AUGUST 1993

WRL Research Report 93/4

Unreachable Procedures in Object-oriented Programming

Amitabh Srivastava

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There two other research laboratories located in Palo Alto, the Network Systems Laboratory (NSL) and the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Unreachable Procedures in Object-oriented Programing

Amitabh Srivastava

August 1993

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Abstract

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 paper, we present our analysis of C++, C and Fortran programs, and discuss how object-oriented programming style generates unreachable procedures.

This paper will appear in the *ACM LOPLAS Vol 1, #4.*. It replaces Technical Note TN-21, an earlier version of the same material.

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 [6] and Fortran [1] programs, but if the programming style emphasizes modeling objects and defining behavior rather than writing procedures when needed, the program may contain unreachable procedures, as all the behavior patterns may not be used. Section 2 of this paper discusses how object-oriented style can generate unreachable procedures.

Object-oriented programming systems like Flavors [14], LOOPS [3], SCOOPS [10, 13] are interactive systems built around Lisp [9] and Scheme [8] with dynamic inheritance. A change made to classes in these systems is propagated throughout the inheritance structure; 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 paper is concerned instead with languages like C++ [5, 12], 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 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, as 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 [2] 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 as 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.

In this paper, 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[11]. 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 other. 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 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 micro-files 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 [11]. 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 virtual 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.

[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.

[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.

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

Figure 1: Program descriptions

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.

[3]Ultrix and DECStation are trademarks of Digital Equipment Corporation

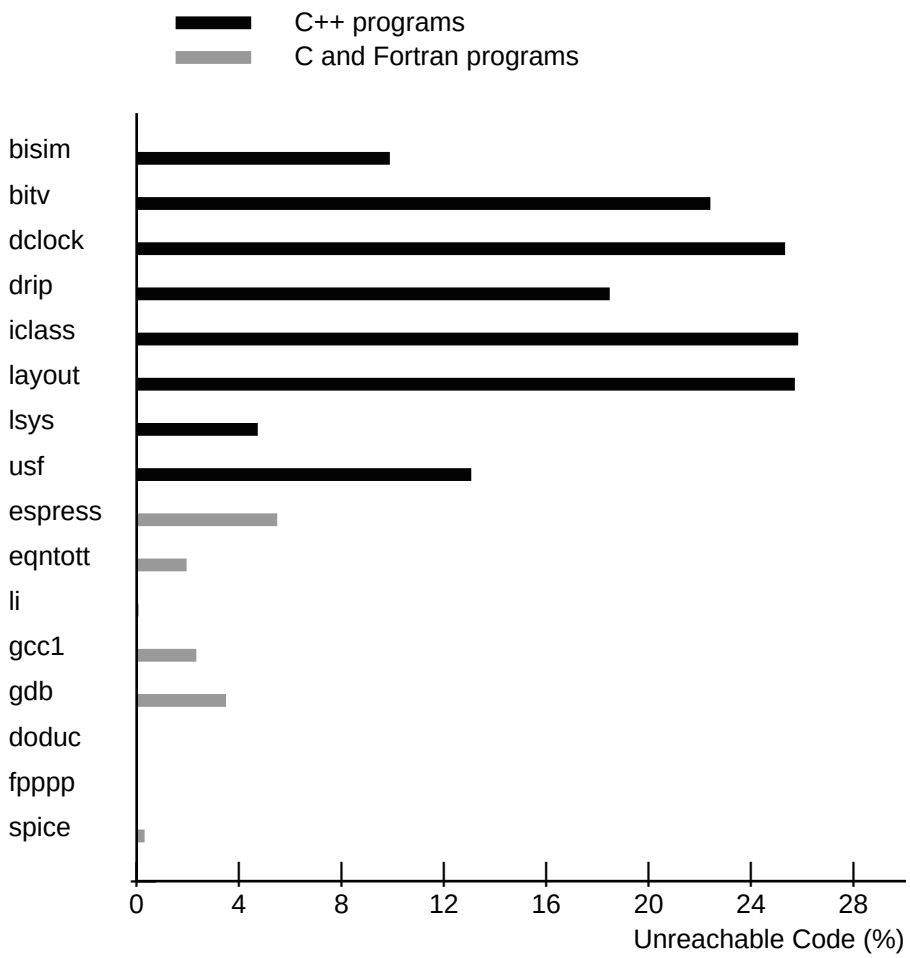

Figure 2: Unreachable Code in User's Program

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.

