An Empirical Study of Static Program Slice Size

DAVID BINKLEY
Loyola College Maryland
and
NICOLAS GOLD and MARK HARMAN
King's College London

This article presents results from a study of all slices from 43 programs, ranging up to 136,000 lines of code in size. The study investigates the effect of five aspects that affect slice size. Three slicing algorithms are used to study two algorithmic aspects: calling-context treatment and slice granularity. The remaining three aspects affect the upstream dependencies considered by the slicer. These include collapsing structure fields, removal of dead code, and the influence of points-to analysis.

The results show that for the most precise slicer, the average slice contains just under one-third of the program. Furthermore, ignoring calling context causes a 50% increase in slice size, and while (coarse-grained) function-level slices are 33% larger than corresponding statement-level slices, they may be useful predictors of the (finer-grained) statement-level slice size. Finally, upstream analyses have an order of magnitude less influence on slice size.

Categories and Subject Descriptors: D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, Reverse Engineering, and Re-engineering; D.2.5 [**Testing and Debugging**]; D.2.8 [**Metrics**]: Product Metrics

General Terms: Algorithms, Measurement

Additional Key Words and Phrases: Program slicing, slice size

ACM Reference Format:

Binkley, D., Gold, N., and Harman, M. 2007. An empirical study of static program slice size. ACM Trans. Softw. Eng. Methodol. 16, 2, Article 8 (Apr. 2007), 32 pages. DOI = 10.1145/1217295.1217297 http://doi.acm.org/10.1145/1217295.1217297

A preliminary version of this article appeared in *Proceedings of the 19th IEEE Conference on Software Maintenance (ICSM 2003)* [Binkley and Harman 2003a]. This research was supported by National Science Foundation Grant CCR-0305330 and Engineering and Physical Sciences Research Council Grants GR/R98938, GR/M58719, and GR/R71733. GrammaTech provided *CodeSurfer* upon which the implementation was based.

Authors' addresses: D. Binkley, Loyola College in Maryland, 4501 North Charles St., Baltimore, MD 21210; email: binkley@cs.loyola.edu; N. Gold, M. Harman, Kings College London, Strand, London WC2R 2LS, UK; email: {nicolas.gold,mark.harman}@kcl.ac.uk.

1. INTRODUCTION

A static program slice is a semantically meaningful portion of a program that captures a subset of the program's computation [Weiser 1984]. A slice is computed from a "slicing criterion" (a program point and variable of interest). Two kinds of slice are considered in this study: backward slices, which contain those parts of the program that potentially affect the slicing criterion, and forward slices, which contain those parts of the program potentially affected by the slicing criterion.

Slicing is a widely-applied technique because it allows a program to be simplified and thus allows attention to be focused on a subcomputation based on a user-selected criterion. Slicing has been applied to reverse engineering [Canfora et al. 1994b; Simpson et al. 1993], program comprehension [De Lucia et al. 1996; Harman et al. 2001], software maintenance [Canfora et al. 1994a; Cimitile et al. 1996; Gallagher 1992; Gallagher and Lyle 1991], debugging [Agrawal et al. 1993; Kamkar 1993; Lyle and Weiser 1987; Weiser and Lyle 1985], testing [Binkley 1997, 1998; Gupta et al. 1992; Harman and Danicic 1995; Hierons et al. 1999, 2002], component reuse [Beck and Eichmann 1993; Cimitile et al. 1995a], program integration [Binkley et al. 1995; Horwitz et al. 1989], and software metrics [Bieman and Ott 1994; Lakhotia 1993; Ott and Thuss 1993]. There are several surveys of slicing techniques, applications and variations [Binkley and Gallagher 1996; Binkley and Harman 2004; De Lucia et al. 2001; Harman and Hierons 2001; Tip 1995].

In all applications of slicing, the size of a slice is crucial: The more statements that can be determined to have no impact upon the slicing criterion, the better. The value of many slicing techniques depends directly on slice size. This leads to a natural question: Just how big *is* a typical program slice? This article seeks to answer this question for the C programming language. In order to provide a definitive answer, a large code base containing a variety of programs is studied. Five factors that influence slice size are also studied: the expansion of structure fields, inclusion of calling context, level of granularity of the slice, presence of dead code, and choice of points-to analysis.

The question of typical slice size is deceptively subtle. What constitutes a "typical" slicing criterion? This is rather subjective and thus hard to identify. In order to overcome the difficulties associated with (subjectively) choosing a selection of criteria, the article considers all possible slicing criteria, thereby guaranteeing to include all typical criteria. This is a time-consuming and demanding goal for an empirical study, only made possible by recent developments in slicing technology and a set of slice construction optimizations described elsewhere [Binkley and Harman 2003b]. Owing to the large size of the subject set and the fact that slices were constructed for every point in every program, the data can be used as a benchmark of static slice size, against which average slice size for different definitions of *typical* slicing criteria can be compared.

The results of the empirical study reported here make the following contributions:

—*Determination and evaluation of average slice size.* Results are presented concerning slice size for backward and forward slices over a large set of programs ranging in scale from several hundred lines of code to over 136,000 noncomment, nonblank lines of code. The slice on every executable statement was taken. Most previous studies concern subject programs of at least one order of magnitude smaller than those considered herein. Furthermore, all such studies take several orders of magnitude fewer slices. Thus, the present article presents the largest study of program slice size to date.

—*Evaluation of the impact of calling context on slice size.* Results are presented that evaluate the impact of calling context sensitivity upon slice computation time and size. Formally, the calling-context problem can be summarized as ensuring that after processing a call from call-site c, results are propagated to c and not to other call sites. Call strings have been used to track calling context [Horwitz et al. 1998]. In essence, these strings are built by labeling the call and return for call-site c_i with unique terminal symbols (e.g., "$(_i$" and "$)_i$") and then ensuring that the string of symbols produced by a graph traversal belongs to a specified context-free language [Reps et al. 1994; Reps 1998; Horwitz et al. 1998] (e.g., each "$(_i$" is matched with "$)_i$"). Few previous studies have addressed this question, and those which have, have produced contradictory results. The advent of slicing tools which, for pragmatic reasons, cannot fully account for calling context [Atkinson and Griswold 1998; Mock et al. 2002] makes this evaluation very timely. The findings suggest that slicing tools which fail to fully account for calling context have to be used with care.

—*Evaluation of the impact of slice granularity on slice size.* Results are presented concerning slices constructed at statement and function granularity. These findings suggest that function slices are typically one-third as large as their corresponding statement slices. In cases where a large program is to be sliced, this reduction in precision may be worthwhile because of the enormous decrease in slice construction time. Furthermore, statistical analysis reveals a strong linear correlation between function slice size and statement slice size. This suggests that function slices might be inexpensive predictors of statement slice size.

—*Evaluation of the impact of structure field expansion on slice size.* Results concerning the impact of collapsing structure fields on slice size suggest that for certain programs, this makes an important difference. Such programs contain structures with function pointers. This finding replicates those by Bent et al. [2000]. Overall, they lead to a 6–7% change in slice size.

—*Evaluation of the effect of points-to analysis.* A smaller-scale study is included that considers the difference between two possible choices for points-to analysis. Overall, points-to analysis precision had a relatively small effect on slice size, namely, of 1.1%.

—*Evaluation of the impact of dead code.* Finally, results are presented that indicate that dead code has little effect on average slice size. Over all 43 programs, removal of dead code reduced average slice size by about 2%.

The remainder of the article is organized as follows: Section 2 describe background information. Section 3 describes the four slicers used to generate the empirical data studied in Section 4, while Sections 5 and 6 describe threats to validity of the results and implications of the results for future slicing work and applications. Section 7 presents an account of related work and how it compares with the results reported here. Finally, Section 8 concludes.

2. BACKGROUND

This section presents a brief summary of program slicing, the subject programs studied, and the environment in which the study was conducted, including threats to the validity of the experiment.

2.1 Program Slicing

To compute the large number of slices required in this study, Grammatech's deep structure analysis tool CodeSurfer [Grammatech 2002] was used. Three other general-purpose program slicing tools exist: Blair and Associates, Inc. commercialized version of the NIST slicer, Unravel [Lyle et al. 1995] and Mock et al.'s Sprite tool [2002], both of which slice by solving (anew) data-flow equations for each slice, and Jayaraman et al.'s Java slicer [2005].

CodeSurfer builds an intermediate representation called the system dependence graph (SDG) [Horwitz et al. 1998]. The program slices studied in Section 4 are computed using graph reachability over the SDG. CodeSurfer's output goes through two preprocessing steps before slicing begins. The first identifies intraprocedural strongly connected components (SCCs) and replaces them with a single representative vertex. The key observation here is that any slice that includes a vertex from an SCC will include all the vertices from that SCC; thus, there is a great potential for saving effort by avoiding redundant work [Binkley and Harman 2003b]. Once discovered, SCC formation is done by moving all edges of represented vertices to the representative. The edgeless vertices are retained to maintain the mapping back to the source. While slicing, the slicer need never encounter them. The second preprocessing step reorders the vertices of each procedure into topological order. This is possible because cycles have been removed by the SCC formation. Topological sorting improves memory performance—in particular, cache performance [Binkley and Harman 2003b].

After preprocessing, two kinds of slices are computed [Horwitz et al. 1998]: backward and forward interprocedural slicing. Taken with respect to vertex v, a backward slice includes the vertices representing computations that potentially affect the computation represented at v. A forward slice is the dual of a backward slice. For statement s represented by vertex v, the forward slice on v includes those program elements potentially affected by the computation at s. In both cases the variables of interest are assumed to be those used by the vertex.

2.2 Subjects

The study considers just over one million lines of C code from 43 subject programs that range in size from 600 LoC (lines of code) to almost 180 KLoC. Figure 1 shows the 43 subject programs along with some statistics related to the programs and their SDGs. These statistics include four measures of

	Size (Loc)		Number	Number	Number	Pace (KloC/sec)	
Program	`wc`	`sloc`	of Vertices	of Edges	of Slices	`wc`	`sloc`
a2ps	63,600	40,222	707,623	1,488,328	58,281	546	345
acct-6.3	10,182	6,764	21,365	41,795	7,250	7,243	4811
barcode	5,926	3,975	13,424	35,919	3,909	5,663	3799
bc	16,763	11,173	20,917	65,084	5,133	7,078	4718
byacc	6,626	5,501	41,075	80,410	10,151	1,148	953
cadp	12,930	10,620	45,495	122,792	15,672	1,695	1392
compress	1,937	1,431	5,561	13,311	1,085	4,403	3253
copia	1,170	1,112	43,975	128,116	4,686	74	71
csurf-pkgs	66,109	38,507	564,677	1,821,811	43,044	472	275
ctags	18,663	14,298	188,856	405,383	20,578	640	490
cvs	101,306	67,828	8,949,186	28,033,287	103,265	29	19
diffutils	19,811	12,705	52,132	104,252	17,092	3,021	1938
ed	13,579	9,046	69,791	108,470	16,533	1,807	1204
empire	58,539	48,800	1,071,321	2,122,627	120,246	549	458
EPWIC-1	9,597	5,719	26,734	56,068	12,492	5,856	3490
espresso	22,050	21,780	157,828	420,576	29,362	793	783
findutils	18,558	11,843	38,033	174,162	14,445	3,532	2254
flex2-4-7	15,813	10,654	49,580	105,954	11,104	1,833	1235
flex2-5-4	21,543	15,283	55,161	234,024	14,114	1,178	836
ftpd	19,470	15,361	72,906	138,630	25,018	1,499	1183
gcc.cpp	6,399	5,731	26,886	96,316	7,460	1,995	1787
gnubg-0.0	10,316	6,988	36,023	104,711	9,556	1,768	1197
gnuchess	17,775	14,584	56,265	165,933	15,069	1,815	1489
gnugo	81,652	68,301	396,010	1,087,038	68,298	790	661
go	29,246	25,665	144,299	321,015	35,863	1,097	963
ijpeg	30,505	18,585	289,758	822,198	24,029	1,708	1041
indent	6,724	4,834	23,558	107,446	6,748	1,666	1197
li	7,597	4,888	1,031,873	3,290,889	13,691	15	10
named	89,271	61,533	1,853,231	8,334,948	106,828	69	48
ntpd	47,936	30,773	285,464	1,160,625	40,199	630	405
oracolo2	14,864	8,333	27,494	76,085	11,812	3,454	1936
prepro	14,814	8,334	27,415	75,901	11,745	3,589	2019
replace	563	512	1,406	2,177	867	9,417	8564
sendmail	46,873	31,491	1,398,832	10,148,436	47,344	30	20
space	9,564	6,200	26,841	74,690	11,277	2,347	1522
spice	179,623	136,182	1,713,251	6,070,256	212,621	43	33
termutils	7,006	4,908	10,382	23,866	3,113	12,997	9105
tile-forth-2.1	4,510	2,986	90,135	365,467	12,076	191	126
time-1.7	6,965	4,185	4,943	12,315	1,044	17,228	10351
userv-0.95.0	8,009	6,132	71,856	192,649	12,517	302	232
wdiff.0.5	6,256	4,112	8,291	17,095	2,421	13,562	8914
which	5,407	3,618	5,247	12,015	1,163	18,101	12112
wpst	20,499	13,438	140,084	382,603	20,889	1,557	1021
sum	1,156,546	824,935	19,865,184	68,645,673	1,210,090		
average	26,896	19,185	461981	1,596,411	28,142	3,336	2,285

Fig. 1. The subject programs studied.

program size: first, column two gives the size of each program as reported by the Unix word-count utility `wc`. Line counts, especially using the word count, provide a rather crude method. The data is included to provide a consistent measure of program size that facilitates comparison with prior studies. To provide a better estimate of program size, the next column reports the number of nonblank noncomment lines of code as reported by sloc_count [Wheeler 2005]. The next two columns provide a measure of the dependence complexity in the program by reporting the number of vertices and edges, respectively, in each SDG.

