Stop-List Slicing

Keith Gallagher[*]
Computer Science Department
University of Durham
South Road
Durham DH1 3LE, UK
k.b.gallagher@durham.ac.uk

David Binkley
Computer Science Department
Loyola College in Maryland
4501 N. Charles St.
Baltimore, MD. 21210 USA
binkley@cs.loyola.edu

Abstract

Stop lists are used in information retrieval to reduce the size of language processing tasks by eliminating non-content words. Stop lists are a collection of stop-words, words that do not contribute information to the language processing task. Stop-list program slicing uses a stop list of variables to reduce the size of a program slice. In a fashion similar to selecting variables of interest for a program slice, the programmer selects variables that are not of interest. Stop-list slices are computed by removing certain data dependences for variables in the non-interest set. In order to further understand and assess stop-list program slicing as a reduction technique, we evaluated the size of reductions obtained by computing slices when dependences involving stop words are ignored during the computation of the slices on a collection of **C** *programs. In a suite of 42 programs of approximately 800,0000 source lines, over 600,000 slices were computed. Using a list of stop words reduced the size of the computed slices by an average of 34%.*

1 Introduction

Mark Weiser[12, 13] devised program slicing to aid program debugging. When debugging, the programmer has a particular variable in mind. In slicing argot, this particular variable is the variable that the programmer has determined to be "of interest." The task of any program slicing technique is to then find the other variables (and statements) that could impact the value of this variable. Two significant problems occur in computing program slices. The first is finding the slicing criterion, i.e., finding the particular variable of interest; the second is that the slices are often large and unwieldy.

While a program slice elides irrelevant computations, it may be that some of the (data flow) relevant computations included in the slice are not of interest to the programmer. For instance, in any piece of software there are variables that *do* the computation (*e.g.*, outputs), and variables that *help* to do the computation (*e.g.*, counters, indices, and temporaries). But, are all variables that are included in the program slice of equal importance to the debugger or comprehender? Could a programmer determine that a variable is *not* of interest and have computations that affect this variable "sliced" away too? Is the removed information necessary to show the comprehender in the first place? As part of a comprehension task could it be that just to "wrap one's head around" what is going on that assignments to control variables, or some other subset of slice's variables, could be sliced out, too? We investigate the reduction in the size of slices when these "helper" variables are omitted from the computation of a program slice.

The problem of finding variables of *non-interest* would seem to be the same sort of problem as finding variables of *interest*. Locating these interesting / uninteresting variables is not the immediate focus of this work; we presume that both the variables of interest and non-interest have been obtained in some fashion. In this work, we are interested only in determining if pursuing this idea has merit by determining the reductions that could be obtained if uninteresting variables were tagged for exclusion in the same way the interesting variables are tagged for inclusion in a program slice.

The motivation for this work is to attempt to reduce the size and complexity of program slices. Previous work in this area has been concerned with devising ever more efficient data structures and algorithms to

[*]On sabbatical leave from Loyola College in Maryland, USA

obtain more precise slices. We consider an alternative approach which can be informally described as taking the slices as emitted from a high quality program slicer and post process them to see if further reductions are possible. Conceptually, this is akin to the technique used [5], in which slices of differing criteria were combined by simple set equivalence.

In this paper we borrow a concept from information retrieval, that of *stop words*, and present evidence that using a set of stop words whilst computing a program slice reduces the size of a slice enough that is indeed worth the effort. First, we present background, motivation and support for the application of stop words to program analysis. Then we present an operational definition and discussion of a stop-list slice via a collection of examples. This discussion is followed by a description the techniques used to compute the stop-list slices. The collected data is summarized and followed by a discussion of how this effort fits into the slicing corpus.

The contributions of this paper are a definition of a stop-list slice and an examination of 42 **C** programs that shows that stop-list slicing is a viable technique for program analysis. Stop-list slicing is a way to "slice the slice" and thereby can aid comprehenders in the same way that slicing does: removal of unneeded, unwanted, or irrelevant information.

2 Background

This section begins with a short discussion of stop-lists; discusses how they might be used in program analysis; and introduces the data set used in the study.

2.1 Stop Lists

A stop list is used in information retrieval to reduce the size of an index.

> A stop list is a list of words that are excluded from some language processing task, usually because they are viewed as non–informative or potentially misleading. Usually they are non–content words like conjunctions, determiners, prepositions, etc. These are often called function words[10].

For instance, words such as "the", "of," "is," etc, need not be included in an exhaustive index of a document and can thus be ignored when constructing the index.

Here is the quotation of the previous paragraph after stop words have been removed:

> stop list list words excluded language processing task viewed non–informative misleading non–content words conjunctions determiners prepositions called function words

With the stop words removed the number of words in the quotation is reduced from 42 to 19, a reduction of almost 55%. *If one knows that this quotation has had the stop words removed, its sense remains, with a little work.*

2.2 Non-interesting Variables

Do programs contain the functional equivalent of stop words? In other words, are there variables in programs that meet the generic criteria of stop words? While the quotation in Section 2.1 notes that stop words may be misleading, we will ignore this possibility; if a variable is used in a program, we will assume that its use is not to mislead a program reader. Two questions are studied. First, "do uninteresting variables exist?" and second (assuming so) "how does elimination of assignments to these variables affect the slice size?"

