A Macroscopic Profile of Program Compilation and Linking

MARK A. LINTON AND RUSSELL W. QUONG

Abstract—**To profile the changes made to programs during development and maintenance, we have instrumented the *make* utility [1] that is used to compile and link programs. With minor modifications, we have used *make* to find out how much time programmers spend waiting for compiling and linking, how many modules are compiled each time a program is linked, and the change in size of the compiled modules.**

Our measurements show that most programs are relinked after only one or two modules are recompiled, and that over 90 percent of all recompilations yield object code that is less than 100 bytes larger in size. We are using these results to guide the design of an incremental programming environment, particularly with respect to an incremental linker.

Index Terms—**Compilation, incremental linking, programming environment, program size distribution, software development profile.**

I. Introduction

INCREASING programmer productivity requires either improving the quality of their programs or reducing the time it takes to produce them. Programmers spend much of their time modifying and then rebuilding programs, particularly during the long and costly period of software maintenance. "Turnaround time" is the time it takes to rebuild a program, namely, the delay after a source code change is made before the object code can be executed; it usually encompasses recompiling, relinking, and starting a debugger. Since turnaround time is a major component of programmer "wait" time, reducing it effectively increases productivity.

Our long-term goal is to understand how to build an environment in which program design time is minimized and in which programmer wait time is eliminated during "edit-compile-execute-debug" iterations. Batch-oriented compilation environments such as UNIX® [3] force programmers to wait for the compiler and linker, while interpretive environments such as Smalltalk [2] sacrifice execution speed and static type-checking for immediate feedback. Our goal is to provide both fast turnaround and fast execution in a programming environment.

Our approach to achieve this end is to use incremental compilation and linking techniques. These techniques attempt to recompile and relink only those parts of the program affected by the latest change. For small changes, incremental compilation and linking can be substantially faster than normal methods.

The effectiveness of these algorithms depends on the type of modifications programmers make today. Knowing quantitative distributions of program size, module size, program turnaround time, and magnitude of object code changes is critical in the design of incremental algorithms and data structures. To gather this information, we have instrumented the UNIX utility **make** and have been collecting data for several months here at Stanford.

In the remainder of this paper, we describe our data collection and analysis methods in detail, present the results of our measurements, and draw conclusions on how the results affect the design of an incremental programming environment. In particular, we describe how these measurements have guided the design of an incremental linker.

II. Approach

We chose to obtain our measurements in the UNIX environment because we are quite familiar with it from everyday usage. Our familiarity has helped us determine a simple, efficient method to collect relevant data from the large community of programmers at Stanford.

We do not claim that our environment is representative of industrial software development; whether observing other environments would yield a similar profile is an open question. However, we do not have a substantial user community of software developers performing both research programming and maintaining production systems. Our results are relevant to ourselves and to other groups conducting similar research. Our approach can be applied to any UNIX environment, and can be adapted to other environments with standard conventions for program compiling and linking.

Our method is novel in that it is *macroscopic*. Unlike system accounting of cumulative compiler usage or the profiling of individual tools, we observe how a program changes over time as it is rebuilt. We were able to gather data easily on UNIX because the **make** utility is normally used to control the compilation and linking of whole programs, allowing us to view the entire process in the context of rebuilding a program and not just individual modules.

Make does not do any compiling or linking itself; in-

Manuscript received January 2, 1987; revised March 27, 1987. This work was supported by the SUNDEC project through a gift from Digital Equipment Corporation.

M. A. Linton is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305.

R. W. Quong is with the Department of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

IEEE Log Number 8826225.

®UNIX is a registered trademark of AT&T Bell Laboratories.

stead, it calls appropriate commands to recompile out-of-date modules based on dependency information supplied by the user. By installing our own version of **make**, we were able to obtain unbiased data automatically and transparently. In addition to recording compilaton and linking data, we record the host machine, user name, and the current file system directory for each **make**. This information identifies the session.

There are a numer of ways of measuring how a program changes: object code size, number of modules, and source code size. Object code size is important for general memory allocation considerations when designing incremental algorithms. More important is how the size of object code changes. This information is helpful in the design of an incremental compiler; it is required for the design of an incremental linker.

The profile of object code size changes is important in understanding how an incremental linker should manage memory. In particular, if large changes happen infrequently the linker can allocate some extra memory for each module and only reallocate all of program memory when a major change is made. Alternatively, if large changes happen frequently, it may be desirable to use indirect addressing for all cross-module references. Ideally, we should also profile how object code addresses change, but this data is too expensive to collect.

The number of modules affects the cost of linking. Increasing the number of modules increases external references and relocation information, which slows down the linker. On UNIX, the smallest unit of compilation is a file, thus in this paper a "module" is a file. However, files are not necessarily logical units; they may comprise several logical modules or be part of a larger logical module.

