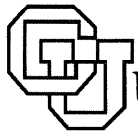
Choosing Program Collections for Performance Evaluation Experiments

Judith A. Stafford
Benjamin G. Zorn

CU-CS-827-97



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Choosing Program Collections for Performance Evaluation Experiments

Judith A. Stafford and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

CU-CS-827-97 January 1997

University of Colorado at Boulder

Technical Report CU-CS-827-97
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Choosing Program Collections for Performance Evaluation Experiments

Judith A. Stafford and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

Contact Author: Benjamin G. Zorn
Telephone: (303) 492-4398
FAX: (303) 492-2844
E-mail: zorn@cs.colorado.edu

January 1997

Abstract

The performance evaluation of programming language implementations often involves selecting and measuring a set of test programs with which to perform the evaluation. The process by which these test programs are selected is often very ad hoc, and in many cases the set of programs selected may directly affect the results of the evaluation. In this paper, we investigate intrinsic characteristics of some widely used program collections, including SPEC92 and SPEC95. We compare these measurements with measurements of a much larger collection of 117 programs that we have collected. Our results show that some important program characteristics, such as median procedure indegree and outdegree, vary widely with the program collection, and show a tendency to increase as the size of programs increases. We also study the effect of program collection size on the variability of the metrics, finding that randomly selected small collections are likely to exhibit characteristics significantly different from the larger collection as a whole.

1 Introduction

In the field of programming language implementation, many performance results are verified experimentally. The process often involves choosing a set of test programs, implementing a performance optimization, and measuring the impact of the optimization on the performance of the test programs. Typically, in such experiments the number and size of the programs in the test suite is limited by the computing and personnel resources available to the researcher. Ever since this method has been employed, researchers (including ourselves) using it have acknowledged that the small numbers of programs used in such studies (typically between 4 and 10 programs) does not constitute a statistically significant sample size. Usually, justifications of the size and content of the test suite rely on suggesting that the programs chosen are interesting in and of themselves, beyond any statistical implications.

The purpose of this paper is to look more carefully at large collections of programs to better understand the statistical significance of results generated from much smaller collections. Based on our large sample of programs, we further seek to understand whether the program characteristics we measure correlate with other aspects of the program (such as size or application domain). Finally, we compare program metrics from industry standard program test suites with those of our much larger program collection.

To understand the motivation for this study, one must first understand why the composition of the test suite is relevant to the results of implementation experiments. To support this point, we describe three current and important areas of programming language implementation research in which the results of experiments rely heavily on the intrinsic characteristics of the test programs measured.

- Implementations of interprocedural pointer alias analysis can be classified as either context-sensitive or context-insensitive algorithms (e.g., see [24]). The context insensitive algorithms account for the fact that procedures can be called from multiple points while the context insensitive algorithms do not. The effectiveness of context insensitive algorithms relative to context sensitive depends heavily on the indegree of procedures in the program's static call-graph. As we show later, the value of this metric varies widely with the suite of test programs chosen. Recently, Ruf pointed out that context-insensitive analysis may perform very well, especially given the characteristics of the benchmark programs used in his analysis [14]. Another optimization significantly affected by the indegree distribution is inlining, where high indegrees can lead to code explosion if selective inlining is not performed.
- The implementation of such diverse techniques as software fault isolation [20], data breakpointing [21], incremental garbage collection [4], and software distributed shared memory [15, 16] all require instrumenting a program's loads and stores. The measured overhead of all of these techniques depends heavily on the dynamic percentage of load and store operations executed by the programs in the test suite.
- Implementation techniques such as instruction scheduling [10] and optimizations to increase instruction level parallelism rely heavily on the size and content of program basic blocks, which may vary significantly depending on the test programs used.

Given that differences in test suite can significantly affect the outcome of implementation experiments, it is still the case that the selection of such suites is largely done on an ad hoc basis and further, to

our knowledge, no studies have been performed to elucidate this process for future experiments. Current experimental approaches include the following:

- Use industry standard benchmark suites. In the field of programming language implementation research, this translates specifically to using the SPEC benchmark suites [19]. These suites include integer (i.e., C) programs, as well as Fortran programs. The number of programs of each type varies, but for the C programs has been 8 in the SPEC92 and SPEC95 suites. These suites have been used extensively, and measured carefully as well. The relatively small size of the programs in the suites (especially the 1992 suite) has also raised questions about their appropriateness for cache architecture studies, for which they are also used.
- Create your own and/or use someone elses ad hoc suite. This is common practice, especially in the case where the application domain does not sufficiently intersect that of the SPEC benchmark programs. Examples of this approach include some of our own previous research, in which we collected a set of allocation intensive programs [6], the Olden [13] and SPLASH [2] benchmark suites (intended for investigating software and hardware to support parallel programming), and the Safe-C suite [3].

Recent trends have shown an increase in the number and size of programs measured. Recent architecture research papers investigating branch prediction implementations provide performance evaluation based on two different benchmark suites [8, 17]. Experimental work by Bart Miller involves testing over 80 Unix utility programs [11]. Recent experimental work of our own involved measuring performance in 43 programs, including several benchmark suites [5].

