheritance. A powerful mechanism of object-oriented programming is inheritance. Higher levels of abstractions are built through inheritance. Specifying large systems through an inheritance structure of classes results in a modular design and avoids specifying redundant information. A derived class is defined by inheriting a base class, adding and redefining class variables and procedures. The derived class uses the information available in the base class. A program might not use procedures that are *hidden* in the inheriting process. The longer the inheritance chain, the higher is the probability of producing hidden unreachable procedures.

```
class PERSON{
  public:
    char *name;
    void printName( );
    void printInfo( );
};
```

```
class PILOT : public PERSON{
  public:
    void printInfo( );
};
```

The class PILOT inherits class PERSON and redefines the method printInfo. It is possible that printInfo in PERSON is not used in the program.

Virtual functions. Virtual functions permit polymorphism; they are used to create the most general definition of a certain concept in a base class. Derived classes inheriting this base class may refine this definition. Depending on the type of object, the correct definition of the concept is invoked. As with inheritance, the original definitions in the base class might not be used.

2.2 Design of Libraries

In design of system libraries for languages like C and Fortran, commonly used procedures are defined for certain fundamental data types such as strings and integers. For example, the string library includes procedures for copying, comparing, and searching strings. Similar to defining a class interface in object-oriented programming, libraries can generate unreachable procedures if few of the defined operations are used. Library designers have used the trick of splitting packages into microfiles to minimize unreachable code. Section 5 discusses the issues in detail and explains why splitting is not an acceptable solution.

2.3 Debugging

Program designers often write code that is useful for them in program development. The debugging routines print intermediate information during program execution. These routines may also be invoked when the program has paused under debugger control because of a breakpoint. In a released program, debugging routines are never called and thus are unreachable.

3. DETECTION OF UNREACHABLE PROCEDURES

We looked for unreachable procedures in C ++, C, and Fortran programs, using our link-time code modification system OM [Srivastava and Wall 1993]. OM analyzes a complete program in the form of a collection of object files and libraries. It can summarize, optimize, or instrument the program based on this analysis.

Unreachable procedures are detected by building a directed call graph. Nodes in the call graph are procedures; edges are statically present procedure calls. We construct the set ADDRPROCS that contains all procedures whose addresses are taken. Procedures in ADDRPROCS might be reached dynamically via indirect calls, which are not present in a static call graph. So we build the set ROOTS that contains the start procedure and the procedures in ADDRPROCS. Using the call graph, a standard algorithm finds the procedures reachable from the set ROOTS; the rest are unreachable. This algorithm is conservative and uses only the static information, it cannot detect procedures that are dynamically unreachable.

Virtual-function call in C ++ is a dynamic invocation. The algorithm discussed above marks all virtual functions reachable[1] even though some of them may be dynamically unreachable. However, if we understand the way virtual functions are constructed, we can determine whether the virtual functions of a class, whose objects are never constructed, are unreachable.

We refine our algorithm in the following way to detect unreachable virtual functions of a class that is never instantiated. We do not include a vir-

[1] Implementation of virtual functions requires their addresses to be stored in a table which the constructor stores in the object. The algorithm adds all virtual functions to ADDRPROCS.

tual function as a member of ADDRPROCS,[2] and to compensate for this we add edges to the call graph from the constructor of a class to each virtual function of the class that is ever referenced. Thus we pretend that virtual functions are invoked from their constructors. As before, we build the set ROOTS from the start procedure and the set ADDRPROCS. Using the modified call graph, the standard algorithm finds the procedures reachable from the set ROOTS; the rest are unreachable.

4. CODE ANALYSIS OF PROGRAMS

Selecting Programs

For measuring unreachable code, we chose programs that were large and were written for serious applications. Small programs often give misleading results, generally a larger proportion of unreachable code. We selected C and Fortran programs from the SPEC suite, two graphical C ++ programs from the Interviews suite, and five computational C ++ programs that are CAD/CAM tools in use at WRL. Program descriptions are given in Figure 1.