In the SDG, global variables are modeled as value-result parameters; thus, corresponding vertices are added at call sites and procedure entry. Each such "global pseudoparameter" counts as a node in the SDG. This is why it is important to measure size in LoC and SLoC and not in SDG nodes. As can be seen in Figure 1, the number of nodes per line of code can vary dramatically. For example, consider the two programs compress and copia. The large number of nodes per line of code for copia (compared to compress) comes from the fact that it has twice as many globals, 20 times the number of calls, and 10 times the number of functions.

The sixth column of Figure 1 reports the number of backward slices taken (the same number of forward slices was taken) for each subject program. At first glance, we might expect the number of slices to equal the number of nonblank noncomment source lines. The two do not match because a slice was computed for each vertex that represented executable code. It turns out that one "line of code" may be represented by multiple vertices. For example, the line "for(i=0; i<10; i++)" generates three vertices, representing initialization, test, and increment of i. In general, each SDG vertex contains, at most, one side-effect, and call parameters are separated in different nodes [Horwitz et al. 1998]. For example, the dependence subgraph for the function "f(int *a)" includes an *entry* vertex labeled "f" and three parameter vertices. Two represent the incoming value of "a" and "*a" and one represents the result value of "*a".

The subject programs cover a wide range of programming styles. For example, the program prepro is Fortranesque in its use of arrays. In contrast, several other programs make heavy use of function pointers. The program ed is rather "single-minded," while the program acct contains many different (although related) computations. Studying 43 different programs provides sufficient diversity to draw general conclusions about the effectiveness of program slicing from the results of this study.

2.3 Study Construction Environment

The experimental process was fairly straightforward. Data was collected by slicing on every vertex that represents source code using each of four slicer configurations. The data was collected on a dual processor computer running Red Hat Linux 7.1 and kernel version 2.4.2-2. Each processor was a 1000MHz Pentium III and had a 256KB cache. The processors share 4GB of main memory. Memory contention between the two processors on the shared memory bus slowed each processor down by 15.5% relative to a single processor in an identical environment; thus, the effective single CPU speed was 845MHz. CPU utilization was $99^{+}\%$ for all runs of the slicer. The slicer was built using the GNU gcc compiler version 2.96, using the -O3 optimization flag.

The times reported in this article are for slicing only. The construction of the SDG and the preprocessing done by the slicer are excluded. Slice times reported represent the time taken to compute all forward and backward slices. This is a considerable number of slices and much work went into optimizing the slicing algorithm to speed up the computation of numerous slices from a single program. This work is reported elsewhere [Binkley and Harman 2003b].

Though slicer pace is not the primary focus of this article, for the reader's information, the final two columns of Figure 1 present the pace of the slicing computation, which is computed by dividing the number of vertices processed per second by the average 14.1 vertices per line of code. The pace is given for both line-count measures.

It is interesting to note that there is a rather large variation in slice pace, which ranges from 10 KLoC to over 12 MLoC per second. There are two coupled causes for this variation: The SDG includes vertices that represent global variables as additional function parameters. Thus, programs with large numbers of both globals and function calls include a high proportion of such vertices. The processing time for an edge is a near-constant 60 nanoseconds; however, programs with large numbers of globals and functions (which is correlated with a large number of function calls) have a considerably higher number of edges per vertex. As the addition of a single global variable (i.e., a single line of code) can introduce many new vertices, LoC does not make a good predictor of slice pace.

Slice size could have been reported as a percentage of the vertices in the SDG (effectively the same vertex set as in a CFG, with the addition of pseudovertices for representation of global variables entering and leaving a procedure). The problem with doing this is that it makes the results hard to compare with other studies of slicers that use a different or no underlying graph. Even with SDG-based slicing, there is variation in the number of vertices that a program's SDG has. For example, the process of collapsing structure fields changes the number of vertices in the graph (but not the number of lines of code in the program). Similarly, a recent enhancement to CodeSurfer added a linear number of vertices in exchange for removing a quadratic number of edges. Such changes make the "percentage of vertices in a graph" a moving and poorly-defined target. Finally, line count was also chosen for a pragmatic reason: Ultimately, a tool user is going to want to know which source lines are relevant to a query. Thus, it is the percentage of these lines that is relevant to a software engineer.

3. EXPERIMENTAL SETUP

Experiments were conducted to explore the effects of five factors that may have an influence upon slice size. The aim of the work is to provide benchmark data on slice size and to explore these influencing factors. The five factors studied were:

(1) the treatment of calling context,
(2) the level of granularity,
(3) the treatment of structure fields,
(4) the choice of points-to analysis, and
(5) the effect of dead code on slice size.

These aspects form a set of options available to a slicer. Some of them, notably points-to analysis and calling context, have been widely-studied in previous empirical work on factors that influence slice size [Binkley and Harman 2004]. The first two aspects represent changes to the actual slicing algorithm. Thus, the three slicers described in the following table are used for the experiments.

Slicer	Calling Context	Granularity
s_1	respected	statement level
s_2	ignored	statement level
s_3	respected	function level

Slicer s_1 is expected to be the most precise, with s_2 allowing the impact of calling context to be researched and s_3 allowing the granularity of a slice to be studied. The remaining three aspects affect the construction of the underlying SDG to be sliced. Slicer s_1 is used to explore the effect on slice size of the impact, or collapsing structure fields, of pointer analysis precision and dead code removal. Thus five empirical comparisons are reported in the next section. This section first describes each comparison in greater detail.

3.1 Calling Context: Respected or Ignored

Slicers s_1 and s_2 allow the treatment of *calling context* to be studied. When slicing, "respecting calling context" refers to the task of tracking call sites to determine the point at which to resume slicing after a called procedure has been sliced. A slicer that ignores calling context treats the calls to a given procedure as indistinguishable, so that a call from one call site is treated as a potential call from all the others. Though safe, this has the potential to dramatically increase slice size. Respecting calling context requires matching an appropriate call site with a return, for example, using a stack. However, the stack approach is somewhat inefficient. The "summary edges" approach of the SDG allows the two-pass algorithm to efficiently track calling context [Horwitz et al. 1998; Reps et al. 1994].

3.2 Granularity: Statement or Function Level

Slicers s_1 and s_3 allow the effect of "slice granularity" to be studied. Traditionally, slicing operates at the statement level of granularity: A slice may include or exclude individual statements. However, as an engineer will typically either consider or not consider the code from a procedure (especially when procedure size is not excessive), an alternative to slicing at the statement level is to slice at the function level, where entire functions are either included or excluded.

Function-level slicing is expected to be much quicker and easier to perform than statement-level slicing, which makes it a good (and perhaps sufficient) first approximation. There are interesting questions as to how much is given up in terms of precision and how much of an impact this has on a programmer. The precision question is addressed in the next section, while the impact question is a topic for future investigation.

	Collapsed (sec)	Expanded (sec)	Decrease (percent)
backward slice			
total	885,328	520,052	41%
per slice	7,607	5,664	26%
forward slice			
total	743,542	557,758	25%
per slice	6,654	5,989	10%

Fig. 2. Total analysis time for slicer s_1. The values given per slice are averaged from the total values. The average per-slice reduction is thus the average reduction on the average slice.

3.3 Structure Fields: Expanded or Contracted

Slicer s_1 is used to study the effect of collapsing structure fields [Yong et al. 1999]. This is best understood by way of example. Consider the following code:

```
struct file_ops
{
    int (*write_fn)();
    int (*read_fn)();
} file1;
```

Assume that `write_fn` potentially points to the two functions `write_ext2` and `write_nfs`, and `read_fn` potentially points to the functions `read_ext2` and `read_nfs`. With structure fields expanded, the function call `(*file1.readfn)(···)` potentially calls `read_ext2` and `read_nfs`. With structure fields collapsed, all structure fields are treated as a single variable. Collapsing `write_fn` and `read_fn` means that the call `(*file1.readfn)(···)` appears to call `write_ext2` and `write_nfs` in addition to `read_ext2` and `read_nfs`.

For programs with structures that contain pointers, especially function pointers, this collapse has the potential for a significant impact on slice size. Thus, in the study each SDG was built twice, once with structure fields collapsed and once with them expanded.

The effect of collapsing structure fields can also be seen on the underlying SDGs. For example, the data in Figure 1 is for expanded-field SDGs. Here, the sum of the number of vertices and edges in all graphs are 19 and 68 million, respectively. The corresponding sums when structure fields are collapsed are 14 and 140 million. The following table summarizes the effect on computation time of this mild decrease in vertices, but rather dramatic increase in edges. By collapsing fields, nodes become merged. This reduces the number of nodes in the graphs (from 19 to 14 million). However, by merging nodes, additional (spurious) dependencies are created. This is reflected by additional edges in the SDGs (the edges rise from 68 to 140 million). This extra dependence means that the slices will be larger (will include more statements).

3.4 Points-To Analysis

Points-to algorithms (e.g., Andersen [1994], Fahndrich et al. [1998], and Heintze and Tardieu [2001]) can be flow- and context-sensitive or insensitive

[Anderson et al. 2001; Shapiro and Horwitz 1997]. Presently, neither context [Callahan 1998; Ruf 1995; Wilson and Lam 1995] nor flow-sensitive analyses [Choi et al. 1993; Emami et al. 1994; Landi and Ryder 1992] scale to the size of the programs considered herein. In the other direction, very imprecise points-to information negatively affects most data-flow analyses, including program slicing. For example, in the extreme case, all pointers are assumed to point to all variables.

Between these extremes lies the spectrum [Shapiro and Horwitz 1997] of context- and flow-insensitive pointer analyses [Andersen 1994; Burke et al. 1995; Steensgaard 1996; Zhang et al. 1996]. At one end of this spectrum, Steensgaard's algorithm [1996], for example, merges all the nodes pointed to by a given pointer. In contrast, Andersen's algorithm [1994] leaves these nodes expanded. The loss of precision present in Steensgaard's algorithm causes a corresponding increase in slice size. For example, with the program `ijpeg`, the use of Steensgaard's pointer analysis led to an average slice size of 2.5 times that computed using Andersen's algorithm. Foster et al. [2000] proposed a context-sensitive (polymorphic) version of Andersen's algorithm, but found that it did not significantly improve on the monomorphic version.

Slicer s_1 is used to consider the impact of points-to analysis precision. The original dependence graphs were configured and constructed using Andersen's context- and flow-insensitive points-to algorithm [1994], which scales to the size of the programs considered in the study. Section 4.4 provides a smaller-scale study that compares the use of Andersen's algorithm with slices of SDG built using Steensgaard's [1996] algorithm.

