# A Large-Scale Empirical Study of Forward and Backward Static Slice Size and Context Sensitivity

David Binkley
Loyola College in Maryland
Baltimore, Maryland
21210-2699, USA
binkley@cs.loyola.edu

Mark Harman
Brunel University
Uxbridge, Middlesex
UB8 3PH, UK
Mark.Harman@brunel.ac.uk

## Abstract

*A large-scale study of 43 C programs totaling just over 1 million lines of code is presented. The study includes the forward and backward static slice on every executable statement. In total 2,353,598 slices were constructed, with an average slice size being just under 30% of the original program. The results also show that ignoring calling-context led to a 50% increase in average slice size and, in contrast to previous results, a 66-77% increase in computation time (due to the increased size). Though not the principal focus of the study, the results also show an average pace for the slicing engine, on a standard PC, of 3 million lines of code per second, thereby providing additional evidence for static slicing's practicability.*

Keywords: Program Slicing, Semantics-Based Tools, Empirical Study

## 1. Introduction

Software development and maintenance are hard tasks. The objective of this research is to improve programmer productivity by encouraging the use of semantics-based software analysis tools. Program slicing's ability to extract semantically meaningful portions of a program has led to its use in many software maintenance tools [8, 9, 10, 18, 25, 37, 38]. A better understanding of program slicing's capabilities will improve present and future semantics-based tools. An important step towards this goal is an in-depth empirical study of slice sizes for a wide range of programs.

Another goal of this work is to assist in the technology transfer of semantics-based tools that incorporate program slicing, from academia to industry. Technology transfer is known to be a difficult process [41, 42]. A study by Redwine and Riddle showed that, in general, software technology takes 15 to 20 years to move from research to industry [31]. An excellent example of this is object-oriented programming, which was introduced in the 1960's, but did not mature and start its commercial success until the mid-1980's.

Introduced in the mid-1980's, program slicing is just entering its mature phase. To date, it has had several commercial successes and is gaining commercial acceptance. Unfortunately, for proprietary reasons, there are no published accounts of most of these projects. They include the reduction of testing costs for a VHDL hardware project, which reduced testing time from weeks to days and work on slice-supported data warehousing. Finally, a Cobol slicer [28] was used by Andersen Consulting as a backup for their reverse engineering consultancy service, and IBM's VisualAge for Cobol compiler includes a form of slicing, which led to studies of how to handle overlapping `PERFORM` statements in Cobol [17].

In addition to these commercial applications to specific problems, three general purpose program slicing tools exist. First, Blair and Associates, Inc. [12] commercialized the NIST slicer *Unravel* [26]. Second, the sprite tool computes program slices[27]. Third, the deep structure analysis tool *CodeSurfer* [19], marketed by Grammatech, computes program slices along with several other related operations. Codesurfer was used to generate the data discussed in Section 3.

Several recent studies have considered the effectiveness of program slicing, both in general and when applied to specific tasks [1, 7, 21, 24, 26, 32]. Interestingly, results of some of these experiments were not replicated. Potential explanations for these discrepancies are given in Section 4.

---

This research is supported by National Science Foundation grant CCR-0305330 and Engineering and Physical Sciences Research Council grants GR/R98938 and GR/M58719. 

This study reports results from experiments to determine the value of slicing using over 1,000,000 lines of C code from 43 programs ranging in size from 600 to over 150,000 lines of code. The primary question to be addressed deals with the size of program slices. In addition, the time taken to compute slices is presented. The following four sections present necessary background material, the data and its analysis, related work, and finally a summary.

## 2. Background

This section presents a brief summary of program slicing, the programs studied, the environment in which the data was collected, and how slicing times are reported.

### 2.1. Program Slicing

The program slices studied in Section 3 are computed using graph reachability over the System Dependence Graph (SDG) [20]. CodeSurfer [19] was used to build the SDGs. Its 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 the SCC; thus, there is a great potential for saving effort by avoiding redundant work. The second preprocessing step re-orders 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 [11].

After preprocessing, two kinds of slices are computed: backward interprocedural slicing 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 $s$.

### 2.2. Subjects

The study considers just over 1 million lines of C code from 43 subject programs that range in size from 600 LOC to 149 KLOC. Figure 1 shows the 43 subject programs along with some statistics related to the programs and their SDGs. These statistics include the size of each program (in LOC as reported by word count) and the size of the resulting SDG (as both vertex and edge counts). The figure also shows the number of backward slices taken (the same number of forward slices was taken). The final two columns contain statistics related to the time taken to compute the slices. These include the total time taken to compute all backward interprocedural slices and the pace of the slicing computation, which as computed by dividing the number of vertices processed per second by the the average 14.1 vertices per line of code.

The subject programs cover a wide range of programming styles. For example, the program `prepro` is Fortran-esque 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 (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. However, external validity would be improved by considering additional programs and especially program written in styles not considered herein. For example, real-time systems, embedded systems, and event-driven systems.

Note that in Figure 1, the number of lines of code and the number of slices do not match for two reasons. First, lines of code were counted using the Unix word count utility which is a rather crude method. Word count was used because it provides a consistent measure of program size. Second, 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`.