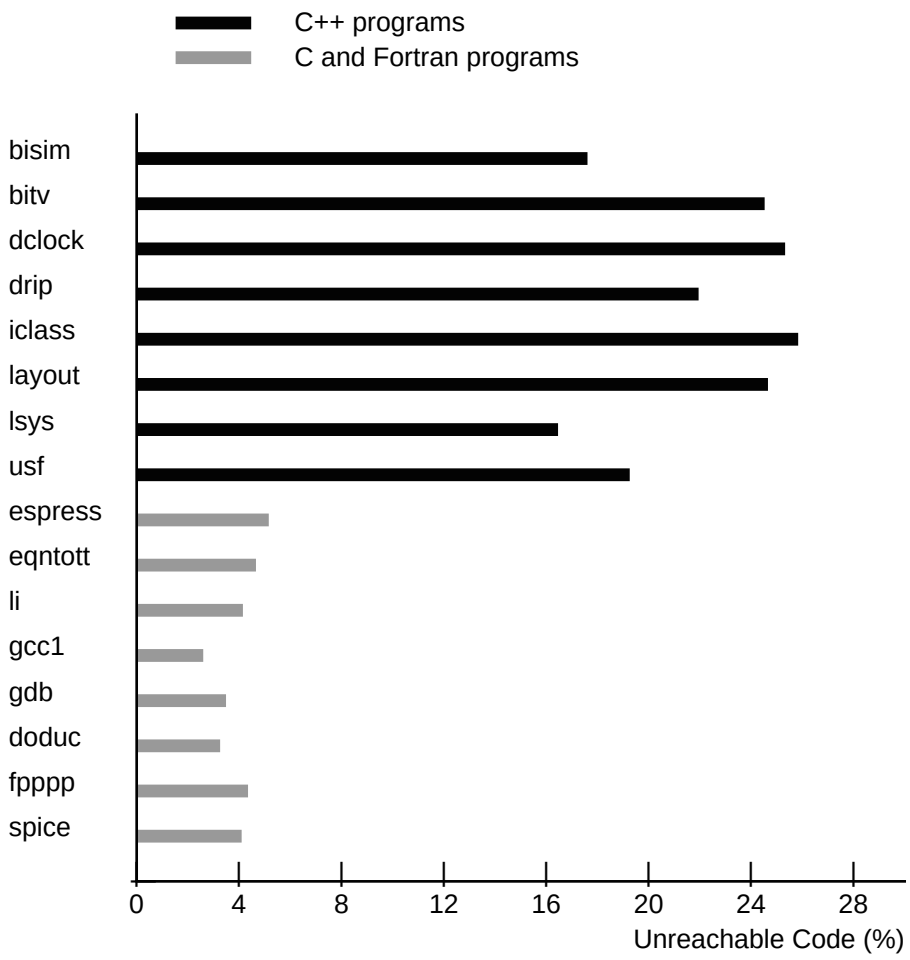

Figure 3: Unreachable Code with Total Procedure Code

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 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++ programs is consistently higher than C and Fortran programs at all values of total code. The unreachable code proportion decreases slightly at large code sizes in C and Fortran programs while it increases in C++ programs.