We did not analyze source code changes because the size of program source is not meaningful, as it is very dependent on programmer style (e.g., use of comments, white space). A minor source code change can generate a substantial object code change. For example, requesting a procedure to be compiled inline might make a module much larger and change the address of every symbol within it. Finally, analyzing source code is difficult and expensive to do.

By collecting data on object code size, number of modules, and turnaround time, we can answer the following questions:

- How many modules are there in the average program?
- How large is the average module?
- How many modules are compiled each time a program is linked?
- How much does a module's object code size change?
- How long does the average program take to compile and link?
- How much time passes between **make** sessions for the same program?
- How do turnaround time, link time, object code size, and number of modules correlate with each other?

The remainder of this section describes the two phases of our experiment in detail. We first describe how we instrumented **make** to gather the desired data, and then how we have condensed the information to an understandable form.

A. Collecting the Data

Instrumenting any program consists of three subproblems: detecting when something of interest occurs, extracting the information of interest, and storing the information. Our **make** records the following commands:

cc	UNIX portable C compiler
pc	Berkeley UNIX Pascal compiler
f77	Fortran 77 compiler
mod	DEC-WRL Modula-2 compiler[4]
CC	C++ compiler[5]
ld	UNIX linker

Whenever **make** calls one of the above commands, we record its name along with the command line arguments, how long the command ran, the name of the expected object file and the size of the object code it generated (text, data, and symbol table). For each **make** session, we also record the machine name, the user name, the full path name of the current working directory, and when **make** started and finished. The following fragment is a sample of the data we collect:

```
#Host=derelict #User=quong
#cwd=/a1/quong/thesis/analyze
CC -g -c table.c
#user=30.800 #sys=3.840
#objname=table.o
#objsize=23160 {3952,1420,863}
CC -o newana analyze.o util.o
   hash.o strtable.o session.o
   table.o error.o -lm
#user=7.770 #sys=5.690
#objname=newana
#objsize=95232 {22528,5120,3344}
#Total_user=38.570 #Total_sys=9.530
#Start=Wed Sep 10 16:18:31 1986
#Finish=Wed Sep 10 16:20:07 1986
```

B. Storing the Data

There are a number of concerns in storing the collected information. Many users on different machines frequently run **make**, so write access to the accumulated data must be synchronized. Since we installed our **make** for all users, it had to be indistinguishable from the original **make**. Our data collection process could not slow **make** down, nor could we change the functionality of **make** in any way. For these reasons, and to simplify our implementation, our **make** mails the results of each session to a central mailbox using the standard UNIX mail system. To avoid waiting for the data to be queued in the mail system, **make** spawns a background process which sends the mail.

Using the mail system to store data solved the following data collection problems.

Central Data File: The UNIX mail system allows mail to be forwarded to a file; thus we could collect all our data in a single file.

Synchronization: Mail systems already synchronize access to mailboxes.

Distributed Access: Our **make** sends data to a mailbox on a specific host; if **make** runs on a different host then the mail system sends the data directly to the desired mailbox.

Fault Tolerance: If a host is down or temporarily unreachable, the mail system queues the data and tries to resend it later.

We spent two man weeks instrumenting **make** and added about 430 new lines of code, almost all of which formed a new module. The changes to the original **make** code simply call routines in the new module before and after **make** issues a command.

C. Reducing the Data

To process the raw data from the **make** sessions, we wrote an analyzer program to extract and accumulate the information of interest. Given a **make** session, the first step is to identify which program is being made. The analyzer program uses only the full path name of the current working directory to identify distinct target programs. We do not use the hostname because our hardware configuration includes several MicroVAXes sharing a common file server. Thus, a program can be made on any of a number of machines. We do not use the user name for identification since several people may be working on the same program. Our only assumption is that there is one program per directory. This identification scheme caused no problems.

In each **make** session, the analyzer examines each command and its arguments. It is impossible to distinguish between a compile and link solely by the command because all the UNIX compilers can also link (e.g., "cc a.o b.o" is a link). Since all the compilers use the "-c" flag to specify that the link phase should not be performed, the analyzer considers a command a compile if "-c" is present.

If "-c" is not present, the command is considered a link if none of the arguments is a source file. If the command contains some object files and some source files (e.g., "cc a.c b.o"), or all source files (e.g., "cc a.c b.c"), then it is both a compile and a link. The analyzer discards such commands because it cannot break down the time for the command into a compile and link time.

The analyzer computes the number of modules linked by counting the number of files in the link command. The analyzer ignores all command arguments beginning with a "-", since this is the UNIX convention for a command line flag. The number of modules actually processed during linking is usually higher than the measured number because modules from libraries are not counted.

The analyzer saves the most recent text and data size of each program and module. This data is necessary to determine how much object code changes over time. Whenever a module is recompiled or a program is relinked, the new size and old size are compared.