3.5 Dead Code

Programs can contain dead code (code which is not executed for any possible initial state). Such dead code may be removed from the construction of any slice. If a program were to contain a large amount of dead code then the slices of the program would necessarily be (somewhat artificially) small. This might produce surprising results and impede the goal of providing benchmark data on slice size for a large set of programs. Programs can contain dead code for a variety of reasons. For example, it is possible that the code is switched on by a flag in order to debug the behavior of the program. Rather than removing the debugging code before release, the debug flag is simply set to `false`, making parts of the program inaccessible.

Without a study of the specific question of the level of dead code in the programs researched, it is not possible to assess its impact upon the findings. Of course, the problem of whether a statement is dead (unreachable) is not decidable [Weyuker 1977]. Fortunately, the dead code in a program can be safely approximated by slicing forward from the start node of the main function. Any nodes not included are considered dead code. Any node not included in this way will not be in any slice. Therefore, it makes sense to check whether there is a large number of these dead code nodes, as this would artificially reduce the size of all slices. In the next section, the impact of this kind of dead code is studied as a separate question in its own right.

4. RESULTS

This section presents data comparing slices from the three slicers along with an investigation of the effect of structure field collapse, points-to analysis, and dead code removal. To begin with, the baseline data for Slicer s_1 is provided in Figure 3. For each program, the figure presents the size of the program and the average slice size for backward and forward slicing. Averages were computed from all slices taken with respect to vertices representing executable code (the number of slices computed for each program are presented in Figure 1).

As seen in the last row of the table, for Slicer s_1 over all 43 programs, the average backward slice contained 28.1% of the program and the average forward slice contained 26.1% of the program. One way to appreciate the assistance that program slicing provides is to take a look at raw code sizes. Consider a program whose average size is near in average size for the entire collection. For example, `ijpeg` contains about 20 KLoC and has an average slice size of 31%, or 5,761 LoC. Clearly, the complexity of comprehending, testing, debugging, or maintaining 5,761 LoC is lower than that of comprehending, testing, debugging, or maintaining 20 KLoC.

Each average is given in terms of lines of code and as a percentage of lines of code from the original program. To get a better understanding of the variation in slice size, the averages are followed by standard deviations in slice sizes (expressed as percentages). These standard deviations indicate the variation in individual slice sizes for each program. There is also a notable variation in the *level of variation* of slice sizes.

The range of the average slice size is striking. For example, the low for backward slicing is 7.4% (for program `acct-6.3`) and its high is 61.7% (for program `go`). Taken together with the standard deviations, these results suggest two things:

(1) Although this article presents results on average slice size that are relatively "stable" as a benchmark (due to the large number of programs studied), they can be no more than a benchmark against which other slicing results can be compared. For example, they allow us to say that a program has "an unusually large (respectively, small) set of slice sizes" or that a particular program slice is "unusually large (respectively, small)."

(2) The variation in average slice size between programs and within the set of slices for a given program, together with the variation in standard deviations for the set of slices of a given program, suggest that a number of slice-based measurements may be worth considering. For example, the authors have argued that the phenomenon of programs with large numbers of slices of identical size can be an indication of "dependence pollution," that is, avoidable clusters of dependence, consisting of a large set of mutually interdependent nodes. This has been studied in more detail elsewhere [Binkley and Harman 2005], where slicing is used to identify dependence pollution and refactoring is used to remove it. In addition, slicing has previously been used as the basis for a definition of analytical code-level software metrics [Bieman and Ott 1994; Harman et al. 1997; Ott and Thuss 1989;

	Backward Slices			Forward Slices		
	Average Size		Standard	Average Size		Standard
Program	LoC	Percent	Deviation	LoC	Percent	Deviation
a2ps	12,348	30.7%	15.8%	11,624	28.9%	23.8%
acct-6.3	501	7.4%	7.7%	487	7.2%	9.6%
barcode	1,212	30.5%	19.9%	1,141	28.7%	21.5%
bc	5,452	48.8%	12.4%	5,173	46.3%	23.5%
byacc	996	18.1%	13.3%	902	16.4%	20.6%
cadp	828	7.8%	8.7%	765	7.2%	10.3%
compress	356	24.9%	26.0%	323	22.6%	18.0%
copia	252	22.7%	3.6%	182	16.4%	17.5%
csurf-packages	6,007	15.6%	14.0%	5,738	14.9%	14.1%
ctags	5,948	41.6%	17.5%	5,619	39.3%	24.9%
cvs	31,404	46.3%	11.8%	29,912	44.1%	23.4%
diffutils	2,871	22.6%	20.0%	2,668	21.0%	18.5%
ed	4,822	53.3%	15.1%	4,333	47.9%	29.9%
empire	16,104	33.0%	15.9%	13,762	28.2%	24.1%
EPWIC-1	646	11.3%	12.0%	618	10.8%	12.1%
espresso	6,708	30.8%	17.0%	6,512	29.9%	26.3%
findutils	3,257	27.5%	17.2%	2,949	24.9%	22.1%
flex2-4-7	2,642	24.8%	18.7%	2,365	22.2%	19.8%
flex2-5-4	3,194	20.9%	14.2%	2,797	18.3%	21.5%
ftpd	5,361	34.9%	16.9%	4,808	31.3%	24.3%
gcc.cpp	2,625	45.8%	18.2%	2,424	42.3%	28.0%
gnubg-0.0	1,558	22.3%	16.1%	1,761	25.2%	18.9%
gnuchess	6,271	43.0%	21.7%	5,877	40.3%	28.0%
gnugo	23,769	34.8%	20.7%	19,329	28.3%	20.1%
go	15,835	61.7%	19.8%	15,245	59.4%	21.2%
ijpeg	5,761	31.0%	21.5%	5,576	30.0%	24.4%
indent-1.10.0	2,117	43.8%	25.9%	1,938	40.1%	24.5%
li	1,872	38.3%	19.2%	1,823	37.3%	21.0%
named	23,075	37.5%	15.2%	21,352	34.7%	25.2%
ntpd	8,278	26.9%	17.9%	8,278	26.9%	20.0%
oracolo2	983	11.8%	9.0%	1,017	12.2%	18.3%
prepro	967	11.6%	8.9%	1,000	12.0%	18.1%
replace	111	21.6%	14.0%	111	21.7%	17.8%
sendmail	10,487	33.3%	16.5%	9,542	30.3%	24.2%
space	800	12.9%	9.3%	818	13.2%	19.2%
spice	29,688	21.8%	17.2%	29,007	21.3%	19.8%
termutils	1,232	25.1%	17.7%	1,134	23.1%	17.7%
tile-forth-2.1	1,550	51.9%	35.6%	1,424	47.7%	23.0%
time-1.7	347	8.3%	8.7%	268	6.4%	9.0%
userv-0.95.0	1,190	19.4%	16.6%	1,079	17.6%	17.9%
wdiff.0.5	419	10.2%	9.8%	395	9.6%	10.3%
which	1,075	29.7%	23.2%	980	27.1%	18.6%
wpst	1,478	11.0%	12.6%	1,424	10.6%	13.0%
sum	252,398			234,481		
min	111	7.4%	3.6%	111	6.4%	9.0%
max	31,404	61.7%	35.6%	29,912	59.4%	29.9%
average	5,870	28.1%	16.1%	5,453	26.1%	20.1%

Fig. 3. Slice size averages and standard deviations for backward and forward interprocedural slices, computed using Slicer s_1.

Meyers and Binkley 2004]. These forms of observation about slice size and variations in slice size might lead to improved slice-based metrics.

4.1 The Effect of Calling Context

Slicer s_2 differs from Slicer s_1 in that it ignores calling context. Several other studies have reported on the impact of calling context on slice size [Agrawal and Guo 2001; Atkinson and Griswold 1998; Krinke 2002]. However, the data presented herein represents by far the largest, most general such study to-date. This data is compared with previous studies in the Related Work section.

In short, ignoring calling context saves the time required to set-up Pass 2 of the two-pass slicing algorithm. However, it produces larger slices (the slices can be no smaller, and thus there is never a reduction in the number of statements encountered). Furthermore, increasing slice size brings an increase in work and thus takes proportionately *more* time to discover the statements of the slice.

Figure 4 shows this comparison for each program. Over all programs, the average slice size increase is 50% and ranges from 0.17% for `copia` (rounded to 0% in the figure) to 330% for `acct`. The weighted average is smaller: 21% for backward and 28% for forward slicing. For most programs, the size increase is the dominant effect, and slicing time increases. The most dramatic example of this is `acct`, whose average slice size more than quadruples and whose backward slicing time more than quintuples. This is due to the fact that `acct` contains a lot of separate functionality using some common code, but with little overlap otherwise. Consequently, context-insensitive slicing includes multiple functionalities, but these are separated when context is respected.

However, with a few of the programs, the overhead of tracking calling context dominates and the slicing time actually decreases, even though the resulting slices are larger (these programs have negative time "increases" in Figure 4). In the worst case, Pass 2 must revisit every vertex (statement), effectively doubling the work. In practice, `li` shows the most dramatic change: a 29% *reduction* in computation time. An examination of the code shows that `li` contains a large number of small functions. This serves to increase the time spent on the interface between Pass 1 and Pass 2 because the interface relates to procedure boundaries. A closer examination, using multiple runs with a profiler, shows an average of 16% reduction in computation time. Three main factors account for this. First, the test to see if a particular edge is traversed in a given pass, which is unneeded when calling context is ignored, is executed over 4 billion times when processing `li`. This accounts for half of the time increase. The other half comes from the interface between Pass 1 and Pass 2, with the simple linked list code that the slicer uses between passes accounting for 20% of the overall time difference.

4.2 Effect of Level of Granularity

To assess the level of correlation between statement- and function-level slice size, two statistical tests, Spearman and Pearson correlations, were used. Pearson's linear correlation attempts to construct quantitative linear models from sets of data. The output of a Pearson correlation is a *correlation coefficient*,

	Average Backward Slice Size			Average Forward Slice Size			Time Increase	
Program	s_1: Respect CC	s_2: Ignore CC	Incr ease	s_1: Respect CC	s_2: Ignore CC	Incr ease	Backward Slicing	Forward Slicing
a2ps	12,348	15,925	29%	11,624	15,330	32%	69%	51%
acct	501	2,151	330%	487	1,896	289%	418%	301%
barcode	1,212	1,647	36%	1,141	1,620	42%	21%	19%
bc	5,452	5,499	1%	5,173	5,193	0%	8%	-4%
byacc	996	1,590	60%	902	1,823	102%	132%	160%
cadp	828	2,087	152%	765	1,967	157%	248%	231%
compress	356	379	6%	323	329	2%	-25%	0%
copia	252	253	0%	182	183	0%	-5%	6%
csurf-pkgs	6,007	9,492	58%	5,738	8,821	54%	76%	59%
ctags	5,948	7,176	21%	5,619	6,072	8%	116%	85%
cvs	31,404	34,920	11%	29,912	31,199	4%	23%	-13%
diffutils	2,871	6,125	113%	2,668	5,204	95%	194%	141%
ed	4,822	4,865	1%	4,333	4,392	1%	19%	15%
empire	16,104	18,803	17%	13,762	15,110	10%	101%	52%
EPWIC-1	646	1,892	193%	618	1,833	197%	258%	242%
espresso	6,708	10,493	56%	6,512	9,516	46%	124%	103%
findutils	3,257	6,100	87%	2,949	5,237	78%	169%	118%
flex2-4-7	2,642	3,543	34%	2,365	3,374	43%	98%	68%
flex2-5-4	3,194	4,434	39%	2,797	4,075	46%	81%	53%
ftpd	5,361	5,872	10%	4,808	5,193	8%	59%	42%
gcc.cpp	2,625	2,910	11%	2,424	2,674	10%	6%	-1%
gnubg-0.0	1,558	2,152	38%	1,761	2,257	28%	76%	58%
gnuchess	6,271	7,082	13%	5,877	6,400	9%	56%	49%
gnugo	23,769	25,998	9%	19,329	20,620	7%	56%	55%
go	15,835	15,873	0%	15,245	15,352	1%	40%	37%
ijpeg	5,761	6,709	16%	5,576	6,627	19%	14%	-6%
indent-1.10	2,117	2,566	21%	1,938	2,251	16%	-7%	35%
li	1,872	1,875	0%	1,823	1,827	0%	-29%	-29%
named	23,075	24,559	6%	21,352	22,934	7%	-12%	-7%
ntpd	8,278	9,743	18%	8,278	8,914	8%	60%	49%
oracolo2	983	1,715	74%	1,017	2,143	111%	120%	152%
prepro	967	1,694	75%	1,000	2,108	111%	123%	156%
replace	111	132	19%	111	133	20%	0%	-50%
sendmail	10,487	11,142	6%	9,542	10,046	5%	-28%	-24%
space	800	1,408	76%	818	1,750	114%	120%	160%
spice	29,688	34,426	16%	29,007	31,600	9%	4%	-9%
termutils	1,232	2,247	82%	1,134	1,950	72%	73%	38%
tile-forth	1,550	1,563	1%	1,424	1,444	1%	-1%	3%
time-1.7	347	401	15%	268	320	20%	0%	-67%
userv-0.95.	1,190	1,933	63%	1,079	1,656	53%	99%	96%
wdiff.0.5	419	1,030	146%	395	1,005	155%	100%	113%
which	1,075	1,503	40%	980	1,340	37%	0%	-20%
wpst	1,478	3,537	139%	1,424	3,683	159%	274%	304%
sum	252,398	305,443		234,481	277,399		3327%	2823%
average	5,870	7,103	50%	5,453	6,451	51%	77%	66%
weighted ave			21%			18%		