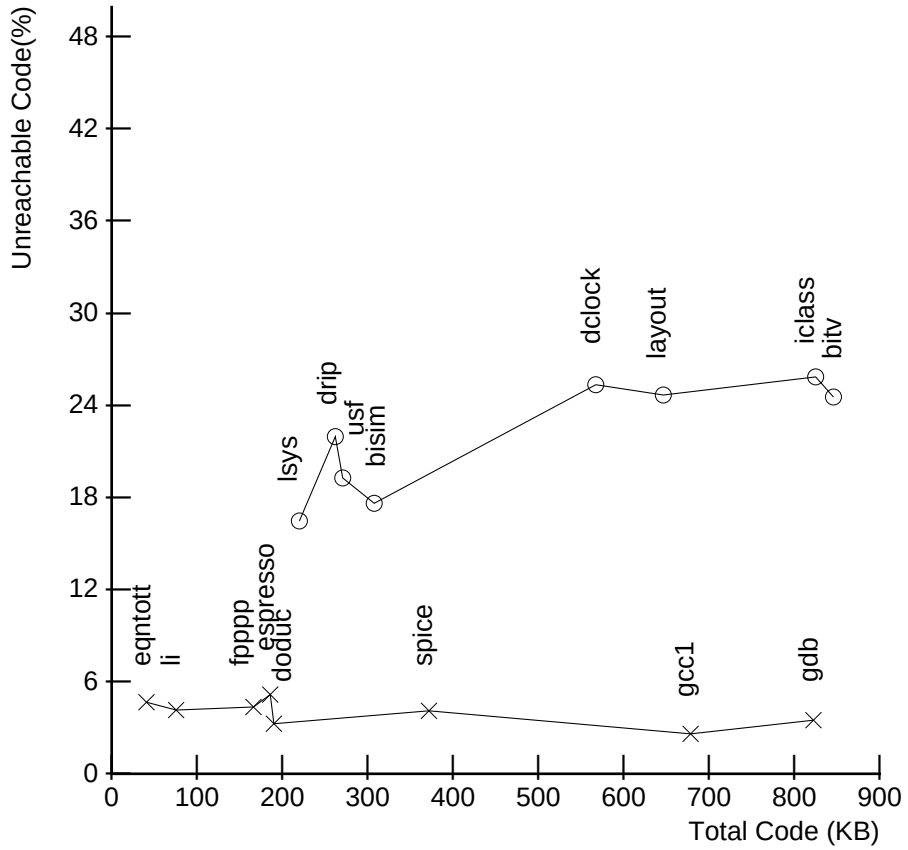

Figure 4: Unreachable Code with Total Procedure Code

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 as 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%. As there are substantial number of unreachable procedures with predecessors, the algorithm discussed in Section 3 should be used.

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 module 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 [4] have been suggested. Most involve writing one procedure per file and present unnecessary

complications for the library implementor.

The Eiffel [7] 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. As C++ enables the easy design of large applications, this seems more than enough to have noticeable effects on disk utilization and program locality.

Acknowledgment

I had many discussions with David Wall. 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 the earlier drafts of the paper. I thank them all for their valuable help.

References

[1] ANSI Standard Fortran, American National Standard Institute, NewYork, 1966.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1988.

[3] D.G. Bobrow and M. J. Stefik, *The LOOPS Manual*, Xerox Corporation, 1983.

[4] James Coggins and Gregory Bollella, *Managing C++ Libraries*, ACM SIGPLAN Notices, June 1989.

[5] Margaret Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[6] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall Software Series, 1988.

[7] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall, 1992.

[8] Jonathan Rees and William Clinger (Editors), *Revised*^{3} *Report on Algorithmic Language Scheme*, SIGPLAN Notices 21(12):37-39, December, 1986.

[9] Guy L. Steele, *Common Lisp : The Language*, Digital Press, 1984.

[10] Amitabh Srivastava, *SCOOPS: Scheme Object-oriented Programming System*, Texas Instruments Report CSL-23, 1985.

[11] Amitabh Srivastava and David W. Wall, *A Practical System for Intermodule Code Optimization at Link-Time*, Journal of Programming Language, vol 1, March 1993, pp 1-18. Also available as WRL Research Report 92/6, December 1992.

[12] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.

[13] Texas Instruments, *TI Scheme Language Reference Manual*, Texas Instruments 1985.

[14] D. Weinreb, D. Moon and R. Stallman, *Lisp Machine Manual*, MIT 1983.

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

''Simple and Flexible Datagram Access Controls for Unix-based Gateways.''
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

''Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.''
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

''Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.''
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

''A Unified Vector/Scalar Floating-Point Architecture.''
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

''Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.''
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

''Integration and Packaging Plateaus of Processor Performance.''
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

''A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.''
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

''The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.''
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

''Long Address Traces from RISC Machines: Generation and Analysis.''
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

''Link-Time Code Modification.''
David W. Wall.
WRL Research Report 89/17, September 1989.

''Noise Issues in the ECL Circuit Family.''
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

''Efficient Generation of Test Patterns Using Boolean Satisfiablilty.''
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

''Two Papers on Test Pattern Generation.''
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

''Virtual Memory vs. The File System.''
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

''Efficient Use of Workstations for Passive Monitoring of Local Area Networks.''
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

''A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.''
John S. Fitch.
WRL Research Report 90/6, July 1990.

''1990 DECWRL/Livermore Magic Release.''
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

''Pool Boiling Enhancement Techniques for Water at Low Pressure.''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

''Writing Fast X Servers for Dumb Color Frame Buffers.''
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’
David Boggs.
WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’
Joel Bartlett.
WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’
G. May Yip.
WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’
David W. Wall.
WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’
Russell Kao.
WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.

‘‘Cache Replacement with Dynamic Exclusion’’
Scott McFarling.
WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach’’
John S. Fitch.
WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter’’
David Boggs.
WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS’’
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package’’
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics’’
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0’’
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture’’
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors’’
Scott McFarling.
WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates’’
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.