The study was constructed carefully to avoid confounding factors. For example, slice size could have been reported as a percentage of the vertices in the SDG, (effectively the same vertex set as in a CFG). The problem with doing so is that it makes the results hard to compare with other studies that use different graphs. For example, expanding 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 "percentage of vertices in a graph" a moving and poorly defined target even given the same input program. In contrast, while crude, word count always returns the same count regardless of graph constructed. For version 1.7 of CodeSurfer, the reduction as a percentage of SDG vertices is 5.6% less for expanded structure fields and 4.7% less for collapsed structure fields than the reduction reported in Section 3, which is based on line counts. As CodeSurfer slices include non-executable lines, such as declarations, the difference is accounted for primarily by blank and comment lines.

| Subject Programs | | | | | | |
|---|---|---|---|---|---|---|
| program | Size (LOC) | vertices | edges | slices taken | time (sec) | slicing pace (KLOC/sec) |
| replace | 563 | 1,097 | 3,500 | 867 | 0.02 | 9,417 |
| time-1.7 | 6,033 | 4,583 | 14,092 | 1,049 | 0.03 | 14,922 |
| which | 4,880 | 4,666 | 13,327 | 1,156 | 0.06 | 16,337 |
| compress | 1,234 | 5,211 | 14,749 | 1,085 | 0.06 | 2,805 |
| wdiff.0.5 | 5,958 | 7,213 | 21,793 | 2,421 | 0.12 | 12,916 |
| termutils | 6,697 | 8,869 | 27,442 | 3,096 | 0.40 | 12,424 |
| barcode | 5,562 | 10,916 | 35,796 | 3,824 | 1.47 | 5,315 |
| bc | 14,609 | 14,903 | 49,523 | 5,105 | 7.29 | 6,169 |
| acct-6.3 | 9,536 | 18,531 | 80,584 | 7,222 | 0.97 | 6,783 |
| gcc.cpp | 4,079 | 19,095 | 89,513 | 7,354 | 12.47 | 1,272 |
| space | 9,126 | 20,018 | 64,537 | 10,311 | 13.63 | 2,240 |
| prepro | 14,328 | 20,578 | 65,353 | 10,779 | 13.50 | 3,471 |
| oracolo2 | 14,326 | 20,654 | 65,734 | 10,846 | 14.38 | 3,329 |
| EPWIC-1 | 8,631 | 22,217 | 79,026 | 12,447 | 2.45 | 5,267 |
| gnubg-0.0 | 7,229 | 25,138 | 80,922 | 9,447 | 20.88 | 1,239 |
| indent-1.10.0 | 6,100 | 25,737 | 582,728 | 6,602 | 14.64 | 1,511 |
| findutils | 16,891 | 30,322 | 170,697 | 14,320 | 25.21 | 3,215 |
| byacc | 6,337 | 34,019 | 114,377 | 10,150 | 16.75 | 1,098 |
| diffutils | 18,374 | 36,079 | 195,348 | 16,622 | 28.63 | 2,802 |
| flex2-4-7 | 15,143 | 39,037 | 186,523 | 11,104 | 30.69 | 1,756 |
| cadp | 11,068 | 42,212 | 122,478 | 15,672 | 13.42 | 1,451 |
| copia | 1,170 | 43,513 | 126,943 | 4,680 | 22.08 | 74 |
| ed | 12,493 | 44,387 | 222,858 | 16,368 | 85.59 | 1,662 |
| gnuchess | 16,659 | 44,957 | 163,465 | 15,069 | 73.50 | 1,701 |
| wpst | 17,321 | 54,658 | 209,465 | 20,667 | 50.52 | 1,316 |
| flex2-5-4 | 20,252 | 55,019 | 386,413 | 14,114 | 46.76 | 1,108 |
| ftpd | 15,914 | 56,981 | 366,095 | 24,820 | 157.22 | 1,225 |
| tile-forth-2.1 | 3,717 | 59,247 | 215,288 | 11,940 | 142.44 | 157 |
| ijpeg | 24,822 | 62,698 | 265,650 | 23,954 | 193.84 | 1,390 |
| userv-0.95.0 | 7,150 | 70,796 | 272,726 | 12,511 | 141.41 | 270 |
| espresso | 22,050 | 93,326 | 467,165 | 29,044 | 351.29 | 793 |
| go | 28,547 | 111,246 | 416,404 | 35,594 | 785.72 | 1,071 |
| ctags | 16,946 | 120,014 | 422,034 | 20,313 | 360.34 | 581 |
| ntpd | 45,647 | 204,420 | 804,652 | 39,573 | 1,452.69 | 600 |
| gnugo | 15,217 | 278,766 | 1,044,525 | 58,373 | 3,145.85 | 147 |
| csurf-pkgs | 36,593 | 295,220 | 973,703 | 42,777 | 1,363.73 | 261 |
| a2ps | 53,900 | 417,513 | 6,303,686 | 58,009 | 3,835.59 | 463 |
| empire | 53,895 | 496,033 | 2,022,295 | 105,918 | 7,664.52 | 505 |
| li | 6,916 | 889,260 | 3,762,858 | 13,521 | 5,181.00 | 14 |
| sendmail | 75,156 | 1,111,054 | 37,774,280 | 46,889 | 53,917.14 | 48 |
| named | 103,670 | 1,326,070 | 21,824,057 | 104,567 | 100,196.17 | 81 |
| spice | 149,050 | 1,774,846 | 31,308,396 | 213,625 | 425,938.17 | 36 |
| cvs | 93,309 | 6,219,909 | 28,981,757 | 102,353 | 280,298.17 | 27 |
| total | 1,007,098 | 14,241,028 | 140,412,757 | 1,176,158 | 885,620.81 | 129,267 |
| average | 23,421 | 331,187 | 3,265,413 | 27,353 | 20,595.83 | 3,006 |

**Figure 1.** The subject programs studied (sorted by number of vertices).

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 are relevant to a maintenance engineer.