Fig. 4. Data for s_2 shows the effect of ignoring calling context on slice size and computation time.

reported as the value R, and the coefficients of a linear model $y = mx + b$. The statistical significance of the results, as captured by the reported R-value, can be summarized as follows:

0.8–1.0 strong association
0.5–0.8 moderate association
0.0–0.5 weak or no association

A negative value indicates an inverse correlation.

Dependent Variable	Independent Variable	Pearson R	Pearson m	Pearson b	Spearman R
SL average size	Statement count	0.996	0.736	-33690	0.965
SL average size	FL average size	0.697	2110	-237900	0.963
SL average size	Function count	0.529	1190	-168000	0.875
FL average size	Statement count	0.728	0.00018	182	0.956
FL average size	Function count	0.916	0.68070	-15.7	0.924

Fig. 5. Correlations between function and statement slice average sizes (SL denotes a statement-level slice and FL a function-level slice).

The Spearman test does not attempt to construct a linear correlation between the two variables, but merely gives a correlation coefficient indicating whether increases in one variable are accompanied by increases in the other. The presence of a Pearson correlation is always accompanied by that of a Spearman correlation. However, it is possible for data to have a strong Spearman correlation without having a strong Pearson correlation. When this happens, it gives some evidence that there may be a nonlinear increase in the dependent variable corresponding to increases in the independent variable.

Various attempts were made to construct linear regression models to explain the data. No statistically significant correlation was found between the impact of calling context and several structural features of the code studied, including number of global variables, number of functions, and number of calls per function. The only models found to provide a strong correlation were those that related the impact of calling context and the number of calls in the program. Of course, this is a somewhat unsurprising correlation. The model is

```
additional Sloc per slice = 0.58 * calls + 467
```

with $R = 0.75$, $R^2 = 0.54$ (i.e., just over half of the increase in slice size when ignoring context is explained by the number of calls). In the model, `calls` plays a significant role, with a p value < 0.0001. However, this was the only correlation found, and it is an unsurprising one.

The data for Slicer s_3 is shown in Figure 7. Here an entire procedure is collapsed into a single vertex. Obviously, collapsing the dependence graph for a procedure to a single node greatly speeds-up the computation of slices. For example, the program `spice` has approximately 2,500 functions and 1.75 million vertices. However, at the function level of abstraction, the graph to be sliced has one vertex per function, so slicing `spice` at this level of abstraction is a similar problem to statement-level slicing of the very smallest of programs considered in the study. As the data shows, this gain in speed is achieved at the expense of precision.

Slice size can be measured in absolute terms using the number of statements or functions in a slice. It can also be measured in relative terms as a proportion of the statements (respectively, functions) in the slice. The correlation between both these measures, as well as between them and the size of the program (measured in statements and functions), is presented.

Figure 5 gives the correlations between statement and function slice sizes. As may be expected, the size of a slice is linearly correlated to the number of

Dependent Variable	Independent Variable	Pearson R	m	b	Spearman R
SL average size	Statement count	0.449	0	0.4	0.694
SL average size	FL average size	0.848	0.749	0.021	0.853
SL average size	Function count	0.440	0.00023	0.3	0.539
FL average size	Statement count	0.364	0.00000	0.5	0.493
FL average size	Function count	0.245	0.00015	0.5	0.315

Fig. 6. Correlations between function and statement slice average proportions (SL denotes a statement-level slice and FL a function-level slice).

statements in the program; bigger programs have bigger slices. The correlation is stronger ($R = 0.996$ compared to $R = 0.728$) at the statement level of granularity, though the rank correlation is equally strong for both levels. This can be accounted for by the granularity "mismatch" embodied in attempting to predict the size of a function slice in terms of statements. Observe that the strength of linear correlation of the size of a function slice, as predicted by the number of functions, is much greater ($R = 0.916$).

Figure 6 presents correlations between statement and function slice proportions. For these data there is no evidence of a linear correlation, apart from the correlation between the average percentage of statements in a slice and average percentage of functions in a slice (Row 2 of the figure). The strong linear fit indicates that the trend is for the percentage of statements to be about three-quarters of that of functions. This arises because of the coarseness of a function-level slice.

Notice also that there is some evidence for a nonlinear trend in the growth of the average proportion of statements in a statement slice against the number of statements and the number of functions. For these, the best fit comes from an exponential model, but it must be stressed that no fit gives a strong correlation (including rank correlation).

In addition to measuring function-level slice size in terms of functions, Figure 7 includes the size of function-level backward slices measured in non-comment nonblank (LoC). Depending on whether a function-level slice includes predominately small or large functions, measuring slice size at the statement level can include a larger or smaller percentage of a program than measuring at the function level. However, when averaged over all slices, these two effects cancel each other: In half of the programs, slices favor smaller functions (i.e., counting lines of code produces a smaller percentage) and in the other half, larger functions are favored.

This difference is not dramatic for most programs: For two of three programs the difference is less than 10%. The largest change in favor of small functions (for `ijpeg`) is −16.1%. The largest change in favor of large functions (for `termutils`) is 30.9%. This is indicative of a general pattern in which measuring function-level slice size using SLoC reports smaller slices. This behavior occurs with programs that include a few large functions which are not in most slices.

Function slices can be computed in almost negligible time. Since they are only one-third larger than statement-level slices, they may be useful surrogates for finer-grained (and more precise) statement-level slices (particularly for larger

		backward				forward			
program	slices	average functions per slice	percent of functions	average sloc per slice	percent of sloc	average functions per slice	percent of functions	average sloc per slice	percent of sloc
a2ps	58,281	600.5	57.1%	27,209	67.6%	449.7	42.7%	18,972	47.2%
acct	7,250	20.2	19.4%	1,058	15.6%	13.8	13.2%	663	9.8%
barcode	3,909	41.8	63.4%	1,865	46.9%	27.4	41.6%	1,317	33.1%
bc	5,133	90.7	82.5%	8,306	74.3%	82.4	74.9%	7,018	62.8%
byacc	10,151	80.2	35.8%	2,256	41.%	36.2	16.2%	765	13.9%
cadp	15,672	81.1	18.%	2,001	18.8%	48.7	10.8%	1,068	10.1%
compress	1,085	8.5	35.3%	280	19.6%	5.4	22.5%	173	12.1%
copia	4,686	235.5	96.9%	1,008	90.7%	112	46.1%	507	45.6%
csurf-packages	43,044	475.8	34.9%	13,909	36.1%	366.8	26.9%	10,238	26.6%
ctags	20,578	407	78.0%	11,165	78.1%	335.4	64.3%	8,837	61.8%
cvs	103,265	1063.5	90.0%	60,603	89.3%	895.7	75.8%	49,973	73.7%
diffutils	17,092	88	32.5%	3,909	30.8%	71.5	26.4%	3,613	28.4%
ed	16,533	193.9	85.4%	7,569	83.7%	149.9	66.0%	5,878	65.0%
empire	120,246	635.8	76.0%	39,408	80.8%	459.1	54.8%	27,761	56.9%
epwic-1	12,492	30.4	16.0%	993	17.4%	23.3	12.3%	768	13.4%
espresso	29,362	328.9	61.4%	13,864	63.7%	240.8	44.9%	10,072	46.2%
findutils	14,445	147.9	49.6%	4,515	38.1%	112.7	37.8%	3,812	32.2%
flex2-4-7	11,104	75.1	49.4%	5,416	50.8%	64	42.1%	3,617	33.9%
flex2-5-4	14,114	81.7	36.3%	7,016	45.9%	72.8	32.3%	4,094	26.8%
ftpd	25,018	183	71.5%	11,725	76.3%	128.6	50.2%	8,426	54.9%
gcc.cpp	7,460	66.1	66.8%	3,656	63.8%	57.5	58.1%	2,967	51.8%
gnubg-0.0	9,556	84.9	39.3%	3,053	43.7%	69.2	32.0%	2,431	34.8%
gnuchess	15,069	86.7	59.0%	9,237	63.3%	83.9	57.1%	7,189	49.3%
gnugo	68,298	537.2	73.4%	45,137	66.1%	494	67.5%	40,725	59.6%
go	35,863	336.1	90.4%	22,095	86.1%	318.9	85.7%	21,012	81.9%
ijpeg	24,029	194.8	40.5%	10,515	56.6%	180.9	37.6%	9,276	49.9%
indent-1.10.0	6,748	33	71.6%	2,647	54.8%	21	45.7%	1,961	40.6%
li	13,691	291.2	80.0%	3,443	70.4%	265.4	72.9%	3,154	64.5%
named	106,828	777.8	58.6%	36,685	59.6%	777.8	58.6%	35,651	57.9%
ntpd	40,199	271	64.8%	16,140	52.4%	219.3	52.5%	12,894	41.9%
oracolo2	11,812	56.2	25.7%	3,122	37.5%	37.4	17.1%	1,508	18.1%
prepro	11,745	55.1	25.1%	3,089	37.1%	35.4	16.2%	1,482	17.8%
replace	867	10.5	49.8%	269	52.6%	7.1	33.7%	172	33.6%
sendmail-8.7.5	47,344	218.9	71.5%	24,422	77.6%	167.7	54.8%	18,045	57.3%
space	11,277	58	34.5%	2,467	39.8%	36.6	21.8%	1,179	19.0%
spice	212,621	665.5	40.9%	44,944	33.0%	590.1	36.2%	31,559	23.2%
termutils	3,113	31.1	47.1%	1,595	32.5%	29	43.9%	1,294	26.4%
tile-forth-2.1	12,076	198.2	66.3%	1,056	35.4%	238.8	79.9%	1,297	43.4%
time-1.7	1,044	5.3	37.8%	735	17.6%	2.1	14.7%	308	7.4%
userv-0.95.0	12,517	99.7	41.0%	2,579	42.1%	70.4	29.0%	1,780	29.0%
wdiff.0.5	2,421	8.7	20.3%	729	17.7%	6	14.0%	449	10.9%
which	1,163	7.3	48.5%	782	21.6%	7.6	50.6%	676	18.7%
wpst	20,889	140.1	22.5%	3,659	27.2%	104.7	16.8%	2,419	18.0%
sum	1,210,090	9,102.90		466,134		7,517.00		367,000	
average	28,142	211.7	52.7%	10,840	50.1%	174.8	41.8%	8,535	37.4%

Fig. 7. Function level slice sizes. Slice slices are reported both in functions and SLoC.

programs). Furthermore, the strong linear correlation between statement- and function-level slice size means that function-level slices be useful predictors of statement-level slice sizes. In this way, a programmer may first construct function-level slices from a large procedure and then choose from these a set of interesting slices to examine at the statement level. The overall computation cost of this process would be significantly reduced.