Given that the use of program test suites for performance evaluation is widespread, and the methods of choosing such suites is ad hoc, our goal is to better understand the characteristics of large program suites. Some of the questions we are interested in investigating are:

- It is generally believed that "bigger" programs make for better benchmark suites, based on the assumption that most important programs (e.g., database servers, word processors, operating system kernels) are quite large. One goal of this paper is to investigate whether large programs are intrinsicly different in structure than small programs.
- How large does a benchmark suite have to be before we have confidence that results based on that sample are representative of a larger collection of programs as a whole? If we assume that the available suites of 8 programs are drawn randomly from a much larger sample, how much impact does the size of the sample have on the variance of program properties we observe?
- How representative of larger collections of programs are the existing industry standard benchmark suites?

We have collected a group of 117 C programs and measured a number of intrinsic static properties of the programs, including indegree and outdegree of procedures, the number of basic blocks per procedure, and the number of executable lines per procedure. The programs were collected from a number of diverse sources, and represent a wide variety of application domains. The programs range from very small to large (approximately 100,000 executable lines of code). In this study, we determine intrinsic metrics for the group as a whole and compared these with the metrics for various subgroups including the SPEC92 and SPEC95 integer benchmarks.

Our results show that certain metrics, such as the median procedure indegree vary widely between programs, and have a tendency to increase as program size increases. On the other hand, the number of instructions per basic block measured appears to remain relatively constant as a function of program size. We also investigate the effect of test suite sample size on the variance of the program metrics measured. We conclude that a sample size of approximately 30 programs is far more likely to result in consistent experimental results than the 8–10 programs that are now frequently used.

This paper has the following organization. In Section 2 we discuss related work, and the SPEC benchmark suites in particular. In Section 3 we describe our evaluation methods, including the programs collected and the metrics measured. Section 4 presents our results and Section 5 concludes and suggests directions for future work.

2 Background

Much of work related to this research has focused on the appropriateness of the various SPEC benchmark suites (e.g., see [19] for the latest release). The SPEC benchmark suites have proven very effective in standardizing the set of measurements used to describe CPU performance for integer and floating point benchmarks on new systems. They have also been adopted widely by the research community in computer architecture and programming languages because the benchmark sources are available to universities for a minimal charge.

One of the main goals of the SPEC effort has been to benchmark the CPU performance of entire systems, including the hardware and whatever compiler optimizations the vendor compiler provides. As a result, it is necessary that the benchmark sources be made entirely available, that the benchmarks do not rely heavily on vendor-specific library code and/or operating system code, and that the programs themselves be highly portable. As a result, the benchmarks are typically versions of publicly-available software that have been modified to reduce their dependence on system libraries and make them more portable.

Because it is so widely used, the various instantiations of the SPEC benchmark suite have been carefully scrutinized. Recently, several researchers have specifically questioned the actual value of the benchmarks themselves [7], as well as the representativeness of SPEC performance results with respect to standard compiler optimization settings [12].

The SPEC benchmarks are selected to cover a diverse collection of application areas with the intention of allowing users of the performance results to look specifically at the result of the program in the application area in which they are interested. As far as we are aware, the programs chosen in the SPEC suites have never been intended to serve as a representative sample of a larger collection of programs. The goal of our work is

to attempt to better characterize program structure based on large collections of programs and understand how current benchmarks relate to the structure we observe.

Other benchmark efforts have attempted to characterize a body of programs with statistical measures, and to construct a synthetic benchmark based on such characterization. In particular the Dhrystone benchmark was constructed after the author conducted a survey of the published literature on source language feature usage in systems programs [23]. While our effort does focus on surveying source-language level features in a large group of programs, our intention is not to construct a synthetic benchmark. Furthermore, the size of the collection we consider is significantly larger, both in terms of numbers of programs and in terms of program size, than the collections considered in the Dhrystone effort. For a survey of benchmarking efforts related to the Dhrystone, see [22].

Measures of program source code have been used widely in the field of software engineering for some time (e.g., see [9]). These measures are used to estimate project costs, evaluate productivity, predict staffing requirements, and to evaluate software quality. In these cases the metrics are used as tools for prediction or evaluation to somehow improve the production of software. In our case, while we measure similar characteristics of programs, we are doing so in an effort to understand properties of large groups of programs as a whole, and to make inferences about the process of performance evaluation based on what we observe.

3 Evaluation Methods

In this section, we discuss the programs we measured, the metrics we measured, and the tools we used to measure them.

3.1 The Programs

For this study we focused on collecting programs that would be representative of those in common use today. We collected 117 programs with two goals in mind. We wanted to collect various sized programs and we wanted to collect programs from varied application domains. Table 1 provides a listing of the programs grouped by where we the program came from. The programs vary in size from the smallest, "hello world" (hello) with three lines, to Xdec, the X server, which is 91091 lines of executable code. For a variety of application domains we chose to look at compilers, interpreters, program development tools, program language development tools, games, numerical programs, text processing tools, and window system tools. We feel these choices have provided us with a broad spectrum of programs that is a good reflection of programs from many application domains that are generally in high use at this time.