Another potential confounding factor is "slice selection bias." 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." Previous empirical work has focused on a detailed examination of selected slices. These slices were chosen by experienced programmers as representative [6], or for selected types of graph nodes [1, 21]. This selection introduces a selection bias into the results. One cannot know if a favorable result was caused by fortunate selection. The approach adopted herein is to choose all possible criteria, which avoids bias accruing from experimenter choice. It also makes the results easier to compare with related techniques when no selection is performed. This issue is explored further in the related work section.

### 2.3. Data Collection Environment

The data was collected on a dual processor linux box running Red Hat Linux 7.1 and kernel version 2.4.2-2. Each processor is a 1000MHz Pentium III and has a 256Kb cache. The processors share 4Gb of main memory. Memory contention between the two processors on the shared memory bus slow each processor down by about 15.5% relative a single processor in an identical environment; thus, the effective single CPU speed was be 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.

Data was collected by slicing on every vertex in the program that represents source code. The rationale for this was to tie the slices and thus the results on slice size to the source code rather than the underlying SDGs, which include vertices not directly tied to a particular source statement. Slicing on these vertices 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 from procedure to procedure. A global is modeled as an additional parameter to each procedure that may transitively modify it. Finally, slices were not taken with respect to vertices that represent code from the standard C library `libc` nor were statements from `libc` counted in the reported slice sizes.

### 2.4. Times Reported

The times reported in this paper 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 for all statement-level slicing criteria. This is a considerable number of slices and much work went into optimizing the slicing algorithm to speed up the computation of many slices from a single program. The approach to speeding up the computation of multiple slices is interesting in its own right, but is beyond the scope of the present paper.

## 3. The Data

This section presents and analyzes the slice size data collected from the 43 subject programs. The primary discussion centers around the slice size data presented in Figure 2. A related experiment, which concerns the effect of calling context on slice size, is presented at the end of the section.

Codesurfer, the tool used to build the SDGs, has several options that affect its operation. Two of these have potential impact on slice size. The first is the choice of pointer analysis algorithm, which, in general, can be flow-sensitive or a flow-insensitive and context-sensitive or context-insensitive [34, 3]. Neither context-sensitive analyses [14, 39, 33] nor flow-sensitive analyses [22, 15, 16] scale to the size of the programs considered herein. The study could have considered subsets of smaller programs for certain techniques but this would have changed the emphasis to a study on *slicing techniques* rather than *slices*, and would have also violated the goal of considering a large code base. Thus, context- and flow-insensitive pointer analyses was considered [43, 13, 36, 2].

Even within context- and flow-insensitive pointer analyses, there is a spectrum of precisions [34]. At one end of the spectrum, Steensgaard's algorithm [36], for example, merges all the nodes pointed to by a given pointer. In contrast, Andersen's algorithm leaves these nodes expanded.

The loss of precision present in Steensgaard's algorithm causes a corresponding size increase in the slices computed by CodeSurfer. For example, with the program `ijpeg`, the use of Steensgaard's pointer analysis led to an average slice size 2.5 times that computed using Anderson's algorithm. Furthermore, the summary edge computation [32] becomes prohibitively expensive during the SDG build when imprecise points-to information is used. Thus, Andersen's algorithm [2] is used in the study.

The second option that affects slice size is the expansion of structure fields [40]. This is best understood by way of example. Consider the following code:

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

Assume `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 treaded as a single variable. Collapsing has the effect that `write_fn` and `read_fn` are treated as a single field and the call `(*file1.readfn)(···)` appears to also call `write_ext2` and `write_nfs` in addition to `read_ext2` and `read_nfs`. For programs with structures that contain pointers and 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 structure fields expanded. In the sequel, results are for the collapsed-field SDGs unless otherwise noted.

For example, the data in Figure 1 is for the collapsed field SDGs. Here the sum of the number of vertices and edges in all graphs are 14 and 140 million, respectively. The corresponding sums when structure fields are expanded are 16 and 68 million. The effect that the mild increase in vertices but the dramatic reduction in edges has on compute time is summarized in the following table.

| | Total Analysis Time | | |
|---|---|---|---|
| | collapsed (sec) | expanded (sec) | percent decrease |
| 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% |

Figure 2 shows data for both collapsed and expanded structure fields. For each program the figure presents, the size of the program and the average slice size for backward slicing with fields collapsed, forward slicing with fields collapsed, backward slicing with fields expanded, and forward slicing with fields expanded. Each average is given in terms of lines of code and as a percentage of lines of code from the original program. Averages were computed from all slices taken with respect to vertices that represent executable code (the number of slices is given in Figure 1).

As seen in the last row of the table, the average slice over all 43 programs, contained between 26.3% and 30.2% of the program source. The range of the average slice size is striking. For example, backward slicing low is 7.5% (for program `acct-6.3`) and its high is 62% (for program `go`).

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 the average size for the entire collection. For example, `ijpeg` contains about 25 KLOC and has an average slice size of 31%, or 7,695 LOC. Clearly, the complexity of testing, debugging, or maintaining 7,695 LOC is lower than that of testing, debugging, or maintaining 25 KLOC.

