The Frequency of Dynamic Pointer References in ‘‘C’’ Programs

Barton P. Miller

Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, Wisconsin 53706

1. Introduction

A collection of ‘‘C’’ programs was measured for the number of dynamic references to pointers. The number of dynamic references to pointers is presenteded with respect to the total number of instructions a program executes, giving the percentage of pointer references executed in a ‘‘C’’ program. The measurements were done on a VAX 11/780 running the Berkeley UNIX operating system. The measured programs were selected by examining the most commonly run programs on the Computer Sciences Department UNIX machines. The measurement process was performed in two steps: (1) the dynamic counting of pointer references, and (2) the counting of the total number of instructions executed by the program.

There are several uses for the results presented in this report. One use was for a study of how well ‘‘C’’ programs would run on a CPU that did not easily support 32 bit pointers. Each time a pointer was used, a couple of instructions were needed to form the 32 bit value. If this occurred too frequently, the programs would not run well on this hardware. A second use of these results was for estimating the cost of monitoring pointer references in a debugging system. If pointer references occur too frequently, the cost of monitoring would be prohibitive.

The remainder of this report includes three sections. These are the process used to obtain the measurements, the actual numerical results, and a short discussion of the results.

2. Measurement Process

Each program, in each situation, required two measurements. These were the dynamic count of pointer references in a ‘‘C’’ program, and the count of the total number of instructions executed by the program.

The pointer references were measured by modifying the ‘‘C’’ compiler to add instructions to the object code to meter each reference to a pointer. Each reference to a pointer is accompanied by an increment of a variable totaling these references. When the program completes execution, that value is printed. The code added to the program to meter pointer references has no effect on the correct execution of the program. Explicit pointer references, as well as implied references that causes the ‘‘C’’ compiler to treat them as pointer references (e.g., array references) were counted. ‘‘C’’ programmers often take advantage of the fact that arrays are actually implemented as pointers. This means that it not possible to change the way the ‘‘C’’ compiler implements its arrays (currently working programs may stop doing so), and that the implied pointer references must be counted.

The runtime library must also be considered when measuring the programs. The programs were measure both using a metered version of the runtime library (i.e., one that had been compiled with the modified compiler), as well as using the standard (unmetered) version of the library. This is to determine what percentage of pointer references are generated by the actual program, and that are caused by the runtime library. It is possible to rewrite the runtime library in a way to reduce the number of pointer references. While there can be no control on how user programs use pointers, it is still possible to have control over the (standard) libraries that these programs use. If minimizing the number of pointer references is desirable, then rewriting the standard library provides an additional tool.

The measurement of the total number of instructions executed by a program was done by adapting an instruction measurement program. The program being measured runs as a debugged task, with every instruction being trapped and examined (in this case, only counted). The program being measured is an unchanged version (no pointer counting instructions included).

The following programs have been measured, in the situations described:

ex/vi
: A UNIX screen or line editor. This was run in three situations: (1) start-up the editor with no file, and immediately quit; (2) use the editor to start a new file, adding approximate 20 lines; and (3) editing a 450 line file, adding lines, deleting lines, doing global changes.

csh
: The Berkeley UNIX shell. This was run in two situations: (1) start-up with input; and (2) a 20 line script of typical UNIX commands, including programs to be run and internal shell commands.

sh
: The Bourne shell (from AT&T). This was run on input similar to that for csh.

''C'' compiler
: The UNIX ''C'' compiler. The main compiler phase was used. The preprocessor (macro expansion), and the postprocessor (optimizer) were not measured. Two situations were measured: (1) a short program (size.c, 49 lines); (2) and a medium length program (ls.c, 663 lines).

f77
: The UNIX Fortran-77 compiler. The main compiler phase was used. The postprocessor (optimizer) was not measured. Two situations were measured: (1) a short program (27 lines); (2) and a medium length program (270 lines).

nroff
: The UNIX word processor for non-typeset devices. This was run in two situations: (1) a small file (17 lines, about 2/3 a page output); and (2) a medium file (178 lines, 5 1/2 pages output).

ls
: The UNIX list file utility. This was run in four situations: (1) short format list on an empty directory; (2) long format with sorting by reverse time order on an empty directory; (3) short format list on an large directory; and (4) long format with sorting by reverse time order on an large directory;

Note that the traced programs were not run on large inputs. This is because the dynamic instruction counting has a 100:1 execution cost factor. It is possible to do a few of these, but it is not anticipated that is will produce results that differ significantly from the situations measured.