Programs were compiled and linked on a DECStation[3] running under Ultrix[3] using the AT & T C ++ translator, the DEC C ++ compiler, and host C and Fortran compilers. As system libraries may be dynamically loaded, we measure the unreachable code in the programs both with and without system libraries linked in. The *unreachable-code percentage* is the quotient of the sizes of the unreachable procedures and the total size of all procedures.

Programs without System Libraries

We first measure the unreachable code in user programs *without* C, Fortran, and C ++ system libraries. Figure 2 shows the results. The C and Fortran programs have 0–5% unreachable code. Their programming style involves writing a procedure only if it is needed. C ++ programs using object-oriented programming style have up to 26% unreachable code.

Programs with System Libraries

We study the effects of system libraries by measuring the unreachable code in programs with C ++, C, and Fortran system libraries linked in. Figure 3 presents the fraction of unreachable code in the same format as Figure 2, while Figure 4 presents the fraction of unreachable code as a function of the total amount of code in the programs. The graph in Figure 4 highlights the difference between C ++ and C/Fortran programs. Unreachable code in C ++ program *is* consistently higher than C and Fortran programs at all values of total code. The unreachable code proportion decreases slightly at

[2] We ignore the fact that the address of a virtual function appears in a table; however, if the address of a virtual function is explicitly taken it will be added to ADDRPROCS.
[3] Ultrix and DECStation are trademarks of Digital Equipment Corporation.

program	*description*	*language*
bisim	transistor-level simulator	C++
bitv	bipolar timing verifier	C++
dclock	scalable digital clock	C++
drip	vlsi interpreter	C++
iclass	class browser	C++
layout	vlsi layout program	C++
lsys	generate complex models	C++
usf	ultra-fast simulator	C++
espresso	set operation benchmark	C
eqntott	truth table generator	C
li	lisp interpreter	C
gcc1	gnu C compiler	C
gdb	debugger	C
doduc	hydrocode simulation	Fortran
fpppp	quantum chemistry	Fortran
spice	circuit simulation	Fortran

Fig. 1. Program descriptions.

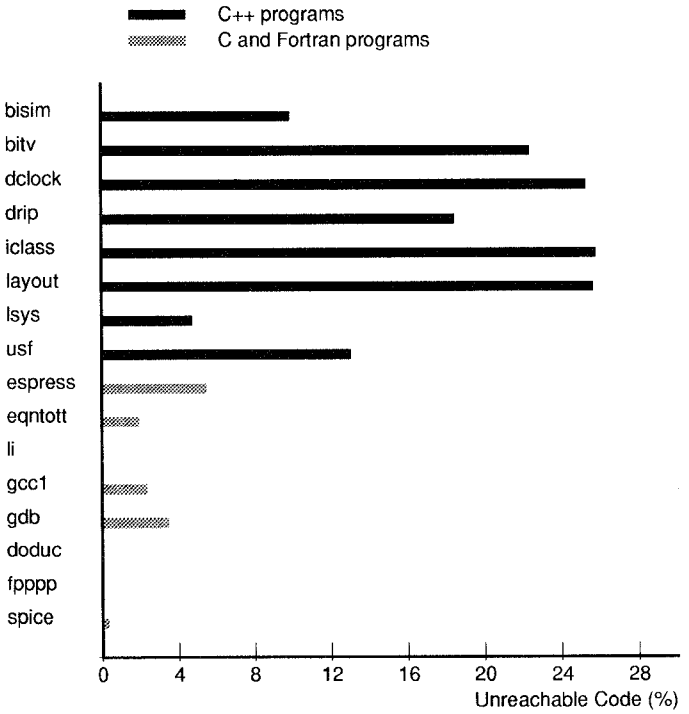

Fig. 2. Unreachable code in user's program.