Preliminary analysis does not indicate any trend relating program size and slice size. Program size is only one attribute of a program that potentially affects slice size, programming style, use of features such as globals, (function) pointers, etc., also affect slice size. A more detailed study of these effects is planned as future work.

Finally, it is interesting to note that the programs `wpst` and `csurf-pkgs`, which are part of CodeSurfer itself (and thus written by "slicing aware" programmers) have comparatively small slices (although they were not written with slice-ability in mind). Program `wpst`, which performs the Andersen pointer analysis, has an average slice size of about 11%, while `csurf-pkgs`, which is an ADT repository for CodeSurfer, has an average slice size of about 15%. Whether these averages, which are half the overall average, are influenced by programmer familiarity with program slicing is difficult to evaluate without further experimentation on a larger data set.

The remainder of this section considers a related experiment first conducted by Agrawal and Guo [1], who report that ignoring calling-context caused an increase in the memory needed to compute a slice, but decreased slicing time. Other authors have since repeated this experiment. A comparison appears in the related work section. Ignoring calling context treats the calls to a given function as indistinguishable, so that a call from one call-site is treated as a potential call from all other call-cites on the same function. Clearly, though safe, this has the potential to dramatically increase slice size.

The standard slicing algorithm was modified to ignore calling context to allow the effect of calling context upon slice size to be investigated. The overall results are quite different from those reported by Agrawal and Guo. First, using the SDG, ignoring calling context takes no more memory than handling calling context. In fact, it led to a minor reduction. Furthermore, the change had the opposite effect on slicing time. In essence, ignoring calling context saves the time required to setup Pass 2 of the two pass algorithm. However, it produces larger slices (the slices can be no smaller, and thus there is never a reduction in the number of vertices encountered). Increasing size brings an increase in work and thus it takes proportionately more time to discover the vertices of the slice.