4.3 Effect of Structure Field Collapse

Figure 8 shows the slice size data gathered using Slicer s_1 with SDGs built with structure fields both expanded and collapsed. The figure presents the size of

	Structure Fields Expanded				Structure Fields Collapsed				Percent Increase	
	Average Backward Slice Size		Average Forward Slice Size		Average Backward Slice Size		Average Forward Slice Size		Backward	
Program	LoC	Percent	LoC	Percent	LoC	Percent	LoC	Percent		Forward
a2ps	12,348	30.7%	11,624	28.9%	12,791	31.8%	11,946	29.7%	3.6%	2.8%
acct-6.3	501	7.4%	487	7.2%	507	7.5%	487	7.2%	1.4%	0.0%
barcode	1,212	30.5%	1,141	28.7%	1,260	31.7%	1,181	29.7%	3.9%	3.5%
bc	5,452	48.8%	5,173	46.3%	5,486	49.1%	5,195	46.5%	0.6%	0.4%
byacc	996	18.1%	902	16.4%	996	18.1%	902	16.4%	0.0%	0.0%
cadp	828	7.8%	765	7.2%	881	8.3%	765	7.2%	6.4%	0.0%
compress	356	24.9%	323	22.6%	356	24.9%	323	22.6%	0.0%	0.0%
copia	252	22.7%	182	16.4%	252	22.7%	182	16.4%	0.0%	0.0%
csurf-packages	6,007	15.6%	5,738	14.9%	6,046	15.7%	5,776	15.0%	0.6%	0.7%
ctags	5,948	41.6%	5,619	39.3%	6,248	43.7%	5,891	41.2%	5.0%	4.8%
cvs	31,404	46.3%	29,912	44.1%	31,472	46.4%	30,014	44.3%	0.2%	0.3%
diffutils	2,871	22.6%	2,668	21.0%	3,011	23.7%	2,782	21.9%	4.9%	4.3%
ed	4,822	53.3%	4,333	47.9%	4,894	54.1%	4,387	48.5%	1.5%	1.3%
empire	16,104	33.0%	13,762	28.2%	20,398	41.8%	17,178	35.2%	26.7%	24.8%
EPWIC-1	646	11.3%	618	10.8%	669	11.7%	641	11.2%	3.5%	3.7%
espresso	6,708	30.8%	6,512	29.9%	6,926	31.8%	6,730	30.9%	3.2%	3.3%
findutils	3,257	27.5%	2,949	24.9%	3,340	28.2%	3,020	25.5%	2.5%	2.4%
flex2-4-7	2,642	24.8%	2,365	22.2%	2,642	24.8%	2,376	22.3%	0.0%	0.5%
flex2-5-4	3,194	20.9%	2,797	18.3%	3,592	23.5%	3,286	21.5%	12.4%	17.5%
ftpd	5,361	34.9%	4,808	31.3%	5,438	35.4%	4,885	31.8%	1.4%	1.6%
gcc.cpp	2,625	45.8%	2,424	42.3%	2,722	47.5%	2,510	43.8%	3.7%	3.5%
gnubg-0.0	1,558	22.3%	1,761	25.2%	2,013	28.8%	1,915	27.4%	29.1%	8.7%
gnuchess	6,271	43.0%	5,877	40.3%	6,329	43.4%	5,936	40.7%	0.9%	1.0%
gnugo	23,769	34.8%	19,329	28.3%	26,911	39.4%	24,247	35.5%	13.2%	25.4%
go	15,835	61.7%	15,245	59.4%	15,964	62.2%	15,425	60.1%	0.8%	1.2%
ijpeg	5,761	31.0%	5,576	30.0%	5,817	31.3%	5,631	30.3%	1.0%	1.0%
indent-1.10.0	2,117	43.8%	1,938	40.1%	2,243	46.4%	2,045	42.3%	5.9%	5.5%
li	1,872	38.3%	1,823	37.3%	1,901	38.9%	1,843	37.7%	1.6%	1.1%
named	23,075	37.5%	21,352	34.7%	23,813	38.7%	21,906	35.6%	3.2%	2.6%
ntpd	8,278	26.9%	8,278	26.9%	13,725	44.6%	12,555	40.8%	65.8%	51.7%
oracolo2	983	11.8%	1,017	12.2%	1,642	19.7%	1,642	19.7%	66.9%	61.5%
prepro	967	11.6%	1,000	12.0%	1,617	19.4%	1,608	19.3%	67.2%	60.8%
replace	111	21.6%	111	21.7%	111	21.6%	111	21.7%	0.0%	0.0%
sendmail	10,487	33.3%	9,542	30.3%	10,612	33.7%	9,668	30.7%	1.2%	1.3%
space	800	12.9%	818	13.2%	1,314	21.2%	1,321	21.3%	64.3%	61.4%
spice	29,688	21.8%	29,007	21.3%	37,722	27.7%	35,680	26.2%	27.1%	23.0%
termutils	1,232	25.1%	1,134	23.1%	1,242	25.3%	1,144	23.3%	0.8%	0.9%
tile-forth-2.1	1,550	51.9%	1,424	47.7%	1,574	52.7%	1,442	48.3%	1.5%	1.3%
time-1.7	347	8.3%	268	6.4%	360	8.6%	272	6.5%	3.6%	1.6%
userv-0.95.0	1,190	19.4%	1,079	17.6%	1,269	20.7%	1,147	18.7%	6.7%	6.3%
wdiff.0.5	419	10.2%	395	9.6%	419	10.2%	395	9.6%	0.0%	0.0%
which	1,075	29.7%	980	27.1%	1,085	30.0%	991	27.4%	1.0%	1.1%
wpst	1,478	11.0%	1,424	10.6%	1,492	11.1%	1,424	10.6%	0.9%	0.0%
sum	252,398	1207.2%	234,481	1123.8%	279,103	1298.0%	258,498	1202.0%	na	na
average	5,870	28.1%	5,453	26.1%	6,491	30.2%	6,012	28.0%	10.3%	9.1%

Fig. 8. Sizes for backward and forward interprocedural slices for s_1 with structure fields collapsed and expanded.

each program, as well as its average slice size for backward slicing with fields expanded, forward slicing with fields expanded, backward slicing with fields collapsed, and forward slicing with fields collapsed. Each average is given both in terms of lines of code and as a percentage of lines of code from the original program. As seen in the last row of the table, over all 43 programs, the average backward slice increased in size from 28.1% of the program to 30.2% and the average forward slice increased from 26.1% to 28.0%.

The first five columns of Figure 8 repeat data from Figure 3. Columns 6 to 9 show the slice sizes when structure fields are collapsed. Collapsing structure fields have the smallest influence of the three variants on slice size. These values are only a few percentage points above those obtained with expanded structure fields. The only significant increase that accompanies the collapse of structure fields occurs with programs where structures contain multiple pointers and, in

	Model	
Program	s	a
barcode	1,260	1,260
bc	5,497	5,486
copia	252	252
ctags	6,248	6,248
ed	4,894	4,894
EPWIC-1	696	669
flex-2-5-4	3,831	3,592
indent	2,243	2,243
replace	117	111
space	1,314	1,314
average	2,635	2,607
improvement		1.1%

Fig. 9. Pointer analysis model comparison. Figures are average slice size in SLoC.

particular, pointers to functions. For example, the most extreme case of this is the program `prepro`. This effect has also been observed and studied by Mock et al. [2002]. The final two columns report the percent increase in slice size caused by collapsing structure fields.

4.4 Steensgaard versus Andersen

In general, the choice of points-to analysis can affect the results of "downstream" analyses such as slicing. In practice, greater precision in points-to analysis does not tend to yield greater precision in slicing, as this precision is masked by other dependences in the program [Mock et al. 2002]. This section presents results for a subset of the programs studied. It explores the effect of using Steensgaard's algorithm in place of (the more precise) Andersen's algorithm. The imprecision in Steensgaard's algorithm makes it impossible to construct the SDG for the entire collection of subject programs. In other words, the imprecision of Steensgaard pointer analysis leads to a dramatic increase in the number of edges on the SDG, making computation of the SDG for larger programs infeasible.

Figure 9 shows the results of using Steensgaard's and Andersen's algorithms. As can be seen, there is a small overall reduction in the average slice size that can be achieved using Andersen's algorithm, as opposed to than Steensgaard's.

4.5 Effect of Dead Code

The data presented in this subsection measures the effect of restricting Slicer s_1 to ignore vertices representing dead code. A statement (vertex) is considered dead if there is no context-sensitive dependence path from the entry of the main procedure to the statement. This is a safe approximation to the set of statements that can not be executed on any input. As explained in Section 3, such code is identified by slicing forward on the entry vertex of function main.

Figure 10 shows the resulting data. In addition to each program and its size (in SLoC), the figure includes the percentage of live code in each program. This varies significantly from 61% to 99.9%. However, the distribution is highly

program	sloc count	live code sloc	live code percent	slices: all code (repeated) sloc	all code (repeated) percent	live code only sloc	live code only percent	percent reduction
a2ps	40,222	39,497	98.2%	12,348	30.7%	12,339	30.7%	0.07%
acct-6.3	6,764	5,536	81.9%	501	7.4%	410	6.1%	18.12%
barcode	3,975	3,784	95.2%	1,212	30.5%	1,192	30.0%	1.66%
bc	11,173	11,026	98.7%	5,452	48.8%	5,443	48.7%	0.17%
byacc	5,501	5,395	98.1%	996	18.1%	995	18.1%	0.06%
cadp	10,620	6,478	61.0%	828	7.8%	750	7.1%	9.42%
compress	1,431	1,310	91.5%	356	24.9%	351	24.5%	1.52%
copia	1,112	1,111	99.9%	252	22.7%	252	22.7%	0.00%
csurf-pkgs	38,507	34,904	90.6%	6,007	15.6%	5,788	15.0%	3.65%
ctags	14,298	14,084	98.5%	5,948	41.6%	5,930	41.5%	0.30%
cvs	67,828	67,766	99.9%	31,404	46.3%	31,380	46.3%	0.08%
diffutils	12,705	12,187	95.9%	2,871	22.6%	2,864	22.5%	0.25%
ed	9,046	8,871	98.1%	4,822	53.3%	4,814	53.2%	0.16%
empire	48,800	47,313	97.0%	16,104	33.0%	16,097	33.0%	0.04%
EPWIC-1	5,719	4,067	71.1%	646	11.3%	591	10.3%	8.50%
espresso	21,780	21,249	97.6%	6,708	30.8%	6,703	30.8%	0.07%
findutils	11,843	11,097	93.7%	3,257	27.5%	3,233	27.3%	0.72%
flex2-4-7	10,654	10,249	96.2%	2,642	24.8%	2,634	24.7%	0.29%
flex2-5-4	15,283	14,054	92.0%	3,194	20.9%	3,171	20.7%	0.73%
ftpd	15,361	14,984	97.5%	5,361	34.9%	5,335	34.7%	0.49%
gcc.cpp	5,731	5,596	97.6%	2,625	45.8%	2,613	45.6%	0.46%
gnubg-0.0	6,988	6,352	90.9%	1,558	22.3%	1,555	22.3%	0.22%
gnuchess	14,584	13,823	94.8%	6,271	43.0%	6,245	42.8%	0.42%
gnugo	68,301	65,211	95.5%	23,769	34.8%	22,726	33.3%	4.39%
go	25,665	25,159	98.0%	15,835	61.7%	15,810	61.6%	0.16%
ijpeg	18,585	17,949	96.6%	5,761	31.0%	5,758	31.0%	0.05%
indent	4,834	4,558	94.3%	2,117	43.8%	2,085	43.1%	1.54%
li	4,888	4,876	99.8%	1,872	38.3%	1,872	38.3%	0.01%
named	61,533	60,251	97.9%	23,075	37.5%	22,825	37.1%	1.08%
ntpd	30,773	29,485	95.8%	8,278	26.9%	8,258	26.8%	0.24%
oracolo2	8,333	7,851	94.2%	983	11.8%	982	11.8%	0.17%
prepro	8,334	7,830	93.9%	967	11.6%	965	11.6%	0.18%
replace	512	508	99.3%	111	21.6%	111	21.6%	0.00%
sendmail	31,491	31,265	99.3%	10,487	33.3%	10,484	33.3%	0.02%
space	6,200	5,953	96.0%	800	12.9%	798	12.9%	0.18%
spice	136,182	90,645	66.6%	29,688	21.8%	24,197	17.8%	18.49%
termutils	4,908	4,781	97.4%	1,232	25.1%	1,229	25.0%	0.25%
tile-forth-2.1	2,986	2,842	95.2%	1,550	51.9%	1,483	49.7%	4.32%
time-1.7	4,185	3,854	92.1%	347	8.3%	342	8.2%	1.53%
userv-0.95.0	6,132	6,028	98.3%	1,190	19.4%	1,189	19.4%	0.04%
wdiff.0.5	4,112	3,870	94.1%	419	10.2%	418	10.2%	0.41%
which	3,618	3,493	96.5%	1,075	29.7%	1,054	29.1%	1.95%
wpst	13,438	11,596	86.3%	1,478	11.0%	1,450	10.8%	1.92%
sum	824,935	748,736		252,398		244,722		
average	19,185	17,948	93.6%	5,870	28.1%	5,691	27.7%	1.98%

Fig. 10. Live code experiment.

skewed towards the 90's: Two-thirds of the programs are 95% or more live code and nine in ten are 90% or more.