The analyzer is approximately 4000 lines of code, and took two man months to write and refine. For the current 11Mb of session data the analyzer takes about 25 minutes of CPU time on a MicroVAX-II.

III. Results

We installed our instrumented version of **make** on a variety of hosts here at Stanford, and have collected data over the past 6 months. During this period, 93 distinct users working on 812 different programs ran **make** 13,012 times. Of these 93 users, 35 generated around 90 percent of the runs. Table I shows the frequency of use for the individual compile and link commands.

Table I also shows that compilation time dominates the overall cost of making a program. Less than 20 percent of the CPU time is spent linking versus compiling.

The ratio of CPU time spent compiling versus linking varies with the number of modules in a program. For small programs (1–5 modules), 27 percent of the turnaround time is spent linking; for medium programs (6–15 modules), the percentage is 17 percent; for large programs (more than 15 modules), the percentage is 24 percent.

A. Program Size

Fig. 1 shows the distribution of program object code size (both instructions and data), with the median size at approximately 30K bytes. Fig. 2 is the distribution of the number of modules in a program. Approximately 40 percent of the programs consist of 3 modules or less. Although most programs in our data sample are small, Fig. 1 shows two-thirds of the total object code is contained in one-fifth of the programs, those 128K or larger.

B. Module Recompilation

Fig. 3 shows the actual number of modules recompiled for individual **make** sessions. About 20 percent of all **makes** were purely links, involving no compiles. Another 50 percent compiled only one module, and about 10 percent compiled two modules. Therefore, approximately 80 percent of all **makes** compiled two or fewer modules.

Fig. 4 shows the distribution as a percentage of the modules in a program that were recompiled. Over 60 percent of the **makes** recompiled less than 20 percent of the modules in the program. Most of the time, therefore, very little of the program is recompiled.

Since 40 percent of all programs contain 3 or fewer modules, we assumed that small programs altered the recompiling distribution. However, we found a similar distribution when considering only programs with 15 or more modules. Thus, the number of modules recompiled does not depend on the number of modules in a program.

Fig. 5 shows the distribution of module size. Three-fourths of the module object code sizes are between 8K and 128K. The narrowness of this range is partially due

TABLE I
COMMAND USAGE

Cmd	Uses	CPU Time per Use (seconds)		
		User	System	Total
cc	10568	15.2	2.4	17.6
CC	5559	26.2	4.4	30.6
mod	2572	62.8	8.4	71.2
pc	1486	67.3	5.8	73.1
f77	684	28.1	3.8	31.9
ld	13775	8.9	4.6	13.5

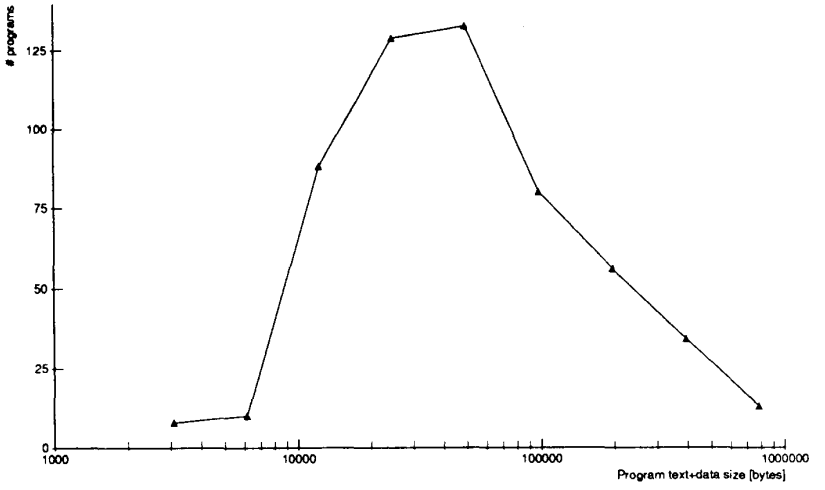


Fig. 1. Number of programs by object size.

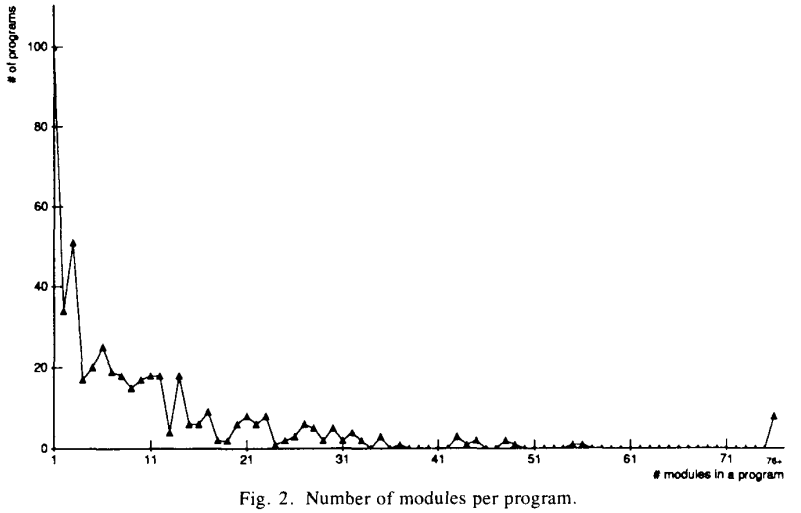


Fig. 2. Number of modules per program.

to modules being files. Small logical modules are often merged into a single file to reduce compilation overhead; large logical modules are often split into several files to avoid recompiling the whole module for every change.