Figure 3 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

| Program | Program Size (LOC) | Structure Fields Collapsed: Average Backward Slice Size (LOC) | percent | Average Forward Slice Size (LOC) | percent | Structure Fields Expanded: Average Backward Slice Size (LOC) | percent | Average Forward Slice Size (LOC) | percent |
|---|---|---|---|---|---|---|---|---|---|
| replace | 563 | 122 | 21.6% | 122 | 21.7% | 122 | 21.6% | 122 | 21.7% |
| copia | 1,170 | 266 | 22.7% | 192 | 16.4% | 266 | 22.7% | 192 | 16.4% |
| compress | 1,234 | 307 | 24.9% | 279 | 22.6% | 307 | 24.9% | 279 | 22.6% |
| tile-forth-2 | 3,717 | 1,959 | 52.7% | 1,795 | 48.3% | 1,929 | 51.9% | 1,773 | 47.7% |
| gcc.cpp | 4,079 | 1,938 | 47.5% | 1,787 | 43.8% | 1,868 | 45.8% | 1,713 | 42.0% |
| which | 4,880 | 1,464 | 30.0% | 1,337 | 27.4% | 1,449 | 29.7% | 1,322 | 27.1% |
| barcode | 5,562 | 1,763 | 31.7% | 1,652 | 29.7% | 1,696 | 30.5% | 1,596 | 28.7% |
| wdiff.0.5 | 5,958 | 608 | 10.2% | 572 | 9.6% | 608 | 10.2% | 572 | 9.6% |
| time-1.7 | 6,033 | 519 | 8.6% | 392 | 6.5% | 501 | 8.3% | 386 | 6.4% |
| indent-1.10. | 6,100 | 2,830 | 46.4% | 2,580 | 42.3% | 2,672 | 43.8% | 2,446 | 40.1% |
| byacc | 6,337 | 1,147 | 18.1% | 1,039 | 16.4% | 1,147 | 18.1% | 1,039 | 16.4% |
| termutils | 6,697 | 1,694 | 25.3% | 1,560 | 23.3% | 1,681 | 25.1% | 1,547 | 23.1% |
| li | 6,916 | 2,690 | 38.9% | 2,607 | 37.7% | 2,649 | 38.3% | 2,580 | 37.3% |
| userv-0.95.0 | 7,150 | 1,480 | 20.7% | 1,337 | 18.7% | 1,387 | 19.4% | 1,258 | 17.6% |
| gnubg-0.0 | 7,229 | 2,082 | 28.8% | 1,981 | 27.4% | 1,612 | 22.3% | 1,482 | 20.5% |
| epwic-1 | 8,631 | 1,010 | 11.7% | 967 | 11.2% | 975 | 11.3% | 932 | 10.8% |
| space | 9,126 | 1,935 | 21.2% | 1,944 | 21.3% | 1,177 | 12.9% | 1,205 | 13.2% |
| acct-6.3 | 9,536 | 715 | 7.5% | 687 | 7.2% | 706 | 7.4% | 687 | 7.2% |
| cadp | 11,068 | 919 | 8.3% | 797 | 7.2% | 863 | 7.8% | 742 | 6.7% |
| ed | 12,493 | 6,759 | 54.1% | 6,059 | 48.5% | 6,659 | 53.3% | 5,984 | 47.9% |
| oracolo2 | 14,326 | 2,822 | 19.7% | 2,822 | 19.7% | 1,690 | 11.8% | 1,748 | 12.2% |
| prepro | 14,328 | 2,780 | 19.4% | 2,765 | 19.3% | 1,662 | 11.6% | 1,719 | 12.0% |
| bc | 14,609 | 7,173 | 49.1% | 6,793 | 46.5% | 7,129 | 48.8% | 6,764 | 46.3% |
| flex2-4-7 | 15,143 | 3,755 | 24.8% | 3,377 | 22.3% | 3,755 | 24.8% | 3,362 | 22.2% |
| gnugo | 15,217 | 5,995 | 39.4% | 5,402 | 35.5% | 5,296 | 34.8% | 4,306 | 28.3% |
| ftpd | 15,914 | 5,634 | 35.4% | 5,061 | 31.8% | 5,554 | 34.9% | 4,981 | 31.3% |
| gnuchess | 16,659 | 7,230 | 43.4% | 6,780 | 40.7% | 7,163 | 43.0% | 6,714 | 40.3% |
| findutils | 16,891 | 4,763 | 28.2% | 4,307 | 25.5% | 4,645 | 27.5% | 4,206 | 24.9% |
| ctags | 16,946 | 7,405 | 43.7% | 6,982 | 41.2% | 7,050 | 41.6% | 6,660 | 39.3% |
| wpst | 17,321 | 1,923 | 11.1% | 1,836 | 10.6% | 1,905 | 11.0% | 1,836 | 10.6% |
| diffutils | 18,374 | 4,355 | 23.7% | 4,024 | 21.9% | 4,153 | 22.6% | 3,859 | 21.0% |
| flex2-5-4 | 20,252 | 4,759 | 23.5% | 4,354 | 21.5% | 4,233 | 20.9% | 3,706 | 18.3% |
| espresso | 22,050 | 7,012 | 31.8% | 6,813 | 30.9% | 6,791 | 30.8% | 6,593 | 29.9% |
| ijpeg | 24,822 | 7,769 | 31.3% | 7,521 | 30.3% | 7,695 | 31.0% | 7,447 | 30.0% |
| go | 28,547 | 17,756 | 62.2% | 17,157 | 60.1% | 17,613 | 61.7% | 16,957 | 59.4% |
| csurf-pkgs | 36,593 | 5,745 | 15.7% | 5,489 | 15.0% | 5,709 | 15.6% | 5,452 | 14.9% |
| ntpd | 45,647 | 11,777 | 25.8% | 10,316 | 22.6% | 12,279 | 26.9% | 10,773 | 23.6% |
| empire | 53,895 | 22,528 | 41.8% | 18,971 | 35.2% | 17,785 | 33.0% | 15,198 | 28.2% |
| a2ps | 53,900 | 17,140 | 31.8% | 16,008 | 29.7% | 16,547 | 30.7% | 15,577 | 28.9% |
| sendmail | 75,156 | 25,328 | 33.7% | 23,073 | 30.7% | 25,027 | 33.3% | 22,772 | 30.3% |
| cvs | 93,309 | 43,295 | 46.4% | 40,869 | 43.8% | 43,202 | 46.3% | 41,149 | 44.1% |
| named | 103,670 | 40,120 | 38.7% | 36,907 | 35.6% | 38,876 | 37.5% | 35,973 | 34.7% |
| spice | 149,050 | 41,287 | 27.7% | 39,051 | 26.2% | 32,493 | 21.8% | 31,748 | 21.3% |
| sum | 1,007,098 | 339,140 | | 314,673 | | 308,527 | | 295,574 | |
| min | | | 7.3% | | 6.5% | | 7.4% | | 6.4% |
| max | | | 62.2% | | 60.1% | | 61.7.4% | | 59.4% |
| average | 23,421 | 7,877 | 30.2% | 7,318 | 28.0% | 7,715 | 28.1% | 6,874 | 26.3% |

**Figure 2.** Slices sizes for backward and forward interprocedural slices using both collapsed and expanded structure fields.

figure) to 330% for `acct`. 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!

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 examples have negative 'increases' in Figure 3). Program `li` shows the most dramatic change: a 29% *reduction* in computation time. This is caused, in the worst case, when Pass 2 must revisit every vertex; thus, effectively doubling the work done. A cursory examination of the code shows that `li` contains a large number of small functions. This would increase the time spent on the interface between Pass 1 and Pass 2 of the slicing algorithm because this interface relates to procedure boundaries. The present slicer uses a simple linked list for this interface, which, accounts for most of the computation time interfacing Pass 1 and Pass2. Profiling shows that for `li` this cost is about 10% of total run time. The rest of the time difference appears to be caused by the increased overhead from considering edge types during the graph walk. Further work is needed to verify or refute this conjecture.

## 4. Related Work

There is a large body of work on slicing. This section considers only work related to empirical studies of program *slices*. In particular, research dealing with empirical studies of program *slicers* is not considered. The Unravel Project report, considered all slices of a single program [26]. Work by Orso et al [29, 30],. Sinha et al. [35] and Liang and Harrold [23], has also considered slice sizes, 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 of the order of thousands of lines of code, Orso, Sinha, Liang and Harrold were able to slice on every statement-level criteria. For those of 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 of programs considered and the number of programs. The authors believe that the work reported herein is at least one order of magnitude larger than any related previous study.