The final five columns of Figure 10 illustrate the effect on slice size. They include the average slice size, repeated for comparison from Figure 3, the average slice size counting only live code, and finally, the change in slice size obtained by considering only live code. For most programs the amount of dead code is minimal; thus, its effect on slice size is minimal: Two-thirds of the programs show less than 1% difference and all but four show less than 5% difference in average slice size. Pearson correlations between the percentage of live code and the percentage of reduction in average slice size show a statistically significant ($p < 0.00001$) correlation with $R = -0.80$ (a rise in live code produces a decrease in slice size reduction). The impact on forward slicing (not shown in Figure 10) is similar, but slightly muted. For example, while `acct` shows an 18% change for backward slicing, it shows only a 13% change for forward slicing.

5. THREATS TO VALIDITY

In any empirical study, it is important to consider *threats to validity*. In the absence of human subjects, only two potential threats need be considered. These are threats to external and internal validity. External validity, sometimes referred to as selection validity, is the degree to which the findings can be generalized. Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variable on the dependent variable.

In the following discussion, a distinction is made between algorithm selection (which may threaten external validity) and implementation detail (which may threaten internal validity). Here different algorithms have different input-output relationships, while the details of implementation affect the performance (execution timing and memory footprint) of an algorithm, but not its semantics.

Three selections were made in the experiment: the programs analyzed, slicing criterion, and SDG construction options. Regarding the first, it is possible that the selected programs are not representative of programs in general and thus results from the experiment do not apply to “typical” programs. Considering 43 subject programs helps to mitigate this concern. As described previously, these programs include diverse programming styles and domains. This diversity makes it more likely that the conclusions made about the techniques generalize. However, it remains possible that programs written in styles not considered herein (e.g., real-time, embedded, and event-driven systems), or nonopen source programs will exhibit significantly different behaviors.

The second form of external validity comes from *slice selection bias:* the degree to which results depend on the particular slices chosen for study. As discussed in the Related Work section, previous empirical work has focused on detailed examination of a limited number of slices. These slices were chosen either by experienced programmers [Bent et al. 2000] or by selecting particular vertex types [Agrawal and Guo 2001; Krinke 2002]. This selection introduces bias to the results: It is not known whether a favorable result was caused by fortunate slice selection. To mitigate this concern, the slice with respect to every executable line of code (every vertex that represents executable code) was taken.[1]

[1]As an interesting aside, slicing on all vertices (not just those representing executable code) does not change the results other than taking proportionally longer to process each program. The most common source of these vertices is the passing of global variables between procedures, as globals are modeled as additional procedure parameters.

Slicing on all executable lines of code allows the experiment to investigate the question of what a typical slice size might be for a slice chosen "at random." It is, however, possible that the results for only those slices of interest to an engineer are statistically different from all slices. It is encouraging to note that related work indicates that this is not the case [Bent et al. 2000].

The final form of external validity deals with the extent to which results can be generalized to alternate "upstream" SDG constructions. It is conceivable that there may be some algorithm alternatives or future improvements that affect SDG construction, resulting in a dramatic effect on slice size. For example, improved points-to analysis precision might reduce slice size. However, existing empirical evidence for slicing "in the small" suggests that this is not the case [Landi and Ryder 1992; Mock et al. 2002; Liang and Harrold 1999a, 1999b].

The second threat to validity is internal: the degree to which conclusions can be drawn about the causal effect of the independent variable on the dependent variable. In this experiment, the only serious threat comes from the potential for faults in the slicers. Other common forms of internal validity, for example, construct validity (the degree to which the variables used in the study accurately measure the concepts they purport to measure), are not an issue, as the variable measured (slice size) can be measured with high precision. Thus, the only serious internal validity concern comes from the assumption that tools correctly implement each slicing algorithm. In practice, the slicing tools might contain errors, or employ imprecise analyses (e.g., imprecise data-flow analysis or points-to analysis). To mitigate this concern, mature slicing tools were used and thoroughly tested. This reduces the impact that implementation faults may have on the conclusions reached regarding dependence analysis.

6. IMPLICATIONS OF THE FINDINGS

Weiser's original motivation for slicing was that it would be a technique to support debugging [1979]. The argument was elegant, simple, and compelling:

> Why should programmers waste time in debugging activity that considers parts of the program that cannot possibly have an effect upon the point at which an error was noticed, when automated tool support could reveal this?

One of the original motivations for the study of dynamic slicing [Korel and Laski 1988] was to improve the applicability of slicing to debugging. Initial anecdotal evidence from small-scale static slicing prototype tools was that static slices were disappointingly large and therefore of less help to debugging than had been hoped.

However, many authors continued to develop new application areas for static slicing, such as reverse engineering [Canfora et al. 1994b; Simpson et al. 1993], program comprehension [De Lucia et al. 1996; Harman et al. 2001], testing [Binkley 1998; Gupta et al. 1992; Harman and Danicic 1995; Hierons et al. 2002, 1999], and software metrics [Bieman and Ott 1994; Lakhotia 1993; Ott and Thuss 1993]. Following Weiser's initial work, many authors also developed techniques for slicing, but not until comparatively recently have sufficiently mature program slicing tools been able to handle large programs written using the entirety (rather than merely a subset) of a popular programming language. The advent of these tools has made large-scale empirical study of slice size possible so that we may address the fundamental question: "How large can we expect a typical static slice to be?"

The answers to this question provided herein have important implications for the many applications for slicing that have been proposed and studied since Weiser's original work [1979]. For example, the fact that slices tend to consist of about one-third of the programs from which they are constructed is an optimistic result for all applications, since size of slice is crucial in all cases.

Bieman and Ott [1994] use static slicing as a basis for the measurement of functional cohesion. The essential idea is that the larger degree of overlap existing among a set of a function's slices, the more cohesive the function must be, since it contains a larger proportion of its "shared" computation. However, the presence of dead code will tend to depress the value of the metric because dead code will not appear in any slice and so will not be in the intersection. The presence of large amounts of dead code would tend to suggest caution in the measurement of cohesion using the approach of Bieman and Ott.

Fortunately, as the results presented in Section 4.5 show, the impact of dead code on slice size is generally low. However, the simple forward slicing technique used for eliminating code which is dead to the slicer might be used as prudent augmentation to the Bieman and Ott [1994] cohesion metric. In other words, in assessing functional cohesion, it would be advisable to remove code not in the forward slice on the entry vertex from all computations.

In work on reverse engineering, the finding that slice sizes tend to be about a third of the program means that slicing can be a useful technique for supporting extraction of program fragments for reengineering. Furthermore, the variation in slice size noted in Section 4 is very important for work on reengineering. It suggests, as might be expected, that amenability to a slice-based approach to reengineering will vary dramatically from one program to another. This suggests that some prior analysis of slice sizes could form the basis of an assessment of suitability for slice-based approaches to reengineering.

In work on software testing, slicing has been proposed as a technique for reducing testing effort. For example, when attempting to generate test data to execute a particular branch or statement (for coverage adequacy purposes [Beizer 1990]), there will be no point in executing statements that cannot affect the node or predicate to be covered. In automated test data generation using heuristic techniques, such as the chaining method [Ferguson and Korel 1996] and evolutionary testing [Wegener et al. 2001], the program under test may need to be executed many tens of thousands of times in order to generate test data to cover some hard-to-reach statement or branch. In such cases it makes sense to first slice the program to make a smaller version (the slice) tailored to the testing objective at hand. The observation that slices are, on average, about one-third of the program from which they are constructed suggests that this is worthwhile.

In other approaches to software test data generation, such as symbolic execution [Clarke 1976], the complexity of the symbolic execution mechanism is a crucial factor in applicability. Here, once again, the ability to (on average) slice away about two-thirds of the program may have significant benefits.

For program comprehension the findings are also important. As the results show, there are many programs that have sets of very small slices, making these programs highly amenable to slice-based approaches to comprehension. Even for programs with larger slices, any reduction in code size is going to be beneficial to comprehension, since it avoids the human wasting time on code not relevant to the comprehension question at hand. The fact that the study found very few extremely large slices indicates that optimism is not misplaced.

Furthermore, the ability to quickly produce slices at the function level of granularity may be beneficial in work on slice-based comprehension. Strong statistical correlations between statement- and function-level slicing indicate that the function-level slice could be used as predictor of statement-level slice size. This would provide a mechanism for quickly identifying parts of the program for which slice-based comprehension may have the biggest impact.

7. RELATED WORK

There has been considerable previous work concerning the tradeoffs between algorithmic precision and slice size [Agrawal and Guo 2001; Atkinson and Griswold 1998; Beszédes and Gyimóthy 2002; Binkley and Harman 2003b; Harrold and Ci 1998; Krinke 2002; Liao et al. 1999; Liang and Harrold 1999a; Mock et al. 2002; Orso et al. 2001b; Reps et al. 1994; Sinha et al. 1999; Lyle et al. 1995; Venkatesh 1995; Zhang et al. 2003]. Although this work touches upon the issue of slice size, much of it has been concerned with the development of slicing algorithms. Consequently, such work uses changes in slice size that arise from differing algorithmic choices to provide empirical evidence to support a particular choice of algorithm, rather than addressing the question of typical slice size per se.

The results of experiments reported in the present article on the impact of the choice of points-to analysis on slice size tend to bear out the findings of previous authors, who find that increasing points-to precision for slices of C programs does not appear to yield significant additional gains, once a certain level of precision is reached. Mock et al. [2002] attribute this to the fact that increasing points-to precision may reduce the number of edges in the SDG, but this does not necessarily increase the average slice size. In other words, while increasing edges creates more transitive dependence paths in the SDG, in order for some node t to be in the slice, all that is required is a *single* transitive dependence path from a node s which is in the slice to node t.

Liang and Harrold [1999b] compare slice size using four points-to analyses, namely, Steensgaard's [1996], Andersen's [1994], Landi and Ryder's (a flow- and context-sensitive analysis) [1992], and their own flow-insensitive, context-sensitive points-to analysis [Liang and Harrold 1999a]. In this experiment, slices are taken of the programs, which range in size from 1,132 to 17,263 LoC. For all but one program, which shows a 14% difference, the difference in slice size is less than 2%. Relevant to this discussion, it was only possible to run

Cluster	Slices in Cluster	Percent of Total Slices
Size <= 1%	155	37%
1% < Size <= 25%	135	32%
25% < Size <= 50%	129	31%
50% < Size	0	0%
Sum	419	100%

Fig. 11. `Unravel` slice size analysis.

Landi and Ryder's points-to analysis on smaller programs, that is, those with fewer than 5,000 LoC.

Mock et al. [2002] show that using precise dynamic pointer information yields only small improvement in slice size. This is because removing a dependence that connects two statements does not change the slice when there exists another (possibly transitive) dependence connecting the two statements.

The rest of this section considers other previous empirical work on program *slices*. In particular, research dealing with empirical studies of program *slicers* is not considered. To begin with, the Unravel project report considered all slices of a single program [Lyle et al. 1995]. Figure 11 presents a summary of the results from this study.