We have included the C programs from both the SPEC92 and SPEC95 benchmark sets. Three of the programs in the SPEC92 set (gcc, li and compress) are included in both SPEC92 and SPEC95. We wanted

SOURCE	PROGRAMS
SPECINT92	backprop†, compress†, ear, eqntott, espresso, gcc(version 1.35)†, li†, sc†
SPECINT95	compress*, gcc(version 2.5.3)*, go, ijpeg, li*, m88ksim, perl, vortex
XC(X11R6)	appres, atobm, bdftopcf, bitmap, bmtoa, editres, fsinfo, fslsfonts, fstobdf, iceauth, lndir, mkfontdir, oclock, resize, sessreg, smproxy, twm, x11perf, xauth, xclipboard, xclock, xcmsdb, xconsole, xcutsel, xdm, xdpyinfo, xfd, xfs, xhost, xieperf, xinit, xkill, xlogo, xlsatoms, xlsclients, xlsfonts, xmag, xmh, xmodmap, xprop, xrdb, xrefresh, xset, xsetroot, xsm, xsmclient, xstdcmap, xterm, xwd, xwininfo, xwud, Xdec
Safe C [3]	anagram, backprop*, bc*, ft, ks, yacr2
GNU	bash, bc†, bison, flex, gawk, gcc†, gnuchess, gnugo, gnuplot, gzip, indent, od, sed, sort, tcsh, wdiff
AF Vers. 3, Release 1	Aaxp, Aj300, Ajv, Alofi, Amsb, aecho, aevents, ahost, alsatoms, apass, aphone, aplay, aprop, arecord, aset, awgn
MISC	burg, cfrac, chameleon, gzip, hello, iburg, imake, indent, magic, makedepend, mrtest, python, scm2, siod, sis, tclsh, txl, yacr

Table 1: Programs Used Grouped by Source. Several programs in the collection were obtainable from more than one of our sources. The instrumented program was taken from the group where the program appears marked with a *. Those not included are marked with a †.

to include measures for both benchmarks sets so have measured both sets of programs and reported the results in Section 4. For other groupings we did not want to include two versions of the same program so we chose to use those from the SPEC95 set.

Our study heavily relied on the use of ATOM, a tool that runs only on DEC Alpha machines. For this reason, we were restricted to finding programs that we could build on the DEC Alphas. The programs were compiled with the cc compiler, shared libraries and with the -g3 debugging flag set.

We have also classified the programs according to their application domain. These groups are listed in Table 2. This table includes the application domain, the number of programs in the group, the mean number of lines of executable code for each grouping and the list of programs. Many programs are used in multiple groups. For instance, gcc(version 2.5.3) appears in No Xclients, SPEC95, Program Development Tools, and in the Biggest groups. The X clients comprise a large percentage of the collection (44%). These programs tend to be utility programs that perform simple tasks such as xkill, that kills a window, or xbiff, that checks for new mail. The X client programs tend to have little interconnection but make heavy use of the X11 libraries. These programs are in heavy use in most Unix environments, therefore they help meet our goal of building a program set that reflects programs in common use. We have taken the significance of their proportion of our population into consideration when analyzing our results.

APPLICATION DOMAIN	MEAN LINES/PROG	PROGRAMS
All(117)	4893	See next two groups
Xclients(51)	1469	appres, atobm, bdftopcf, bitmap, bmtoa, editres, fsinfo, fslsfonts, fstobdf, iceauth, lndir, mkfontdir, oclock, resize, sessreg, smproxy, twm, x11perf, xauth, xclipboard, xclock, xcmsdb, xconsole, xcutsel, xdm, xdpyinfo, xfd, xfs, xhost, xieperf, xinit, xkill, xlogo, xlsatoms, xlsclients, xlsfonts, xmag, xmh, xmodmap, xprop, xrdb, xrefresh, xset, xsetroot, xsm, xsmclient, xstdcmap, xterm, xwd, xwininfo, xwud
No Xclients(66)	7539	aecho, aevents, ahost, alsatoms, anagram, apass, aphone, aplay, aprop, arecord, aset, awgn, Aaxp, Aj300, Ajv, Alofi, Amsb, backprop, bash, bc, bison, burg, chameleon, cfrac, compress, ear, eqntott, espresso, flex, ft, gawk, gcc, gnuchess, gnugo, gnuplot, go, gzip, hello, iburg, ijpeg, imake, indent, ks, li, m88ksim, magic, makedepend, mrtest, od, perl, python, sc, scm2, sed, siod, sis, sort, tclsh, tcsh, txl, vortex, wdiff, xlogo, Xdec, yacr, yacr2
SPEC92(8)	5130	backprop, compress, ear, eqntott, espresso, gcc, li, sc
SPEC95(8)	14440	compress, gcc, go, ijpeg, li, m88ksim, perl, vortex
Audio(11)	1308	aecho, aevents, ahost, alsatoms, apass, aphone, aplay, aprop, arecord, aset, awgn
CAD(6)	19906	chameleon, espresso, magic, sis, yacr, yacr2
Numerical(3)	519	backprop, ear, ft
Interpreter(11)	8435	bash, bc, gawk, li, perl, python, scm2, sed, siod, tclsh, tcsh
Prog. Lang. Development Tool(5)	3037	bison, burg, flex, iburg, txl
Program Development Tool(4)	14423	gcc, imake, indent, makedepend
Games(4)	3732	anagram, gnuchess, gnugo, go
Text Utils(3)	870	od, sort, wdiff
Misc(18)	9941	Aaxp, Aj300, Ajv, Alofi, Amsb, Xdec, cfrac, compress, eqntott, gnuplot, gzip, hello, ijpeg, ks, m88ksim, mrtest, sc, vortex
Biggest(10)	35333	bash, gcc, gnuplot, magic, python, sis, tcsh, vortex, xfs, Xdec
Smallest(10)	87	appres, atobm, backprop, bmtoa, hello, lndir, showrgb, xcutsel, xlsatoms, xlsclients
Smallest(No Xclients)(10)	291	anagram, backprop, compress, ft, hello, iburg, imake, ks, makedepend, wdiff