Beszedes et al. describe the approximation of static slices using the union of dynamic slices [7]. The authors consider three programs ranging from 5,592 to 37,539 lines of code and report an average reduction (portion of the code that was removed by slicing) of 18%. This stands in contrast to the over 70% average removal reported in Section 3. It is not clear to what the difference should be attributed. However, note that the slicing algorithm used by Beszedes et al. is based upon unioning dynamic slices and so it is markedly different from that used herein.

| | percent increase in | | |
|---|---|---|---|
| Program | average slice size | backward slicing time | forward slicing time |
| a2ps | 29% | 69% | 51% |
| acct | 330% | 418% | 301% |
| barcode | 36% | 21% | 19% |
| bc | 1% | 8% | -4% |
| byacc | 60% | 132% | 160% |
| cadp | 152% | 248% | 231% |
| compress | 6% | -25% | 0% |
| copia | 0% | -5% | 6% |
| csurf-pkgs | 58% | 76% | 59% |
| ctags | 21% | 116% | 85% |
| cvs | 11% | 23% | -13% |
| diffutils | 113% | 194% | 141% |
| ed | 1% | 19% | 15% |
| empire | 17% | 101% | 52% |
| EPWIC-1 | 193% | 258% | 242% |
| espresso | 56% | 124% | 103% |
| findutils | 87% | 169% | 118% |
| flex2-4-7 | 34% | 98% | 68% |
| flex2-5-4 | 39% | 81% | 53% |
| ftpd | 10% | 59% | 42% |
| gcc.cpp | 11% | 6% | -1% |
| gnubg-0.0 | 38% | 76% | 58% |
| gnuchess | 13% | 56% | 49% |
| gnugo | 9% | 56% | 55% |
| go | 0% | 40% | 37% |
| ijpeg | 16% | 14% | -6% |
| indent-1.10 | 21% | -7% | 35% |
| li | 0% | -29% | -29% |
| named | 6% | -12% | -7% |
| ntpd | 18% | 60% | 49% |
| oracolo2 | 74% | 120% | 152% |
| prepro | 75% | 123% | 156% |
| replace | 19% | 0% | -50% |
| sendmail | 6% | -28% | -24% |
| space | 76% | 120% | 160% |
| spice | 16% | 4% | -9% |
| termutils | 82% | 73% | 38% |
| tile-forth | 1% | -1% | 3% |
| time-1.7 | 15% | 0% | -67% |
| userv-0.95. | 63% | 99% | 96% |
| wdiff.0.5 | 146% | 100% | 113% |
| which | 40% | 0% | -20% |
| wpst | 139% | 274% | 304% |
| average | 50% | 77% | 66% |

**Figure 3.** The effect of calling context on slice size and computation time.

Liao et al. describe the SUIF tool [24], 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 data set studied herein.

Bent, Atkinson, and Griswold present 18 slices computed for six different programs ranging in size from 842 to 14,554 lines of code [6]. They report an average slice size (measured in non-blank non-comment lines of code) of 12% with the largest slice size being 35%. Two of their example programs appear in Figure 2 (although they may be different versions): for `compress` the three slices included 1.5%, 14%, and 41% of the program, and for `ed` the three slices included 16%, 4.2% and 14% of the program. In contrast to the work reported herein, Bent et al. reported considerably smaller slices. This is encouraging as the slicing criteria were picked by experts as representative. However, their sample size is small (18 versus 1,176,158 backward slices); thus the difference between their results and those reported herein may be accounted for by selection bias.

Two other studies by Atkinson and Griswold also include some slice size data. In the first, six slices (one from each of six programs measuring 2,692 to 1 89,043 non-blank lines of code) were found to include 16%, 27%, 6.7%, 31%, 31%, and 30% of the program [5]. The average of these is 24%, which is below the average given in Figure 2. Care is required in drawing conclusions from this comparison as the number slices actually taken is small. Also, the slices were constructed without regard for calling context, something which the study reported herein indicates can lead to marked increases in imprecision. In the second, slice size (not the focus of the work) is given in terms of the number of "three address statements in the slice [4]. As the initial number of three address statements for each program is not given, a percent reduction computation is impossible.

Agrawal and Guo study the tradeoff between context sensitivity and computation time for a collection of Java programs, ranging up to about 3,000 LOC [1]. The focus of their work was on the effect of different techniques for dealing calling-context. Formally, this 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 [21]. 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 [32] (*e.g.*, each "$(_i$" is matched with "$)_i$").

Krinke undertook an empirical evaluation of slicing (and chopping) [21]. 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 the programs SDGs). The average slice size ranges from 17.3% to 51.0% of the program with an overall average of 39.5%.

Agrawal and Guo found that ignoring context caused an 86% increase in slice size while speeding up slicing by a factor of 1.72 [1]. Their approach to handling context was expensive and, as it turns out, incorrect as shown by Krinke [21], who reports that ignoring context caused a 68% increase in slice size and a 40% increase in slice time. These numbers are less than the 50% increase in size and a 77% increase in time reported in Figure 3. The difference may be accounted for by slicing-criterion selection bias.

It is evident from the related work that there is a need for standardization in reporting statistics related to slices. For example, the research described above uses "non-blank non-comment" lines of code, "non blank" 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 base-line to which results can be compared. As this study has included data for a large number of slices, using this base-line, it is hopped the future studies will do the same to help to 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 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 examples took over 300Mb of memory or failed to finish in 8 hours. Using the SDGs that Codesurfer produces plus the collapsing of 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 Agrawal and Guo or Krinke. Doing so avoids potential selection bias that may be present in the studies of Bent et al., Agrawal and Guo, and Krinke. However, it may mean that the slices studied here could not be called "typical". Of course, deciding upon what constitutes a "typical" slicing criterion (and therefore a typical slice) is an inherently subjective, qualitative and application-dependent judgment and so the results here could be considered as base-line data against which to compare future work, which attempt to consider only "typical" slices.