3. Measurement Results

The data collected is presenteded in Tables 1 and 2 below. Table 1 includes, for each measurement situation, the number of dynamic pointer references, the number of total instructions executed, and the percentage:

$$\frac{\textit{pointer references}}{\textit{instructions executed}} \times 100$$

Table 2 compares the dynamic pointer references for the programs run with and without the metered version of the runtime library. For each measurement situation there is the pointer count with the library being metered, pointer count without the library being metered, and the percentage:

$$\frac{\textit{pointer references w/metered library}}{\textit{pointer references w/o metered library}} \times 100$$

Following are notes on some of the programs that were tested.

The *ex* program makes heavy use of the library routines on startup, doing initialization and reading the file. Adding and deleting text are relatively simple operations and do not involve much pointer manipulation. More complex editing (i.e, global substitutions or use of regular expressions) uses pointers to a much larger extent. The third measurement situation uses the more complex editing facilities, causing a great usage of pointers.

The *nroff* program is a nightmare. It is a direct transliteration of an assembly program into the ‘‘C’’ programming language. The structure is awkward, and there is little use of higher data types. Most data is kept in static, global variables. There is a new version of nroff coming from AT&T, but it will not change in any significant structural fashion. The results included in this study will be applicable to the new version of nroff.

The *ls* program has been recently rewritten to depend heavily on the standard library. All of its I/O, and most of it handling of the directory structures makes use of the library routines. This accounts for the large percentage of pointer references attributable to the runtime library.

The early version of the Bourne Shell (*sh*) that we used makes no use of the runtime library. It calls the UNIX I/O routines directly (doing no buffering), and has all its own routines for string handling.

A bug in the UNIX kernel has prevented the obtaining of the total instruction count from *csh* and *Mail*.

Table 1

Count of pointer references, total instructions executed, and percentage.
(includes library routines)

Sample	Pointer References	Instructions Executed	Percent of Pointer References
LS			
ls (empty directory)	19	1573	1.2%
ls -tl (empty directory)	36	1914	1.9%
ls (big directory)	12251	93894	13.0%
ls -tl (big directory)	25906	374692	6.9%
CC (CCOM)			
size.c (49 lines)	11253	655569	1.7%
ls.c (663 lines)	132925	7040887	1.9%
SORT			
10 line file	802	6305	12.7%
497 line file	655569	2085139	31.4%
MAIL			
Simple (1 recip, short letter)	16374	??	?.?%
Longer (5 recip, long letter)	46392	??	?.?%
EX			
start-up, no file	189	39632	0.5%
new file, add lines	4587	167620	2.7%
existing file (418 lines) edit	95189	658117	14.5%
NROFF			
test.rno (small test)	55372	4473185	1.2%
prop.rno (6 pages)	153867	9489267	1.6%
F77 (PASS1)			
27 line program	14886	3040133	4.9%
270 line program	119496	2163606	5.5%
SH			
trivial script	655	18717	3.5%
complex script	15697	304495	5.1%
CSH			
trivial script	16320	??	?.?%
complex script	34930	??	?.?%

Table 2

Count of pointer references, with and w/o library measured, and percentage.

Sample	w/Metered Library	w/o Metered Library	Percent of References Due to Library
LS			
ls (empty directory)	19	8	42.1%
ls -tl (empty directory)	36	17	47.2%
ls (big directory)	12251	1050	8.6%
ls -tl (big directory)	25906	3240	12.5%
CC (CCOM)			
49 line program	11253	9394	93.0%
663 line program	132925	110970	84.0%
MAIL			
Simple (1 recip, short letter)	16374	13453	92.0%
Longer (5 recip, long letter)	46392	31777	68.5%
EX			
start-up, no file	189	101	53.4%
new file, add lines	4587	4447	96.9%
old file (418 lines) edit	95189	95308	99.8%
NROFF			
test.rno (small test)	55372	55353	99.0%
prop.rno (6 pages)	153867	153848	99.0%
F77			
27 line program	14886	13898	94.0%
270 line program	119496	112659	93.0%
SH			
trivial script	655	655	100.0%
complex script	15697	15697	100.0%
CSH			
trivial script	16320	11663	60.5%
complex script	34930	23116	66.2%