Table 2: Programs Used Grouped by Application Domain. All data for SPEC92 was collected independent of the overall collection of 117 programs. The SPEC92 versions of backprop, compress, gcc and li were instrumented and used to calculate the means for this subgroup but not included in the All group.

3.2 Data Collection

We instrumented the programs using ATOM, a tool developed by Digital Equipment Corporation [18]. ATOM provides a flexible means by which to create program analysis tools based on instrumentation of executable code. ATOM allows a program executable to be analyzed and instrumented by providing an API that allows procedures, basic blocks, and instructions in the executable to be navigated. While ATOM provides for instrumenting and measuring the dynamic behavior, in this study we used it solely to measure properties of the static executable.

We did not include system libraries (e.g., libc.a, libm.a, libX11.a, etc) in our statistics as we felt these measurements would skew the results. We did not want to include library routines that are used by many of the programs since including them with each program would tend to suppress the individual characteristics of the programs under study. On the other hand, if the application included a set of routines compiled into a library (e.g., tclsh uses libtcl.a), then we included these libraries as part of the application program. We did not optimize the code as our purpose is to collect measurements that more closely reflect characteristics of the program source.

For the study we have measured the instructions per block, blocks per procedure, indegree per procedure, outdegree per procedure, lines of executable code per procedure and executable lines of code per program. We chose the first five metrics as being representative of metrics that are of interest in performance analysis research. The executable lines of code measure is important to this work as a means of grouping the programs. The items we measure are mostly compiler and architecture independent since, for the most part, they reflect the source code structure of the program. Therefore, our results may be considered representative of programs being compiled on different systems in different ways.

The lines of code (LOC) measurement was achieved by checking the source line number associated with each instruction. Each line number was recorded exactly once for all instructions associated with that source line. Our LOC measurements reflect the lines of code that will actually be executed. Also, we do not include the system library code, as discussed above, nor do we include generated code such as yyparse that is generated by the parser generator, Yacc. We feel that including the generated code would be useful for performance analysis but would not reflect the code written by the programmer. Our dependence on ATOM, which relies on information from the symbol table for line count information, made the inclusion of line counts from this generated code very difficult. We have chosen to leave these generated lines of code out of our program size measurements. However, the other five metrics do include instrumentation of this code.

We note that our LOC counts are often significantly lower than those reported in other papers. We feel that our counts closely represent the actual user written code. LOC is a metric that is widely used in the research community. The determination of what constitutes a line of code is very difficult. The figures can vary greatly, as is shown in Table 3. We feel that this decision should be made with great care, precision

PROGRAM	LINES OF CODE		EXECUTABLE/
	EXECUTABLE	GROSS	GROSS
bash	17240	37349	.46
burg	2448	5675	.43
cfrac	1090	4213	.26
compress	359	1934	.18
ear*	1084	5239	.21
eqntott*	1139	3139	.36
espresso*	5936	14838	.40
flex	3045	14516	.21
gcc	54424	205085	.27
go	9680	29246	.33
gzip	2130	8206	.26
iburg	632	799	.79
ijpeg	7861	31211	.25
indent	2068	6492	.32
li	2765	7597	.36
m88ksim	6053	19915	.30
magic	45798	219770	.21
perl	12063	26871	.45
python	15870	80657	.20
sc	3168	8515	.37
twm	5994	19096	.31
vortex	22315	67202	.33
xclipboard	298	1037	.29
xclock	308	1267	.24
yacr	2961	9367	.32

Table 3: Comparison of Program Size Measures. The Gross lines of code was found by using wc on the .c and .h source files. We compared our gross lines of code counts with those published by SPEC for the three SPEC92 programs which have been included (marked with *). The counts are very comparable.

and with the goals of the project in mind. For this study, we are concerned with relative size of programs as well as the accurate representation of the executable lines written by the programmer. We feel that the measurements achieved through the use of ATOM and our algorithm provide an accurate reflection, if somewhat conservative, of the size of this code.