## 5. Summary and Future Work

This study has shown that slicing is effective in reducing the volume of code to be analyzed. It presents a comprehensive study of over two million slices from 43 C programs that include over one million lines of code. Unlike previous studies, this study includes the slice on every executable statement in a large collection of programs.

Slicing using the SDGs produced by CodeSurfer, proved to be fast, progressing at an average pace of just over 3,000 KLOC per second. This certainly makes it reasonable for even very large systems. However, the focus of this study is not the speed of slicing, but rather the size of the slices. Overall, when structure fields are expanded, the average backward slices included 28.1% of a program, while the average forward slice included 26.3%. This means that techniques employing program slicing can expect to remove from consideration almost three quarters of a program.

The final experiment presented in Section 3 considered the effect of ignoring calling context. Doing so led to a 50% increase in the size of slices, and, somewhat surprisingly (given previous results) took 66-77% longer (because of the increased size). Future work is needed to verify or refute the hypothesis that the cause of this disagreement is an inefficient algorithm for dealing with calling context used in previous approaches. This future study is one of several suggested where the results from Section 3 disagreed with prior results.

The most intellectually interesting open problem suggested by this work is the formation of interprocedural SCCs that respect calling context. The formation of interprocedural SCCs would dramatically reduce the size of the SDG and thus improve the slice time. As an experiment, interprocedural SCCs were formed for `gnugo`. The graph size dropped from 278,766 vertices to 65,395 vertices with an associated drop in slicing time from 3,145 seconds to 434 seconds. Unfortunately, the formation of interprocedural SCCs by the naive extension of the algorithm that finds intraprocedural SCCs, obliterates calling-context information, and thus causes a loss in precision. Thus, interprocedural SCCs are only viable in situations where the benefits of speed outweigh the cost of precision. Ideally, an algorithm for forming interprocedural SCCs that preserve calling-context information will be found.

Finally, 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." Future work includes an investigation of slice size for different applications. The results presented herein provide a bench mark against which such application-specific results can be measured. Such work could both examine specific problem domains and produce results comparable to previous studies.

## Acknowledgments

GrammaTech provided *CodeSurfer*$^{tm}$ upon which the implementation is based. Alberto Pasquini provided the programs `oracolo2`, `prepro`, and `copia`. Gregg Rothermel provided the program `empire` and and Erin Dyet assisted with the presentation. Finally, the anonymous reviewers provided impressively detailed comments, which have both improved the presentation of this paper and suggested additional directions for future studies.

## References

1. Agrawal, G. and Guo, L., "Evaluating explicitly context-sensitive program slicing," *In Proceedings of the ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*(Snowbird, Utah), pp. 6-12 (2001).
2. Andersen, L.O., "Program analysis and specialization for the C programming language," Ph.D. thesis, DIKU University of Copenhagen (DIKU report 94/19) (May 1994).
3. Anderson, P., Binkley, D., Rosay, G., and Teitelbaum, T., "Flow insensitive points-to sets," *In Proceedings of the 2001 IEEE International Workshop on Source Code Analysis and Manipulation, (Florence, Italy)*, pp. 79-89 (November 2001).
4. Atkinson, D. and Griswold, W., "Effective whole-program analysis in the presence of pointers," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (November 1998).
5. Atkinson, D. and Griswold, W., "Implementation techniques for efficient data-flow analysis of large programs," *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, (November, 2001).
6. Bent, L., Atkinson, D., and Griswold, W., "A Qualitative study of two whole-program slicers for C," Technical Report CS2000-0643, Univ. of California at San Diego, (*a preliminary version appeared at FSE 2000)* (2000).
7. Beszedes, A. and Gyimothy, T., "Union slices for the approximation of the precise slice," pp. 12-20 in *Proceedings of the IEEE Conference on Software Maintenance,* (Montreal, Canada), IEEE Computer Society, Washington, DC (2002).
8. Binkley, D., Horwitz, S., and Reps, T., "Program Integration for Languages with Procedure Calls," *ACM Transactions on Software Engineering and Methodology* **4**(1) pp. 3-35 (January 1995).
9. Binkley, D., "Semantics guided regression test cost reduction," *IEEE Transactions on Software Engineering* **23**(8) pp. 489-516 IEEE Computer Society, (August 1997).
10. Binkley, D., "The application of program slicing to regression testing," *Information and Software Technology* **40**(11-12) pp. 583-595 (1998).
11. Binkley, D. and Harman, M., "Results from a large-scale study of performance optimization techniques for source code analyses based on graph reachability algorithms," $3^{rd}$ *IEEE International Workshop on Source Code Analysis and Manipulation (Amsterdam, Netherlands*, (27 September 2003).
12. Blair and Associates, *http://users.erols.com/bai/baiintro.html.*