The work of Orso et al. [2001a, 2001b], Sinha et al. [1999], and Liang and Harrold [1999a] also considered slice size (though this was not the primary focus of their work). Like the study reported herein, these authors considered all statement-level slicing criteria, but the studies concerned a smaller set of smaller programs. For programs on the order of thousands of lines of code, Orso et al. [2001a, 2001b] were able to slice on all statement-level criteria. For those on the order of tens of thousands of lines of code, the results reported were based on a single slice. The study reported herein therefore continues the style of these approaches, but is considerably larger in scale, both with regard to the size and number of programs considered. The authors believe that the work reported herein is at least one order of magnitude larger than any previous related study.

Beszédes and Gyimóthy [2002] describe the approximation of static slices using the union of dynamic slices. They consider three programs ranging from 5,592 to 37,539 lines of code and report an average reduction (portion of the code removed by slicing) of 18%. This stands in contrast to the over 70% average removal reported in Section 4. It is not clear what the difference should be attributed to. However, note that the slicing algorithm used by Beszédes and Gyimóthy [2002] is based upon unioning dynamic slices and so it is markedly different from that used herein. A more direct comparison might prove interesting because, in theory, the union of dynamic slices could never exceed the size of a static slice.

Liao et al. describe the SUIF tool [1999], with which slices were used to reduce the number of lines of loop code examined by a programmer. While not the focus of their work, some slice size statistics are given. An average reduction of 78% was reported. The loops studied were small (the largest is 268 lines of code). However, it is interesting to note that a similar reduction was found for the much larger dataset studied herein.

Bent et al. [2000] present 18 slices computed for six different programs ranging in size from 842 to 14,554 lines of code. They report an average slice size (measured in nonblank noncomment lines of code) of 12%, with the largest slice size being 35%. Two of the programs they studied also appear in the present article in Figure 3 (although they may be different versions): For `compress` the three slices included 1.5%, 14%, and 41% of the program (compared to the 25% shown in Figure 3), and for `ed` the three slices included 16%, 4.2%, and 14% of the program (compared to the 54% shown in Figure 3). In contrast to the work reported herein, Bent et al. reported considerably smaller slices. This is encouraging, as the slicing criteria in their study were picked by "experts" to be "representative." However, their sample size is small (18 versus 1,176,158 backward slices).

Two other studies by Atkinson and Griswold also include some slice size data. In the first [2001], six slices (one from each of six programs, measuring 2,692 to 189,043 nonblank lines of code) were found to include 16%, 27%, 6.7%, 31%, 31%, and 30% of the program. The average of these is 24%, which is close to, but below, the average given in Figure 3. Care is required in drawing conclusions from this comparison, as the number of slices actually taken is so small. Also, the slices were constructed without regard for calling context. As the study reported herein indicates, this can lead to a marked decrease in precision. In their second study [Atkinson and Griswold 1998], slice size (not the focus of the work) is given in terms of the number of "three address statements in the slice". As the initial number of three address statements for each program is not given, a percentage of the reduction computation is impossible.

Krinke undertook an empirical evaluation of slicing (and chopping) [2002]. His study concentrates on the effects of k-limiting call strings, where context information is only kept to depth k. His work summarizes data from 39,043 slices of 14 programs (slices were computed from all formal-in vertices in each program's SDG). The average slice size ranged from 17.3% to 51.0% of the program with an overall average of 39.5%.

Agrawal and Guo studied the tradeoff between context sensitivity and computation time for a collection of Java programs, ranging up to about 3,000 LoC [2001]. The focus of their work was on the effect of different techniques for dealing calling context.

They found that ignoring context caused an 86% increase in slice size while speeding-up slicing by a factor of 1.72 [2001]. Their approach to handling context was expensive. The authors noted that "in the worst case, the number of times a vertex is added to the work list is exponential in the number of methods in the program." In contrast, the slicing algorithm studied in Section 4 is linear in size of the SDG and handles calling context with no added space overhead. In fact, the data from Section 4 actually shows an *increase* in slicing time when calling context is ignored (due to the increased number of vertices that must be processed). The discrepancy between the runtimes is most likely due to inefficient context handling in the Agrawal and Guo slicer, which, it turns out, was found to be incorrect by Krinke [2002]. Krinke reports that ignoring context caused a 68% increase in slice size and a 40% increase in slice time. The size increased by more than the 50% increase reported in Figure 4, whereas the

40% time increase is less than the 77% increase from Figure 4. The difference may be accounted for by slicing-criterion selection bias.

All of the aforementioned studies considered slice size for C programs, but more recently, there has been initial work on slicing for Java, where similar tradeoffs between calling context and slice size have been investigated [Hammer and Snelting 2004]. It is evident from the related work that there is a need for standardization in reporting statistics related to slice sizes. This has caused some authors to move away from measurement based on code so as to consider function points [Jones 1991]. For example, the research described earlier uses "nonblank noncomment" lines of code, "nonblank" lines of code, "three address statements," and "LoC as reported by word count" to measure program and slice size. While "LoC as reported by word count" is a rather crude metric, it does at least provide a common baseline to which results can be compared. As the present study has included data for a large number of slices and reports both total LoC and NC-NB LoC, it is hoped that future studies will do the same to help facilitate comparison.

Unlike the work reported here, many large-scale studies avoid slicing on all statement-level slicing criteria (for practical reasons). Agrawal and Guo [2001] comment that "considering all possible slicing criteria from every program in our benchmark set would have resulted in a very large total number of [slices], which would not have been feasible to experiment with." Krinke comments that some slice constructions took over 300MB of memory or failed to finish in eight hours. Using the SDGs that CodeSurfer produces, augmented by collapsing SCCs and topologically sorting the vertices, it is possible to slice on every vertex that represents executable code and to do so for larger programs than were considered by either Agrawal and Guo or Krinke. In part, the ability to slice large programs on every slicing criterion was made possible by a set of optimizations to the SDG analysis reported elsewhere [Binkley and Harman 2003b].

The ability to slice on every possible slicing criterion is important: It avoids the potential selection bias that may be present in the studies of Bent et al. [2000], Agrawal and Guo [2001], and Krinke [2002]. However, it may mean that the slices studied here could not be called "typical" in the sense that they would be representative of the subset selected in some particular application of program slicing. Of course, deciding upon what constitutes a typical slicing criterion (and therefore a typical slice) is an inherently subjective, qualitative, and application-dependent judgment. Therefore, the results reported herein form a general baseline against which to compare future work that considers typical slices for a given domain.

8. SUMMARY

In all applications of program slicing, the issue of slice size is important. The smaller the slice, the better. This article reports results from the largest study of static program slice size conducted to date. The results give a benchmark size of a static slice (either forward or backward) of just under one-third of the program (28.1% for backward and 26.1% for forward slices). While one-third of a million lines is still a substantial quantity of code to analyze, slicing a 100 line procedure to produce a 33 line subprocedure can be highly useful and is, indeed, used in many applications of slicing, including measurement [Bieman and Ott 1994; Longworth et al. 1986; Ott and Thuss 1989], testing [Harman and Danicic 1995], and comprehension [Cimitile et al. 1995a, 1995b].

The article considers the impact of calling context, which is a somewhat unresolved issue, since some previous authors (e.g., Krinke [2002]) suggest that calling context is important, while others (e.g., Mock et al. [2002]) suggest it is not. The results presented here go a long way towards settling the issue, due to the size of the study. The results appear to replicate those presented by Krinke [2002] indicating that ignoring calling context causes an increase in both slice size and computation time.

The article investigated the slices constructed at larger, function-level granularity. At this function level, slices proved to be larger (due to the imprecision of larger granularity), but not impractically so. A function slice is around 50% larger than a corresponding statement slice, but can be computed in negligible time. This linear trend in price of the imprecision paid for the speedup was found to be strongly statistically significant.

Finally, the article considers the impact on slice size of three upstream analysis and manipulation phases: structure field collapse, points-to analysis, and dead code removal. Collapsing structure fields increases average slice size by only a few percentage points. For some programs (those which made use of function pointers in structure fields), the impact of structure field expansion was found to be much greater, replicating the results of Bent et al. [2000]. The choice of points-to analysis and the removal of dead code were found to have little effect on average slice size.

Future investigation in this area could fruitfully address the correlation between call-tree depth and slice size. Continued investigation of the programming styles, techniques, and use of function pointers in programs would also be helpful in characterizing the types of slice arising from them.

ACKNOWLEDGMENTS

The anonymous referees provided many detailed and insightful suggestions which prompted several of the experiments, the results of which are reported here. GrammaTech provided *CodeSurfer*$^{\text{tm}}$ upon which the implementation is based. Alberto Pasquini provided the programs `oracolo2`, `prepro`, and `copia`. Gregg Rothermel provided the program `empire` and Erin Ptah assisted with the presentation.

REFERENCES

AGRAWAL, G., AND GUO, L. 2001. Evaluating explicitly context-sensitive program slicing. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, UT, Jun. 18–19). ACM Press, New York. 6–12.

AGRAWAL, H., DEMILLO, R. A., AND SPAFFORD, E. H. 1993. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper. 23*, 6 (Jun.), 589–616.

ANDERSEN, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU Rep. 94/19, University of Copenhagen, May.

ANDERSEN, P., BINKLEY, D., ROSAY, G., AND TEITELBAUM, T. 2001. Flow insensitive points-to sets. In *Proceedings of the 1st IEEE Workshop on Source Code Analysis and Manipulation* (Nov.). IEEE Computer Society Press, Los Alimitos, CA. 79–89.

ATKINSON, D. C. AND GRISWOLD, W. G. 2001. Implementation techniques for efficient data-flow analysis of large programs. In *IEEE International Conference on Software Maintenance* (Nov.). IEEE Computer Society Press, Los Alamitos, CA. 52–61.

ATKINSON, D. C. AND GRISWOLD, W. G. 1998. Effective whole-program analysis in the presence of pointers. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering* (New York, Nov. 3–5). *Softw. Eng. Not. 23*, 6, 46–55.

BECK, J. AND EICHMANN, D. 1993. Program and interface slicing for reverse engineering. In *IEEE/ACM 15th Conference on Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA. 509–518.

BEIZER, B. 1990. *Software Testing Techniques.* Van Nostrand Reinhold, New York.

BENT, L., ATKINSON, D., AND GRISWOLD, W. 2000. A qualitative study of two whole-program slicers for C. Tech. Rep. CS20000643, University of California at San Diego. A preliminary version appeared at *the 2000 ACM SIGSOFT International Symposium on the Foundations of Software Engineering.*

BESZEDES, A. AND GYIMOTHY, T. 2002. Union slices for the approximation of the precise slice. In *Proceedings of the IEEE International Conference on Software Maintenance* (Oct.). IEEE Computer Society Press, Los Alamitos, CA. 12–20.

BIEMAN, J. M. AND OTT, L. M. 1994. Measuring functional cohesion. *IEEE Trans. Softw. Eng. 20*, 8 (Aug.), 644–657.

BINKLEY, D. W. 1998. The application of program slicing to regression testing. *Inf. Softw. Technol. 40*, 11–12, 583–594.

BINKLEY, D. W. 1997. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng. 23*, 8 (Aug.), 498–516.

BINKLEY, D. W. AND HARMAN, M. 2005. Locating dependence clusters and dependence pollution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance.* IEEE Computer Society Press, Los Alamitos, CA. 177–186.

BINKLEY, D. W. AND GALLAGHER, K. B. 1996. Program slicing. In *Advances in Computing*, vol. 43, M. Zelkowitz, Ed. Academic Press. 1–50.

BINKLEY, D. W. AND HARMAN, M. 2004. A survey of empirical results on program slicing. *Adv. Comput. 62*, 105–178.

BINKLEY, D. W. AND HARMAN, M. 2003a. A large-scale empirical study of forward and backward static slice size and context sensitivity. In Proceedings of the *IEEE International Conference on Software Maintenance* (Sept.). IEEE Computer Society Press, Los Alamitos, CA. 44–53.

BINKLEY, D. W. AND HARMAN, M. 2003b. Results from a large-scale study of performance optimization techniques for source code analyses based on graph reachability algorithms. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation* (Sept.). IEEE Computer Society Press, Los Alamitos, CA. 203–212.