The indegree and outdegree of a procedure are metrics that affect program optimizations (e.g., inlining and alias analysis as mentioned). The outdegree of each procedure was measured by recording each subroutine call instruction. The outdegree does include calls to system library routines. The indegree for each procedure was found by looking at the number of times each procedure was a target of a subroutine call. Those subroutines that were never a target were omitted from the data collection. These procedures can be the result of including a library that contains unused functions or they may be the target of an indirect function call. This determination in not possible without performing a dynamic measurement.

We also consider the number of instructions in each basic block and the number of blocks contained in each procedure to be important particularly to the language implementation community. For each procedure we recorded the number of blocks in the procedure and the size of each of those blocks in terms of instructions per block. A basic block of code is considered to be the set of instructions in which, if any one of the instructions is executed, all others in the set will also be executed. These statistics are directly obtained by the use of ATOM procedures.

3.3 Discussion

For each program we computed a histogram of the counts for each of the metrics of interest. From these histograms we were able to find the median number of occurrences of each metric within the program. We computed the average of the medians for each group identified in Table 2. Due to the large percentage of programs obtained from the X release, we have split some application domain groups based on inclusion and exclusion of the X client programs. For example, the Smallest group would be overly represented by the X clients so we also measured the smallest not including these programs.

There are limitations in any attempt to view the source of a program through ATOM instrumentation. Since ATOM instruments the executable all changes made to the code during preprocessing are seen as part of the source. This affects our study in two ways. Preprocessing deletes any lines not applicable to the variant being compiled. That is, if the programmer wrote the program with a debugging and non-debugging variant in mind, and used ifdefs to do this, ATOM would see only the code for the variant that we built. For our study, this means that we are sometimes missing code that was written by the programmer, such as when they have had to provide different code for different compilers or different architectures, which did not contribute to our particular executable. This is appropriate for our measures since this code is basically duplicated code by the programmer and is not of concern in the area of performance analysis. Secondly,

APPLICATION DOMAIN	AVERAGE OF MEDIANS		
	IN DEGREE	OUT DEGREE	LINES/PROC
All(117)	1.556	3.060	10.983
Xclients(51)	1.471	3.725	10.255
No Xclients(65)	1.621	2.545	11.546
Spec92(8)	2.000	2.500	11.000
Spec95(8)	2.000	2.750	13.750
Audio(11)	1.273	2.273	10.636
CAD(6)	1.500	3.000	12.833
Numerical(3)	1.000	1.333	8.667
Interpreter(11)	2.273	2.909	10.636
PL Dev Tool(5)	1.600	3.400	11.000
Prog Dev Tool(4)	2.000	3.250	11.750
Games(4)	1.500	2.250	17.250
Text Util(3)	2.000	2.667	11.667
Misc(18)	1.500	2.222	11.722
Biggest(10)	2.100	3.300	11.900
Smallest(10)	1.300	3.600	9.500
Smallest noxc(10)	1.100	2.100	10.000

Table 4: Procedure Level Metrics.

generated code, that such as yyparse which is generated by Yacc is included. While this code is of interest for the purpose of performance analysis, we did not want to include the lines in our program size measurement so the LOC counts of this code were not included in the program size but we include these measures in all other metrics.

4 Results

In this section, we present our measurements of the test programs. In an effort to better understand the structural characteristics of certain classes of programs, we have grouped the programs by application domain. Unfortunately, in the sample of programs we have collected, some of these application domains are underrepresented (e.g., there are only three numerical programs). As a result any observations we make based on these small subgroups lack statistical significance. We acknowledge this weakness, but still point out interesting trends that we observe. If our conclusions are controversial, we invite other researchers to challenge our results with more thorough investigations.

4.1 Procedure Level Metrics

Table 4 summarizes the means of the medians for three procedure-level metrics across the collections of programs studied. Specifically, we measured the median value for each program in the collection, and

present the mean of the medians sampled over all the programs in each collection. The metrics considered are the median procedure indegree, median outdegree, and median lines of code (discussed in Section 3) per procedure. We discuss in turn how the average of these medians differs across the program collections studied.

4.1.1 Procedure Indegree

As mentioned, the indegree of procedures may significantly influence the effectiveness of optimizations such as interprocedural analysis and procedure inlining. The average median indegree ranges from a low of 1.0 to a high of 2.3 across the collections studied, with the average for the overall collection being 1.56. Note that because we eliminate all procedures with indegree zero, the lowest average indegree possible is 1.0, which corresponds to a collection in which every program had greater than 50% of all procedures with only one caller.

The distribution of procedure indegrees corresponds directly to the amount of procedure-level reuse occuring in a program. The table suggests that a trend exists going from the collection of smallest programs to the collection of largest programs, in which the average median indegree (and correspondingly procedure reuse) increases substantially (from 1.3 to 2.1) as programs get bigger. The value of 2.1 for the 10 largest programs indicates that, on average, more than half of all the procedures in these programs have more than one caller. Such a trend suggests that context insensitive interprocedural analysis may become increasingly less effective as programs become larger.

Comparing the collection of biggest programs with the overall average suggests that, in general, many classes of programs do not have as much reuse as the largest programs. In particular both the SPEC92 and SPEC95 collections approximate the structure of the large programs more closely than the overall average of the entire collection. Another interesting result is that the programs classified as Interpreters, which in general are substantially smaller that the largest programs, have the largest mean indegree, with value 2.3. Trying to establish if there is a common structure in these programs that accounts for such a high overall indegree is a subject of future work.