Fig. 6 shows the distribution of changes to module object code size from one compile to the next. The size of the object code did not change 50 percent of the time, and *increased* less than 100 bytes 90 percent of the time. A module might be recompiled without changing its object size for one of the following reasons:

- an interface file that does not actually affect the module is changed

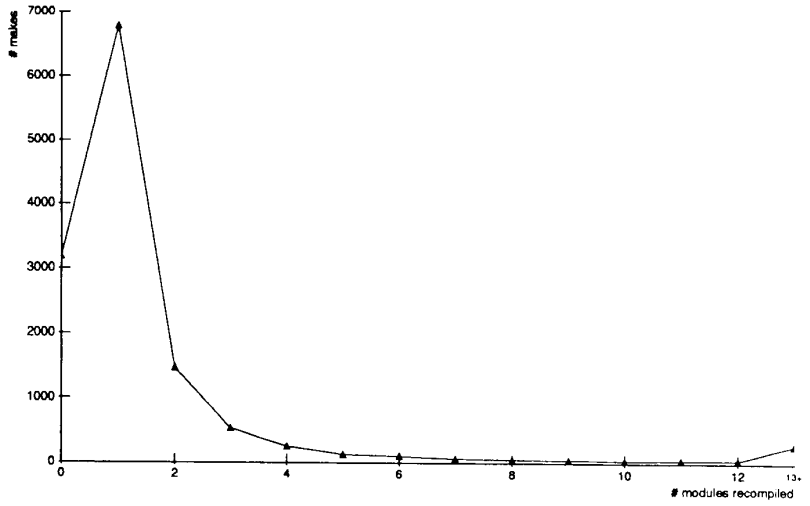


Fig. 3. Number of modules recompiled per **make**.

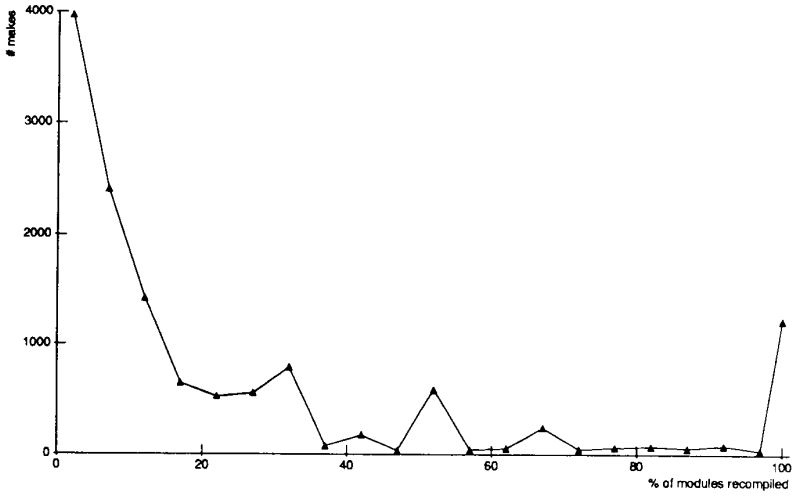


Fig. 4. Percentage of the modules recompiled.

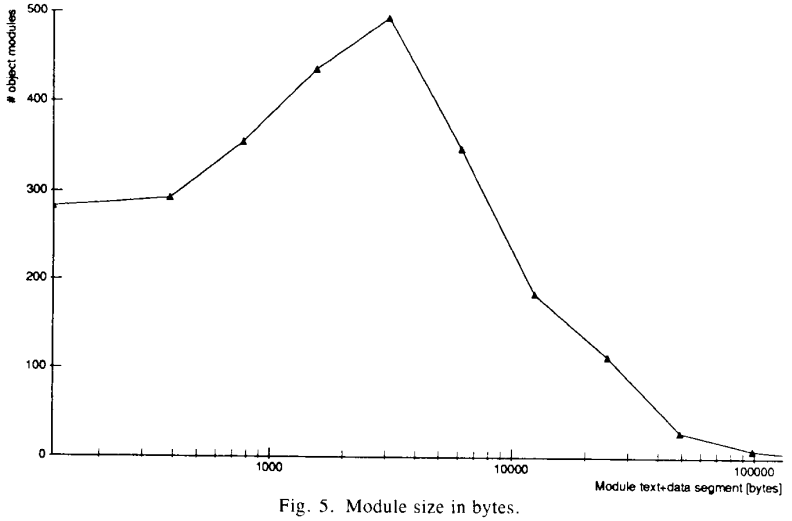


Fig. 5. Module size in bytes.