BINKLEY, D. W., HORWITZ, S., AND REPS, T. 1995. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol. 4*, 1, 3–35.

BURKE, M., CARINI, P., CHOI, J.-D., AND HIND, M. 1995. Flow-Insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 892. Springer Verlag.

CALLAHAN, D. 1988. The program summary graph and flow-sensitive interprocedural data flow analysis. *ACM SIGPLAN Not. 23*, 7 (Jul.), 47–56.

CANFORA, G., CIMITILE, A., DE LUCIA, A., AND LUCCA, G. A. D. 1994a. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance* (Sept.). IEEE Computer Society Press, Los Alamitos, CA. 424–433.

CANFORA, G., CIMITILE, A., AND MUNRO, M. 1994b. RE : Reverse engineering and reuse re-engineering. *J. Softw. Maintenance: Res. Pract. 6*, 2, 53–72.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the 20th Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston, SC, Jan. 10–13). ACM Press, New York. 232–245.

Cimitile, A., De Lucia, A., and Munro, M. 1996. A specification driven slicing process for identifying reusable functions. *Softw. Maintenance: Res. Pract. 8*, 145–178.

Cimitile, A., De Lucia, A., and Munro, M. 1995a. Identifying reusable functions using specification driven program slicing: A case study. In *Proceedings of the IEEE International Conference on Software Maintenance.* IEEE Computer Society Press, Los Alamitos, CA. 124–133.

Cimitile, A., De Lucia, A., and Munro, M. 1996. Qualifying reusable functions using symbolic execution. In *Proceedings of the 2nd Working Conference on Reverse Engineering.* IEEE Computer Society Press, Los Alamitos, CA. 178–187.

Clarke, L. A. 1976. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng. 2*, 3 (Sept.), 215–222.

De Lucia, A. 2001. Program slicing: Methods and applications. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation.* IEEE Computer Society Press, Los Alamitos, CA. 142–149.

De Lucia, A., Fasolino, A. R., and Munro, M. 1996. Understanding function behaviours through program slicing. In *Proceedings of the 4th IEEE Workshop on Program Comprehension.* IEEE Computer Society Press, Los Alamitos, CA. 9–18.

Emami, M., Ghiya, R., and Hendren, L. J. 1994. Context-Sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Not. 29*, 6 (Jun.), 242–256.

Ferguson, R. and Korel, B. 1996. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol. 5*, 1 (Jan.), 63–86.

Foster, J. S., Fahndrich, M., and Aiken, A. 2000. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium.* 175–198.

Fahndrich, M., Foster, J. S., Su, Z., and Aiken, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Jun.). ACM Press, New York. 85–96.

Gallagher, K. B. 1992. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance.* IEEE Computer Society Press, Los Alamitos, CA. 236–244.

Gallagher, K. B. and Lyle, J. R. 1991. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng. 17*, 8 (Aug.), 751–761.

Grammatech Inc. 2002. The codesurfer slicing system. http://www.grammatech.com/products/codesurfer/ReleaseNotes.html.

Gupta, R., Harrold, M. J., and Soffa, M. L. 1992. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance.* IEEE Computer Society Press, Los Alamitos, CA. 299–308.

Hammer, C. and Snelting, G. 2004. An improved sliver for Java. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press, New York. 17–22.

Harman, M. and Danicic, S. 1995. Using program slicing to simplify testing. *Softw. Test. Verif. Reliability 5*, 3 (Sept.), 143–162.

Harman, M. and Hierons, R. M. 2001. An overview of program slicing. *Softw. Focus 2*, 3, 85–92.

Harman, M., Hierons, R. M., Danicic, S., Howroyd, J., and Fox, C. 2001. Pre/Post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance* (Nov.). IEEE Computer Society Press, Los Alamitos, CA. 138–147.

Harman, M., Okunlawon, M., Sivagurunathan, B., and Danicic, S. 1997. Slice-Based meaurement of coupling. In *Proceedings of the 19th Workshop on Process Modelling and Empirical Studies of Software Evolution* (Boston, MA, May), R. Harrison, Ed.

Harrold, M. J. and Ci, N. 1998. Reuse-Driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering* (Apr.). IEEE Computer Society Press, Los Alamitos, CA. 74–83.

Heintze, N. and Tardieu, O. 2001. Ultra-Fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, Jun. 20–22), C. Norris and J. J. B. Fenwick, Eds. *ACM SIGPLAN Not. 36*, 5, 254–263.

HIERONS, R. M., HARMAN, M., AND DANICIC, S. 1999. Using program slicing to assist in the detection of equivalent mutants. *Softw. Test. Verif. Reliability 9*, 4, 233–262.

HIERONS, R. M., HARMAN, M., FOX, C., OUARBYA, L., AND DAOUDI, M. 2002. Conditioned slicing supports partition testing. *Softw. Test. Verif. Reliability 12*, (Mar.), 23–28.

HORWITZ, S., PRINS, J., AND REPS, T. 1989. Integrating non-interfering versions of programs. *ACM Trans. Program. Lang. Syst. 11*, 3 (Jul.), 345–387.

HORWITZ, S., REPS, T., AND BINKLEY, D. W. 1988. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, GA, Jun.). *ACM SIGPLAN Not. 23*, 7, 35–46.

JAYARAMAN, G., RANGANATH, V. P., AND HATCLIFF, J. 2005. Kaveri: Delivering the Indus Java program to Eclipse. In *Fundamental Approaches to Software Engineering.* Lecture Notes in Computer Science, vol. 3442. Springer Verlag. 269–273.

JONES, C. 1991. *Applied Software Measurement: Assuring Productivity and Quality.* McGraw-Hill, New York.

KAMKAR, M. 1993. Interprocedural dynamic slicing with applications to debugging and testing. Ph.D. thesis, Department of Computer Science and Information Science, Linkoping University, Sweden. Available as Linkoping Studies in Science and Technology, Dissertations, no. 297.

KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Inf. Proc. Lett. 29*, 3 (Oct.), 155–163.

KRINKE, J. 2002. Evaluating context-sensitive slicing and chopping. In *Proceedings of the IEEE International Conference on Software Maintenance.* IEEE Computer Society Press, Los Alamitos, CA. 22–31.

LAKHOTIA, A. 1993. Rule-Based approach to computing module cohesion. In *Proceedings of the 15th Conference on Software Engineering.* 34–44.

LANDI, W. AND RYDER, B. G. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Not. 27*, 7.

LIANG, D. AND HARROLD, M. J. 1999a. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference*, held jointly with *the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, O. Nierstrasz and M. Lemoine, Eds. Lecture Notes in Computer Science, vol. 1687. Springer Verlag. 199–215.

LIANG, D. AND HARROLD, M. J. 1999b. Reuse-Driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings of the IEEE International Conference on Software Maintenance.* IEEE Computer Society Press, Los Alamitos, CA. 410–430.

LIAO, S.-W., DIWAN, A., BOSCH, JR., R. P., OHULOUM, A., AND LAM, M. S. 1999. SUIF explorer: An interactive and interprocedural parallelizer. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, A. A. Chien and M. Snir, Eds. *ACM SIGPLAN Not. 34*, 8, 37–48.

LONGWORTH, H. D., OTT, L. M., AND SMITH, M. R. 1986. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of the Computer Software and Applications Conference*. 383–389.

LYLE, J. R., WALLACE, D. R., GRAHAM, J. R., GALLAGHER, K. B., POOLE, J. P., AND BINKLEY, D. W. 1995. Unravel: A CASE tool to assist evaluation of high integrity software, volume 1: Requirements and design. Tech. Rep. NISTIR 5691, US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899.

LYLE, J. R. AND WEISER, M., 1987. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference on Computers and Applications.* IEEE Computer Society Press, Los Alamitos, CA. 877–882.

MEYERS, T. AND BINKLEY, D. W. 2004. Slice-Based cohesion metrics and software intervention. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering.* IEEE Computer Society Press, Los Alamitos, CA. 256–266.

MOCK, M., ATKINSON, D. C., CHAMBERS, C., AND EGGERS, S. J. 2002. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, W. G. Griswold, Ed. ACM Press, New York. 71–80.

ORSO, A. SINHA, S., AND HARROLD, M. J. 2001a. Effects of pointers on data dependences. In *Proceedings of the 9th IEEE International Workshop on Program Comprehension* (May). IEEE Computer Society Press, Los Alamitos, CA. 39–49.

ORSO, A. SINHA, S., AND HARROLD, M. J. 2001b. Incremental slicing based on data-dependences types. In *Proceedings of the IEEE International Conference on Software Maintenance* (Nov.). IEEE Computer Society Press, Los Alamitos, CA. 158–167.

OTT, L. M. AND THUSS, J. J. 1993. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*. IEEE Computer Society Press, Los Alamitos, CA. 71–81.

OTT, L. M. AND THUSS, J. J. 1989. The relationship between slices and module cohesion. In *Proceedings of the 11th ACM Conference on Software Engineering* (May). 198–204.

REPS, T. 1998. Program analysis via graph reachability. *Inf. Softw. Technol. 40*, 11-12, 701–726.

REPS., T., HORWITZ, S., SAGIV, M., AND ROSAY, G. 1994. Speeding up slicing. In *the ACM Conference on Foundations of Software Engineering* (Dec.). *ACM SIGSOFT Softw. Eng. Not. 19*, 5 (Dec.), 11–20.

RUF, E. 1995. Context-Insensitive alias analysis reconsidered. *ACM SIGPLAN Not. 30*, 6 (Jun.), 13–22.

SHAPIRO, M. AND HORWITZ, S. 1997. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, Jan. 15–17). ACM Press, New York. 1–14.

SIMPSON, D., VALENTINE, S. H., MITCHELL, R., LIU, L., AND ELLIS, R. 1993. Recoup–Maintaing Fortran. *ACM Fortran Forum 12*, 3 (Sept.), 26–32.

SINHA, S., HARROLD, M. J., AND ROTHERMEL, G. 1999. System-Dependence-Graph-Based slicing of programs with arbitrary interprocedural control-flow. In *Proceedings of the 21st International Conference on Software Engineering* (May). ACM Press, New York. 432–441.

STEENSGAARD, B. 1996. Points-To analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, Jan.). ACM Press, New York. 32–41.

TIP, F. 1995. A survey of program slicing techniques. *J. Program. Lang. 3*, 3 (Sept.), 121–189.

VENKATESH, G. A. 1995. Experimental results from dynamic slicing of C programs. *ACM Trans. Program. Lang. Syst. 17*, 2 (Mar.), 197–216.

WEGENER, J., BARESEL, A., AND STHAMER, H. 2001. Evolutionary test envirmonment for automatic structural testing. *Inf. Softw. Technol. 43*, 14, 841–854.

WEISER, M. 1984. Program slicing. *IEEE Trans. Sofw. Eng. 10*, 4, 352–357.

WEISER, M. 1979. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, University of Michigan, Ann Arbor, MI.

WEISER, M. AND LYLE, J. R. 1985. Experiments on slicing-based debugging aids. In *Empirical Studies of Programmers*, Soloway and Iyengar, Eds. Molex. 187–197.

WEYUKER, E. J. 1977. Program schemas with semantic restrictions. Ph.D. thesis, Rutgers University, New Brunswick, NJ, June.

WHEELER, D. A. 2005. *SLOC Count User's Guide.* http://www.dwheeler.com/sloccount/sloccount.html.

WILSON, R. P. AND LAM, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. *ACM SIGPLAN Not. 30*, 6 (Jun.), 1–12.

YONG, S. H., HORWITZ, S., AND REPS, T. 1999. Pointer analysis for programs with structures and casting. *ACM SIGPLAN Not. 34*, 5 (May), 91–103.

ZHANG, S., RYDER, B. G., AND LANDI, W. 1996. Program decomposition for pointer aliasing: A step toward practical analyses. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (New York, Oct. 16–18). *ACM Softw. Eng. Not. 21*, 6, 81–92.

ZHANG, X., GUPTA, R., AND ZHANG, Y. 2003. Precise dynamic slicing algorithms. In *Proceedings of the 25th IEEE International Conference on Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA. 319–329.

Received January 2005; revised April 2006; accepted June 2006