4.1.2 Procedure Outdegree

The procedure outdegree corresponds to how many call sites occur in each procedure. Table 4 indicates that the average median outdegree of the overall program collection was approximately 3.1, while some classes of programs deviated from the overall average quite dramatically. First, it is clear that the X client programs, with mean outdegree of 3.7, deviate substantially from the other programs in the collection. This behavior can be explained, in part, by the low-level nature of the X library interfaces, and the significant number of procedure calls necessary to accomplish even very simple tasks. Many of these clients are very simple

programs that glue together different pieces of code in the X libraries. If the X clients are excluded from the overall collection, the average drops from 3.1 to 2.5.

If the X clients are eliminated from the collection of smallest programs, we see that there is again a trend from the smallest non-X client programs to the largest programs in which the mean outdegree increases significantly, from 2.1 to 3.3. This trend suggests an increasing reliance on procedural abstraction in the larger programs, and with it the increased possibility for code reuse, which we have already seen in the previous section.

In both SPEC92 and SPEC95, the collections have means comparable to the overall average excluding X clients, (2.5 and 2.75 versus 2.5), but their mean is significantly lower than the mean outdegree of the largest programs. Also, many of the other program collections (e.g., Interpreters, CAD tools, etc) have mean outdegrees higher than the SPEC95 collection.

4.1.3 Lines per Procedure

The average median number of LOC per procedure is approximately 11, which means that if we suppose that comments and declarations expand that number by a factor of four, more than 50% of procedures will fit on one 60-line page.

The LOC per procedure metric also shows a trend when looking from the collection of the smallest programs (at 9.5 LOC/proc) to the largest (at 11.9 LOC/proc). While this metric remained relatively steady across many of the different collections, the SPEC95 collection showed substantially larger procedures than the other collections.

4.2 Instruction Level Metrics

Table 5 summarizes the means of the medians of two instruction-level metrics in the program collections studied. The two metrics measured are median blocks per procedure and median instructions per block. We discuss the instructions per block first. The table shows that the mean median instructions per block does not vary dramatically among the different collections measured, with the median value being 4 instructions in all but 30 of the 117 programs. Also, the value of this metric does not appear to be sensitive to the size of programs in the program collection, as we have seen in the other metrics investigated. Although the difference is slight, the X clients appear to contain slightly larger basic blocks than non-X clients (4.3 instructions versus 4.1) and the Interpreters appear to contain slightly smaller basic blocks than the overall collection (3.8 instructions versus 4.15). These differences are so slight, however, that they can be accounted for simply by a couple of programs in the collection having a median basic block size of 3 instead of 4.

The median blocks per procedure metric provides another view of the procedure size in the program collections measured. The results provided using this metric correlated closely with the previous results

APPLICATION DOMAIN	AVERAGE OF MEDIANS	
	BLKS/PROC	INST/BLOCK
All(117)	13.179	4.154
Xclients(51)	12.686	4.274
No Xclients(66)	13.560	4.061
Spec92(8)	12.625	4.250
Spec95(8)	16.000	4.125
Audio(11)	11.818	3.636
CAD(6)	16.333	4.167
Numerical(3)	8.000	4.667
Interpreter(11)	13.454	3.818
PL Dev Tool(5)	12.400	4.400
Prog Dev Tool(4)	17.000	4.000
Games(4)	19.250	4.500
Text Util(3)	15.333	4.000
Misc(18)	13.111	4.167
Biggest(10)	14.800	4.100
Smallest(10)	12.300	4.100
Smallest noxc(10)	10.900	4.000

Table 5: Instruction-Level Metrics.

using LOC as a measure of procedure size. As before, we see that the larger programs have more blocks per procedure than the smaller ones.

4.3 How Program Collection Size Affects Metrics

In order to gain some insight into the number of programs required to achieve a good sample of realistic program structure, we considered random program collections of different sizes drawn from our 117 program sample. We considered sample sizes ranging from 6 programs to 110 programs, and we collected statistics over 1000 different randomly selected groups from each sample size. From these runs we calculated the mean and standard deviation of these means for each sample size.

Table 6 shows the means and standard deviations of median indegree, median outdegree, and median LOC per procedure for randomly sampled groups of different sizes as described above. The intention of this table is to illustrate how much variance occurs when a random sample of a particular size is chosen. Likewise, Table 7 presents similar results for the blocks per procedure and instructions per procedure metrics. As the central limit theorem predicts, drawing from a larger random sample reduces the variance of the metrics in all cases.