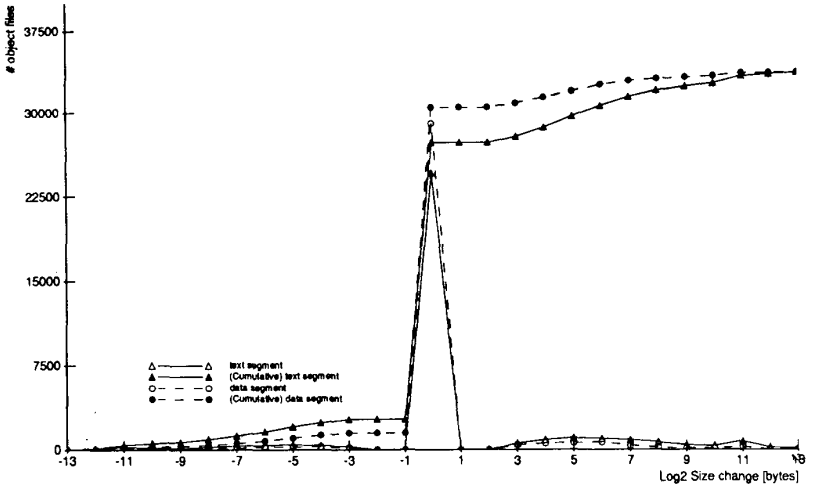


Fig. 6. Change in module size.

- the value of a constant is changed
- the module is copied (e.g., checked in and back out for source code control)
- an unused local variable is deleted.

The data segment size changed even less frequently than code. The data segment size did not change at all 90 percent of the time between compiles, and increased more than 100 bytes only 1 percent of the time.

These results show that code size and data size usually do not change much between compiles. An incremental linker that allocates a small amount of extra space usually will be able to put the new module exactly where the old module was. One hundred bytes of extra space per module code segment should be adequate; less than that is necessary for the data segments.

Compilation errors must be handled specially because a **make** session will have fewer compiles than it would have without compilation errors. For example, suppose three modules need to be recompiled for a program. The programmer runs **make** and the first module compiles but an error is found in the second module. After fixing the error, the programmer again runs **make** and the second module compiles but an error is found in the third module. After fixing that error, the third module compiles and the entire program is linked. To the analyzer, these three sessions appear to be recompiling one or two modules each, whereas there should have been one logical **make** that recompiled the three modules.

To avoid this problem, the analyzer treats such a sequence of **make** sessions as a single logical **make** and ignores compiles that fail. If the same module is compiled more than once without an intervening link, the analyzer only records the last compile, on the assumption the other compiles failed. This assumption was necessary because our **make** does not send the exit status or output from each command.

C. Turnaround Time

Fig. 7 shows the distribution of compilation and link time across **make** sessions. The mean CPU time was a minute, although 80 percent of all sessions took less than a minute. Therefore, while most sessions are relatively short, a few long sessions take up a substantial portion of the total CPU cycles.

The time for compilation consistently dominates the time for linking. The average time for linking is about 10 seconds; the average time for compiling is about 50 seconds.

Fig. 8 shows the distribution of elapsed time between **make** sessions. Ignoring the upper third of the curve, which represents sessions on different days, the mode for the time between program remakes is 10 minutes. Over 80 percent of the time, programs were remade between 50 seconds and 100 minutes of the last **make**. This figure represents the actual time for the "execute-debug-edit" portion of the "edit-compile-execute-debug" iteration.

Fig. 9 shows program turnaround time versus program object code size. For 12K to 200K programs, the turnaround time increases not quite linearly with object code size. The data points below 10K are less significant, as they are not based on many samples. The average turnaround time for programs 800K bytes or larger was less than that for 200K programs. It may be that these very large programs are better organized so that less of the program is rebuilt.

Fig. 10 shows program turnaround time versus the number of modules in a program. On the average, for each doubling of the number of modules the turnaround time increases by 20 seconds for programs with 1 to 24 modules. The turnaround time for larger programs is not correlated with the number of modules they have.