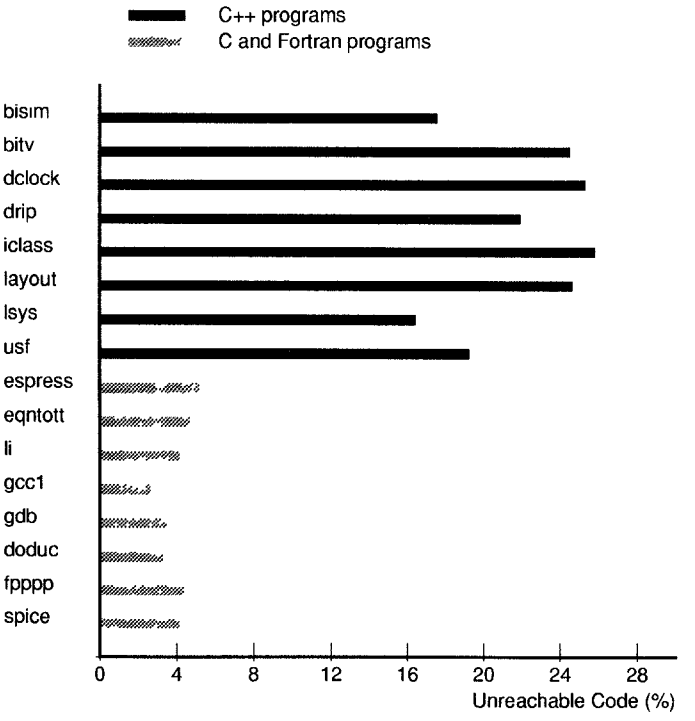

Fig. 3. Unreachable code with total procedure code.

large code sizes in C and Fortran programs while it increases in C++ programs.

Unreachable-Procedure Analysis

We further analyze the unreachable procedures that we found in programs by dividing the unreachable procedures into two groups. The first group consists of unreachable procedures that are never called, that is, unreachable procedures that have no predecessors. The other group consists of unreachable procedures that have predecessors but for which no predecessor is reachable from the program.

The two groups are of interest because unreachable procedures with no predecessors can be easily marked with a simple algorithm since they are never referenced, while detection of unreachable procedures with predecessors requires the algorithm outlined in Section 3. The fraction of unreachable procedures that have no predecessors ranges in C++ programs from 45% to 82% with an average of 64%, and in C and Fortran program from 32% to 82% with an average of 57%. Since there are a substantial number of unreachable procedures with predecessors, the algorithm discussed in Section 3 should be used.

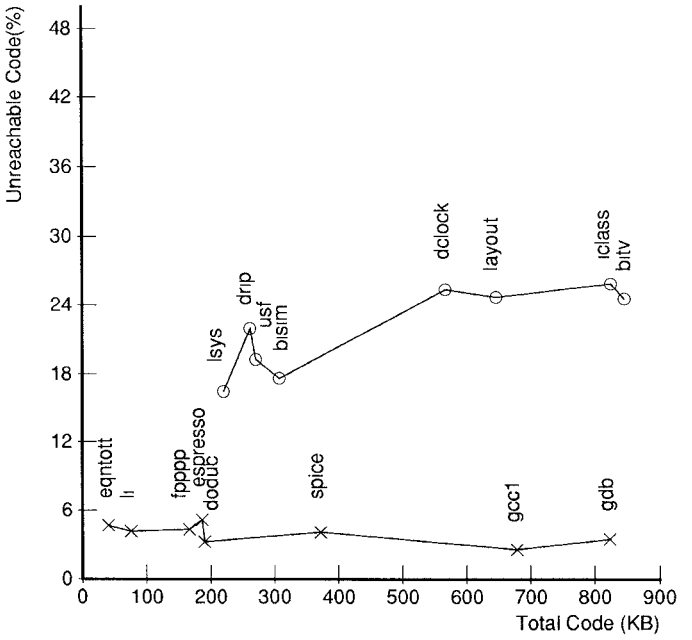

Fig. 4. Unreachable code with total procedure code.

5. LIBRARIES — IS FILE SPLITTING THE ANSWER?

A library is a collection of object modules; each object module contains one or more procedures. During linking, if a procedure in an object model is referenced, the rest of the procedures in that object module are also included. The traditional solution is to split the file into smaller files, each containing a single procedure.

Splitting library files prevents unnecessary library routines from tagging along with necessary ones. But unreachable user routines will still cause unnecessary library routines to be included, which in turn may pull in still more.

Besides being inconvenient, splitting a file may not always be possible. For example, a file may contain two procedures sharing global but unexported variables or procedures (static variables and procedures in C). If such procedures are split into two files, the shared variables and procedures would have to be exported to the whole program.