To get a better feel for how the variance changes as the sample size increases, we have plotted the standard deviation as a percentage of the mean for the different metrics and group sizes in Figure 1. The figure shows that while all variances are decreased as the sample size increases, some metrics are more influenced by

GROUP (PROGS SET SIZE)	IN DEGREE		OUT DEGREE		LINES/PROC	
	MEAN	σ	MEAN	σ	MEAN	σ
6	1.558	0.244	3.050	0.600	10.965	1.260
8	1.559	0.210	3.047	0.523	10.962	1.110
10	1.558	0.188	3.050	0.466	10.971	0.951
20	1.563	0.125	3.066	0.321	10.980	0.651
30	1.558	0.097	3.067	0.252	10.996	0.514
40	1.558	0.080	3.064	0.205	10.989	0.427
50	1.556	0.066	3.058	0.167	10.989	0.343
60	1.555	0.055	3.059	0.137	10.984	0.286
70	1.555	0.047	3.058	0.116	10.984	0.242
80	1.555	0.038	3.058	0.095	10.978	0.199
90	1.555	0.030	3.056	0.079	10.977	0.162
100	1.556	0.023	3.057	0.061	10.984	0.119
110	1.556	0.014	3.059	0.037	10.984	0.071

Table 6: Procedure Level Metrics. The mean values for the programs grouped by numbers of programs reflects the mean of the means of the medians from 1000 randomly chosen sample sets.

GROUP (PROG SET SIZE)	BLKS/PROC		INST/BLOCK	
	MEAN	σ	MEAN	σ
6	13.147	1.551	4.152	0.233
8	13.158	1.315	4.149	0.199
10	13.159	1.144	4.149	0.174
20	13.134	0.766	4.153	0.116
30	13.167	0.594	4.153	0.088
40	13.169	0.488	4.154	0.071
50	13.170	0.409	4.154	0.061
60	13.176	0.350	4.154	0.051
70	13.178	0.291	4.154	0.043
80	13.174	0.237	4.153	0.035
90	13.171	0.193	4.154	0.028
100	13.180	0.142	4.154	0.021
110	13.181	0.086	4.154	0.013

Table 7: Instruction-Level Metrics. The mean values for the programs grouped by numbers of programs reflects the mean of the means of the medians from 1000 randomly chosen sample sets.

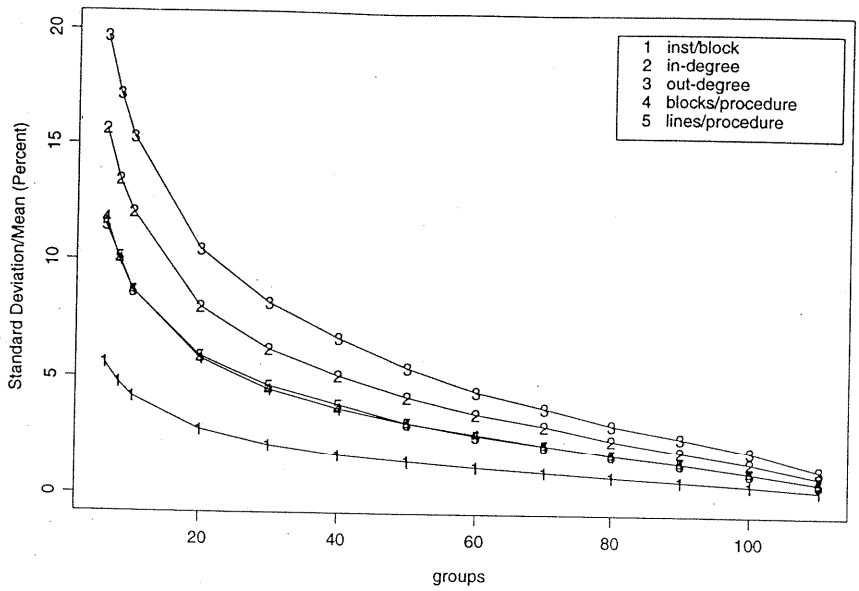


Figure 1: Effect of Group Size on Variance

sample size than others. In particular, for collections with a small number of programs (e.g, 6–10 programs, as are currently used in practice), the standard deviation of metrics such as indegree and outdegree can represent as much as 12–20% of the mean. Often in current practice, performance improvements of small percentages are reported based on program collections with fewer than 10 programs in them. Given our results, if reported performance improvements are related to the indegrees and outdegrees of the programs in the collection, then statistical variance alone may account for the observed improvements in performance. Alternatively, program metrics that are consistent across all the programs in the collection, such as basic block size, show a standard deviation less then 5% of the mean even with small collection sizes.

Another way to understand the effect of sample size on the metrics measured is to plot the probability density of a particular sample average as a function of sample size. Figure 2 shows the probability densities of the mean of the median indegree for a program collection as a function of the collection size. Essentially, this figure illustrates the central limit theorem graphically, but more specifically, it provides insight into the range of values one might expect to see from program collections containing only a few programs. As the figure shows, with a group size of six, the density function is very shallow, and a large fraction of the area under the curve falls quite far from the overall average. Any particular random collection of 6 programs will have a mean of the median indegree that corresponds to the distribution shown, and many such samples will have a mean that differs significantly from the mean of the group as a whole. On the other hand, with a group size of 30, far more sample means fall quite close to the overall mean, and beyond 30 programs the distribution of the means is very consistent.