Figs. 11 and 12 show average link time versus object

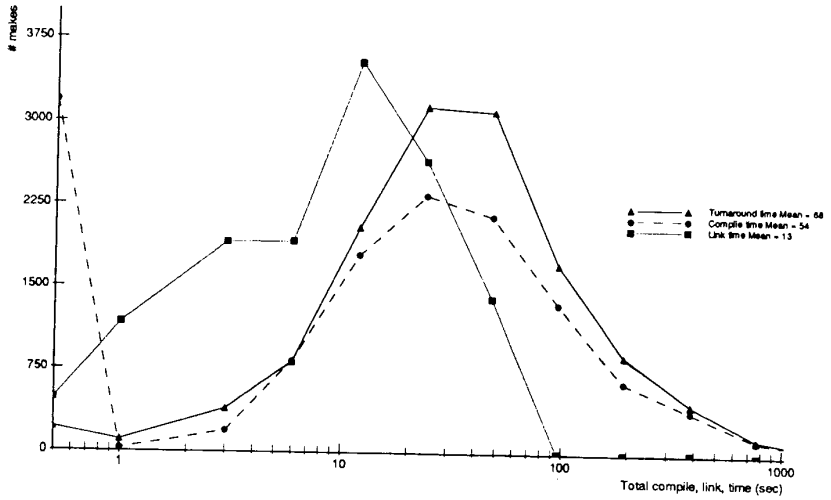


Fig. 7. Total turnaround time.

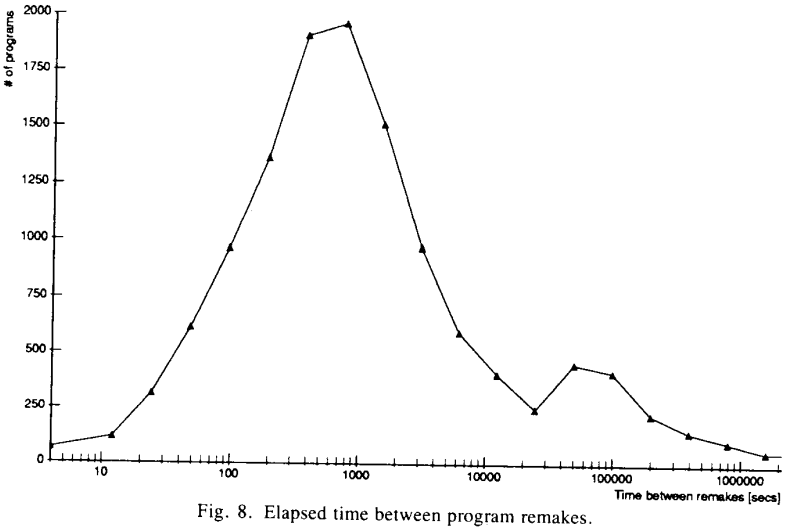


Fig. 8. Elapsed time between program remakes.

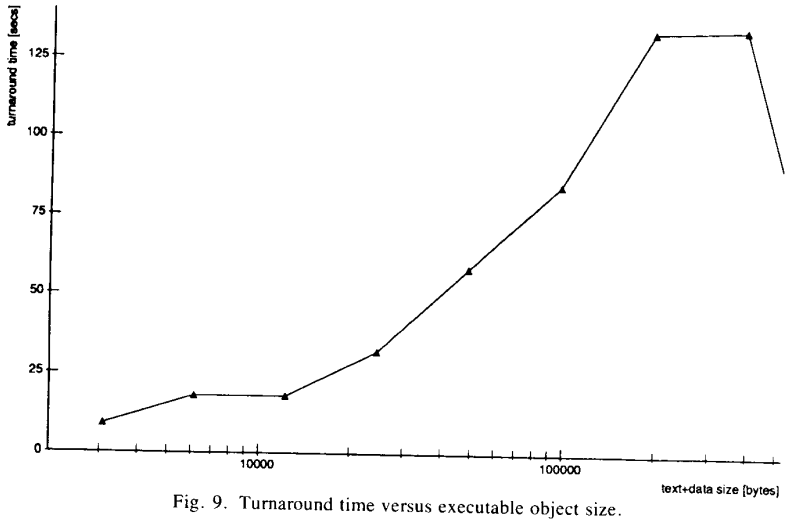


Fig. 9. Turnaround time versus executable object size.

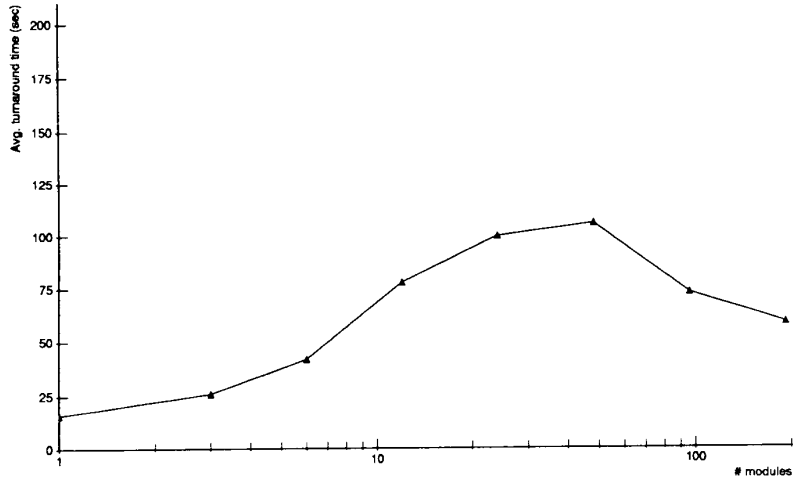


Fig. 10. Turnaround time versus number of modules in the program.