In C and Fortran this problem is usually limited to system libraries. However, any system designed in object-oriented style is generally written like a library. The system designer has two options, either split the files or structure the program as needed and risk having large amounts of unreachable code. For example, the libraries in the X Window system have been carefully structured to minimize procedures per file. Various schemes for managing C ++ libraries [Coggins and Bollella 1989] have been sugges-

ted. Most involve writing one procedure per file and present unnecessary complications for the library implementor.

The Eiffel [Myer 1992] compiler from Interactive Software Engineering also attempts to minimize unreachable procedures. Eiffel code is first converted to an intermediate form and then compiled to C. The final executable is generated by compiling the equivalent C and linking in the system libraries and code from other languages present in object form. The compiler *can* remove unreachable procedures in Eiffel code by compiling all Eiffel code to its intermediate form and generating C code only for those routines that may be needed, but *cannot* remove unreachable procedures in system libraries and routines from other languages that are present in object form and are linked in by the system linker in a later phase.

The correct solution, in our opinion, is to have a link-time option to process the program and remove all unreachable procedures. Since the languages we are concerned with usually end with a link phase, the whole program including system libraries and modules from other languages are present in this phase in the same object module format. The link-time option allows programmers to keep structure in their programs without incurring any penalty. When programming in a higher level of abstraction one should not have to worry about low-level details or be forced to modify the structure of programs to suit them.

6. CONCLUSION

Object-oriented programming style produces substantially more unreachable procedure code than other programming styles. Unfortunately, most existing systems do not remove unreachable code at link-time. This is historically understandable as we found only 0–5% unreachable code in C and Fortran. In contrast, our analysis also found up to 26% unreachable code in C ++ programs. Since C ++ enables the easy design of large applications, this seems more than enough to have noticeable effects on disk utilization and program locality.

ACKNOWLEDGMENTS

I had many discussions with David Wall, and I am grateful to him for his suggestions and perceptive comments. Shell Simpson provided me with Interviews built with DEC C ++. Jeremy Dion, Mary Jo Doherty, Ramsey Haddad, Richard Swan, and David Wall gave comments on earlier drafts of the article. I thank them all for their valuable help.

REFERENCES

AMERICAN NATIONAL STANDARD INSTITUTE. 1966. *ANSI Standard Fortran.* ANSI, New York.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1988. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass.

BOBROW, D. G., AND STEFIK, M. J. 1983. *The LOOPS Manual.* Xerox Corporation, Palo Alto, Calif.

COGGINS, J., AND BOLLELLA, G. 1989. Managing C ++ libraries. *SIGPLAN Not. 24*, 6 (June).

ELLIS, M., AND STROUSTRUP, B. 1990. *The Annotated C ++ Reference Manual.* Addison-Wesley, Reading, Mass.

KERNIGHAN, B. W., AND RITCHIE, D. M. 1988. *The C Programming Language.* Prentice-Hall Software Series, Englewood Cliffs, N.J.

MEYER, B. 1992. *Eiffel: The Language.* Prentice-Hall, Englewood, Cliffs, N.J.

REES, J., AND CLINGER, W., EDS. 1986. Revised report on algorithmic language scheme. *SIGPLAN Not. 21,* 12, 37–39.

STEEL, G. L. 1984 *Common Lisp: The Language.* Digital Press.

SRIVASTAVA, A. 1985. SCOOPS: Scheme object-oriented programming system. Rep. CSL-23, Texas Instruments, Dallas, Tex.

SRIVASTAVA, A., AND WALL, D. W. 1993. A practical system for intermodule code optimization at link-time. *J. Program. Lang. 1* (Mar.), 1–18. Also available as WRL Research Report 92/6, December 1992.

STROUSTRUP, B. 1991. *The C ++ Programming Language.* Addison-Wesley, Reading, Mass.

TEXAS INSTRUMENTS. 1985. *TI Scheme Language Reference Manual.* Texas Instruments, Austin, Tex.

WEINREB, D., MOON, D., AND STALLMAN, R. 1983. *Lisp Machine Manual.* MIT, Cambridge, Mass.

Received February 1993; revised July 1993; accepted July 1993