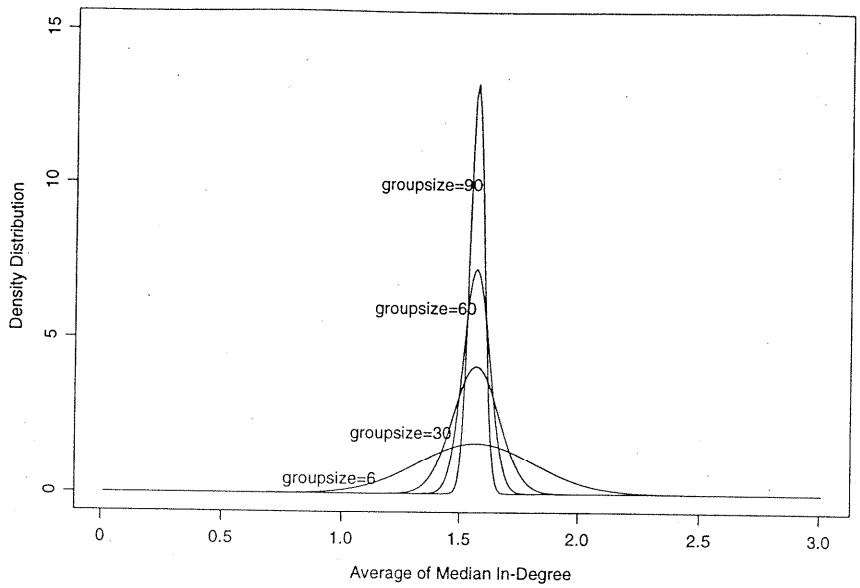


Figure 2: Probability Density as a Function of Program Group Size

5 Summary

Many performance evaluation results are based on measurements of small collections of programs. Previous work has suggested that the program collection chosen can significantly affect the outcome of the evaluation [14]. We were interested in knowing how intrinsic program characteristics such as median procedure in and outdegree vary over the programs in a collection. We were further interested in knowing how the average of these characteristics varies with the size and content of the program collection chosen.

We have shown that the mean over a collection of the median procedure indegree, outdegree, and procedure size appear to increase as the size of the programs in the collection increases, in some cases quite substantially. Among other things, this result suggests that context sensitive pointer alias analysis will be increasing important when it is applied to larger programs. We have also shown that if collections of six or eight programs are randomly drawn from a larger collection, the statistical variance of the metrics measured may be significant enough to affect the result of a performance evalution.

We view this work as the starting point to a more careful evaluation of the methods currently used in experimental computer science. Our goal is to work toward characterizing the size and contents of program collections in an effort to make published performance evaluation results more relevant. In the future, we plan to add more programs to our collection, and correspondingly make the size of the subcollections (e.g., the Numerical programs) more substantial. We also plan to perform additional static measurements, and to include dynamic behavior measurements in our results as well. While in this paper, we only consider the

effect of collection contents on program metrics, in the future we would like to study the effect of collection contents on the outcome of performance evaluation experiments.

References

[1] ACM SIGPLAN. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, June 18–21, 1995. *SIGPLAN Notices,* 30(6), June 1995.

[2] J. Arnold, D. Buell, and E. Davis. SPLASH II. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pages 316–322, San Diego, CA, USA, June 1992. ACM Press.

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

[4] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[5] Brad Calder, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Corpus-based static branch prediction. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 79–92, La Jolla, CA, June 1995.

[6] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software—Practice and Experience*, 24(6):527–542, June 1994.

[7] Ran Giladi and Niv Ahituv. SPEC as a performance evaluation measure. *IEEE Computer*, 28(8):33–42, August 1995.

[8] Nicolas Gloy, Cliff Young, J. Bradley Chen, and Michael D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 12–21, Philadelphia, PA, May 1996.

[9] Stephan H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley, 1995.

[10] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* [1], pages 151–162. *SIGPLAN Notices,* 30(6), June 1995.

[11] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Meganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examinaation of the reliability of unix utilities and services. Technical report, Computer Sciences Dept., Univ. of Wisconsin—Madison, 1996.

[12] Nikki Mirghafori, Margret Jacoby, and David Patterson. Truth in SPEC benchmarks. *Computer Architecture News*, 23(5):34–42, December 1995.

[13] Anne Rogers, Martin C. Carlisle, John H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[14] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* [1], pages 13–22. *SIGPLAN Notices,* 30(6), June 1995.

[15] Daniel J. Scales, Kourosh Garachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, October 1996.

[16] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, San Jose, California, October 4–7, 1994. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 22, October 1994; *Operating Systems Review,* 28(5), December 1994; *SIGPLAN Notices,* 29(11), November 1994.

[17] Stuart Sechrest, Chih-Chieh Lee, and Trevor Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 22–32, Philadelphia, PA, May 1996.

[18] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994.

[19] Standard Performance Evaluation Corportation, Manassas, VA. *SPEC95 Technical Manual*, August 1995.

[20] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, Asheville, NC, December 1993.

[21] Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 1–12, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[22] Reihnold P. Weicker. An overview of common benchmarks. *IEEE Computer*, 23(12):65–75, December 1990.

[23] Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[24] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* [1], pages 1–12. *SIGPLAN Notices,* 30(6), June 1995.