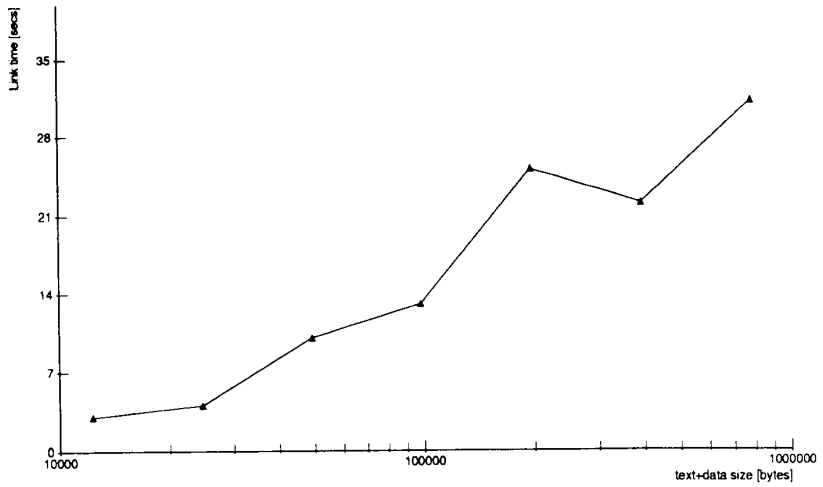


Fig. 11. Link time versus executable object size.

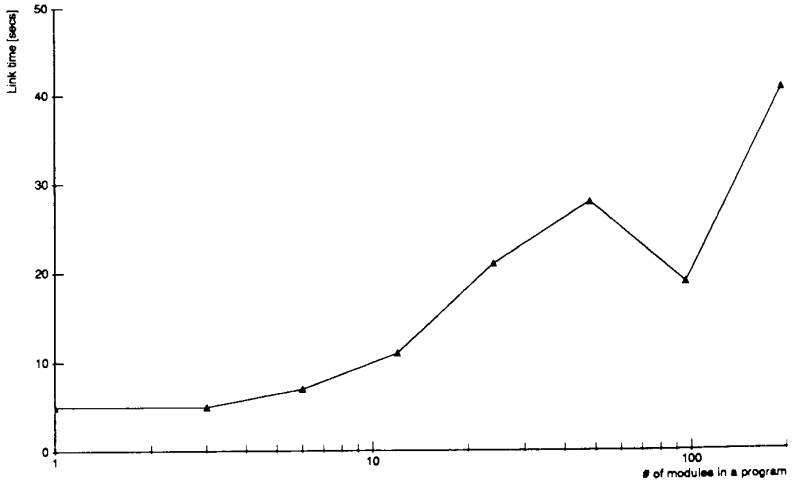


Fig. 12. Link time versus number of program modules.

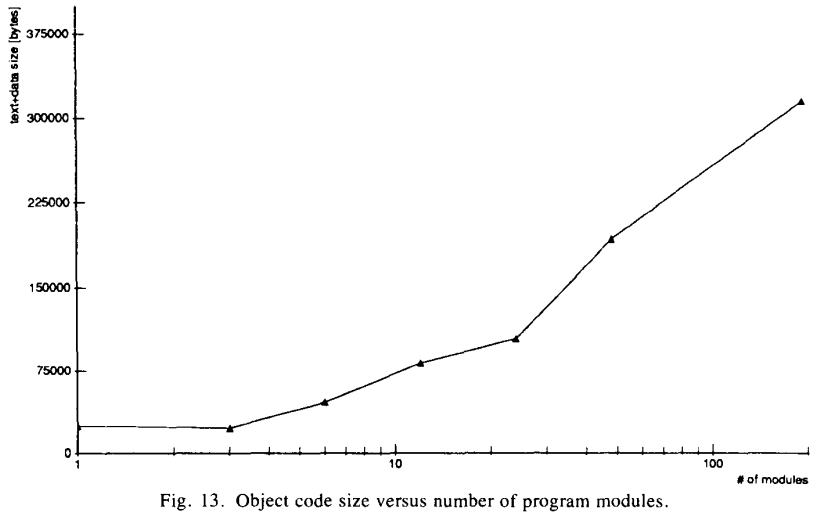


Fig. 13. Object code size versus number of program modules.