13. Burke, M., Carini, P., Choi, J.D., and Hind, M., *Flow insensitive interprocedural alias analysis in the presence of pointers*. August 1994.
14. Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988)*, ACM SIGPLAN Notices* **23**(7) pp. 47-56 (July 1988).
15. Choi, J.D., Burke, M., and Carini, P., "Efficient flow sensitive interprocedural computation of pointer induced aliases and side-effects," *In ACM Symposium on Principles of Programming Languages*, pp. 232-245 (1993).
16. Emami, M., Ghiya, R., and Hendren, L., "Context sensitive interprocedural points-to analysis in the presence of function pointers," *In SIGPLAN Conference on Programming Languages Design and Implementation*, (1994).
17. Field, J. and Ramalingam, C., "Identifying procedural structure in Cobol programs," pp. 1-10 in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (*Software Engeneering Notes (SEN), ACM Press, N.Y., 1-58113-137-2, (September, 1999).
18. Gallagher, K.B. and Lyle, J.R., "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering* **SE-17**(8) pp. 751-761 (1991).
19. Grammatech Inc., *http://www.grammatech.com/ products/codesurfer/codesurfer.html*. 2002.
20. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 26-60 (January 1990).
21. Krinke, J., "Evaluation context-sensitive slicing and chopping," pp. 22-31 in *Proceedings of the IEEE Conference on Software Maintenance,* (Montreal, Canada), IEEE Computer Society, Washington, DC (2002).
22. Landi, W. and Ryder, B., "A safe approximate algorithm for interprocedural pointer aliasing," *In SIGPLAN Conference on Programming Languages Design and Implementation*, pp. 235-248 (June 1992).
23. Liang, D. and Harrold, M. J., "Efficient points-to analysis for whole-program analysis," *Springer Lecture Notes in Computer Science* **1687** pp. 199-215 ().
24. Liao, S., Diwan, A., Bosch, R., Ghuloum, A., and Lam, M., "SUIF Explorer," *ACM SIGPLAN Notices, Proceedings of the seventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* **34**(8)(May 1999).
25. Lyle, J. and Weiser, M., "Experiments on slicing-based debugging tools," in *Proceedings of the First Conference on Empirical Studies of Programming,* (June 1986), Ablex Publishing Co. (1986).
26. Lyle, J.R., Wallace, D.R., Graham, J.R., Gallagher, K.B., Poole, J.E., and Binkley, D.W., "A CASE tool to evaluate functional diversity in high integrity software," *U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD*, (1995).
27. Mock, M., Atkinson, D.C., Chambers, C., and Eggers, S.J., "Improving program slicing with dynamic points-to data," *Proceedings of the 10th International Symposium on the Foundations of Software Engineering Charleston South Carolina)*, (11/18-22/2002).
28. Ning, J., Engberts, A., and Kozaczynski, V., "Automated support for legacy code understanding," *Communications of the ACM* **37**(5) pp. 50-57 (1994).
29. Orso, A., Sinha, S., and Harrold, M.J., "Effects of pointers on data dependences," *9th IEEE International Workshop on Program Comprehension (IWPC'01), Toronto, Canada*, pp. 39-49 (May 2001).
30. Orso, A., Sinha, S., and Harrold, M.J., "Incremental slicing based on data-dependences types," *IEEE International Conference on Software Maintenance (ICSM'01), Florence, Italy*, (November 2001).
31. Redwine, S. and Riddle, W., "Software technology maturation," *Proceedings of the eighth IEEE/ACM International Conference on Software Engineering,* (London, UK, August, 1985), pp. 189-200 (1985).
32. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (New Orleans, LA, December 7-9, 1994)*, ACM SIGSOFT Software Engineering Notes 19, 5 (December 1994), pp. 11-20*, (1994).
33. Ruf, E., "Context-sensitive alias analysis reconsidered," *In SIGPLAN Conference on Programming Languages Design and Implementation*, pp. 13-22 (June 1995).
34. Shapiro, M. and Horwitz, S., "Fast and accurate flow-insensitive points-to analysis," *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (Paris, France)*, (January 1997).
35. Sinha, S., Harrold, M.J., and Rothermel, G., "System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control-Flow," *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pp. 432-441 (May 1999).
36. Steensgaard, B., "Points-to analysis in almost linear time," *International Conference on Compiler Construction*, (April 1996).
37. Ward, M., "The formal approach to source code analysis and manipulation," *In Proceedings of the* $1^{st}$ *IEEE International Workshop on Source Code Analysis and Manipulation (Florence, Italy)*, pp. 185-193 (November 2001).
38. Ward, M., "Program slicing via FermaT transformations," *"*$26^{th}$ *IEEE Annual Computer Software and Applications Conference (COMPSAC 2002) (Oxford, UK)*, pp. 357-362 (August, 2002).
39. Wilson, R. and Lam, M., "Efficient context-sensitive pointer analysis for C programs," *In SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-12 (June 1995).
40. Yong, S., S.Horwitz, and Reps, T., "Pointer analysis for programs with structures and casting," *In Proceedings of the SIGPLAN 99 Conference on Programming Language Design and Implementation (Atlanta, GA)*, (May 1999).
41. Zelkowitz, M., Wallace, D., and Binkley, D., "Culture conflicts in software engineering technology transfer," *In Proceedings of the 23rd NASA Goddard Space Flight Center Software Engineering Workshop (Greenbelt, Maryland)*, pp. 52-62 (December 1998).
42. Zelkowitz, M., Wallace, D., and Binkley, D., "Experimental validation of new software technology," *Accepted for publication as a chapter inSE Empirical Validation (Natalia Juristo and Ana Moreno eds.)*, (2003).
43. Zhang, S., Ryder, B. G., and Landi, W., "Program decomposition for pointer aliasing: a step towards practical analyses," *Proceedings of the 4th Symposium on the Foundations of Software Engineering (FSE'96)*, (October, 1996).