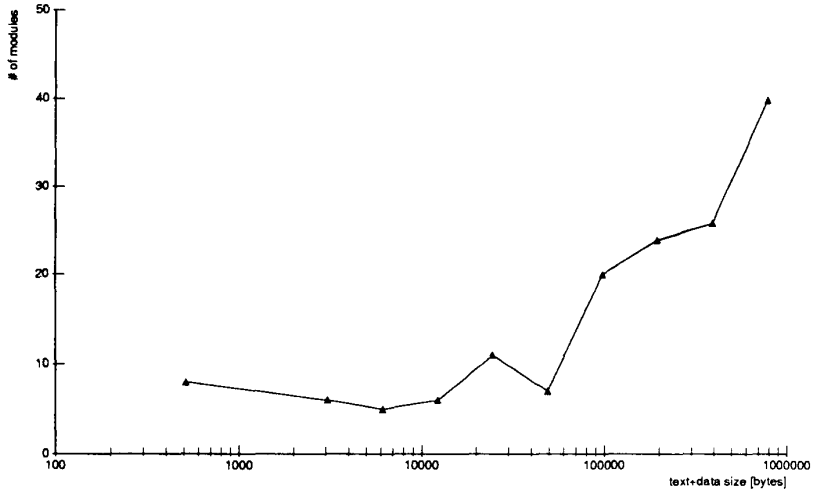


Fig. 14. Number of program modules versus object code size.

code size and number of modules, respectively. On the average, link time increases four seconds for each doubling of the object code size for 12K to 768K programs. This result indicates that link time grows as the logarithm of object code size. Link time also grows as the logarithm of the number of modules, increasing four seconds for every doubling of the number of modules in a program. This result is somewhat misleading since the number of modules does not include libraries linked implicitly and counts an entire user library as a single module. Thus, programs in these figures with 1 user module probably have 6 or 7 modules; programs with 20 user modules probably have 30 to 40 modules when the libraries are counted.

Figs. 13 and 14 compare average object code size with the average number of modules. Fig. 13 shows that object code size increases not quite linearly as the number of modules, with a high correlation between the two measures. Fig. 14 shows that as the object code size increases, the number of modules also increases, but the correlation is not as strong as in the previous figure.

IV. Conclusions

With a small amount of effort, we have instrumented the UNIX **make** utility to give us information about programs and how they are developed. Our approach has given us a broad, macroscopic perspective on the size and magnitude of changes to programs in our environment. We have obtained information about program turnaround time and its breakdown into compile and link time.

Our results indicate that usually a program is remade by compiling a small number of modules no matter how large the program is, and that these modules rarely change by more than 100 bytes. The implication is that an incremental compilation and linking environment is feasible because most changes are already incremental in nature.

We are using these measurements to guide the design of an incremental linker. The relatively small changes in

modules for each link indicate that the linker should allocate slightly more space than the initial size of the module. The infrequent changes to global data and the relatively small number of modules recompiled indicate that incremental linking should be effective. In particular, we see no reason to use indirection through a table of addresses for procedure calls and global data references. The time for an incremental link should be minimal compared to the total turnaround time.

Our measurements also show that compilation is the dominant component of turnaround time, taking almost four times as much CPU time as linking. Much of the cost will be eliminated by performing syntactic and semantic analysis interactively, perhaps with a structure editor. We also plan to investigate further how the costs of compiling can be reduced in an incremental environment.

REFERENCES

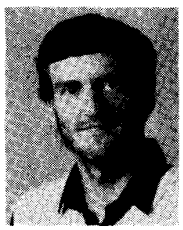
[1] S. I. Feldman, "Make—A program for maintaining computer programs," *Software Practice and Experience*, vol. 9, no. 3, pp. 255–265, Mar. 1979.

[2] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, 1984.

[3] B. Kernighan and J. Mashey, "The Unix programming environment," *Computer*, vol. 14, no. 4, Apr. 1981.

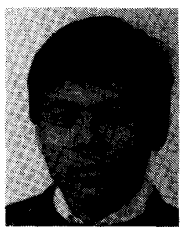
[4] M. L. Powell, "A portable optimizing compiler for Modula-2," *in Proc ACM SIGPLAN '84 Symp. Compiler Construction*, in *SIGPLAN Notices*, vol. 19, no. 6, June 1984.

[5] B. Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.



Mark A. Linton received the B.S.E. degree from Princeton University, Princeton, NJ, in 1978, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1981 and 1983, respectively.

He is an Assistant Professor in the Computer Systems Laboratory of the Department of Electrical Engineering at Stanford University. His current research interests are in programming environments, user interfaces, workstations, and database systems.



Russell W. Quong received the B.S. degree from the California Institute of Technology, Pasadena, in 1983, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1985 and 1988, respectively.

He is an Assistant Professor in the Department of Electrical Engineering at Purdue University in West Lafayette, IN. His current research is in the field of programming languages and the analysis of algorithms.