The study of stop-list slicing begins with an analysis to determine *if* there are variables in programs that qualify as stop words. Such variables would need to be uninteresting to a software engineer. In addition, discounting such variables would have to have an effect on slice size (*i.e.*, the quantity of code a maintainer must consider). To study these questions, the suite of programs shown in Table 1 was analyzed. For each program, the table provides two measures of program size, first, as reported by the UNIX word count utility `wc`. While line counts as reported by word count are useful when comparing with past studies, they provide a rather crude measure. To provide an alternative estimate of program size, the third column reports the number of non-blank non-comment lines of code as reported by `sloc_count` [14].

3 Stop List Slicing

Stop-list slicing is the application of a stop list to the slicing computation. The selection of variables to go on the stop-list is akin to the selection of variables with respect to which to slice. In some sense, their selection can be considered "the dual" of slice variable selection. When program slicing, an engineer selects variables "of interest;" when stop-list slicing, the engineer also picks a set of variables "of *non*-interest."

To compute a stop-list slice, we first obtains a list of stop words (identifiers). Before computing a slice

Program	Size (Loc)	
	`wc`	`sloc`
a2ps	63,600	40,222
acct	10,182	6,764
barcode	5,926	3,975
bc	16,763	11,173
byacc	6,626	5,501
cadp	12,930	10,620
compress	1,937	1,431
copia	1,170	1,110
csurf-pkgs	66,109	38,50
ctags	18,663	14,29
cvs	101,306	67,828
diffutils	19,811	12,705
ed	13,579	9,046
empire	58,539	48,800
EPWIC-1	9,597	5,719
espresso	22,050	21,780
findutils	18,558	11,843
flex2.4.7	15,813	10,654
flex2.5.4	21,543	15,283
ftpd	19,470	15,361
gcc.cpp	6,399	5,731
gnubg-0.0	10,316	6,988
gnuchess	17,775	14,584
gnugo	81,652	68,301
go	29,246	25,665
ijpeg	30,505	18,585
indent	6,724	4,834
li	7,597	4,888
named	89,271	61,533
ntpd	47,936	30,773
oracolo2	14,864	8,333
prepro	14,814	8,334
replace	563	512
sendmail	46,873	31,491
space	9,564	6,200
spice	179,623	136,182
termutils	7,006	4,908
tile-forth	4,510	2,986
time	6,965	4,185
userv	8,009	6,132
wdiff	6,256	4,112
which	5,407	3,618
wpst	20,499	13,438
sum	1,156,546	824,935
average	26,896	19,185

Table 1. The subject programs with simple line counting metrics.

using the standard graph reachability algorithm [8], all data dependences that originate from simple assignments to these identifiers are removed from system dependence graph (the underlying representation used by the slicer [4]). The kinds of assignment that qualify as "simple" are described in Table 2. In the next two sections, we show first, by example, that stop-words, *i.e.*, variables that are *not* relevant to the computation), exist in production code, and then we show, in Section 5, the potential affect on slice size that their removal has.

Deleted Assignments	
`v = ...`	`( v = ...`
`v++`	`v--`
`++v`	`--v`
`*v++`	`*v--`
`v <op>= ...`	`v[...] = ...`

Table 2. Stop List Statement Types

4 Examples

This section presents four examples that illustrate the kind of variables we expect to be placed on a stop-list. We start with a traditional example. Consider the "usual suspect," *Wordcount*, shown on the left of Figure 1. The center of the figure is the program slice on variable `nw`, the *number of words*, at the last statement, which includes definitions and references to the variable `inword`, the status variable that indicates whether or not the scanner is advancing over white space, and to `c`, the input variable. The slice omits 5 statements from the original program.

The right of Figure 1 shows the corresponding stop-list slice computed using the stop list of `c` and `inword`. It thus ignores assignments to these variables. Clearly we have lost execution semantics, for now the program is effectively equivalent to `while (<constant>) { ...}`; indeed, it will not even compile with the declarations removed. In a display environment, the sliced statements might be dithered to indicate that they were elided via the stop list. But note that by simple line counting, we have reduced the slice size by 31%, from 26 lines to 18.

The question arises: is the fragment on the right of Figure 1 comprehensible? We argue that it is *in the context of a comprehension exercise.* Return to the textual example of Section 2.1. If one is merely handed the collection of words in the second quote, confusion reigns. If one knows that this list of words

```
#include <stdio.h>
#define YES 1
#define NO 0
main()
{
 int c, nl, nw, nc, inword;
 inword = NO;
 nl = 0;
 nw = 0;
 nc = 0;
 c = getchar();
 while ( c != EOF )
  {
    nc = nc + 1;
    if ( c == '\n')
      nl = nl + 1;
    if ( c == ' '  ||
           c == '\n' ||
           c == '\t' )
       inword = NO;
    else
    if ( inword == NO )
      {
        inword = YES;
        nw = nw + 1;
      }
    c = getchar();
   }
  printf("%d "%d "%d \n",
             nl, nw, nc);
}
```

```
#include <stdio.h>
#define YES 1
#define NO 0
main()
{
 int c, nw inword;
 inword = NO;

 nw = 0;

 c = getchar();
 while ( c != EOF )
  {



    if ( c == ' '  ||
           c == '\n' ||
           c == '\t' )
       inword = NO;
    else
    if ( inword == NO )
      {
        inword = YES;
        nw = nw + 1;
      }
    c = getchar();
   }
  printf("%d "%d "%d \n",
             nl, nw, nc);
}
```

```
#include <stdio.h>
#define YES 1
#define NO 0
main()
{
 int c, nw inword;
 inword = NO;

 nw = 0;


 while ( c != EOF )
  {



    if ( c == ' '  ||
           c == '\n' ||
           c == '\t' )

    else
    if ( inword == NO )
      {

        nw = nw + 1;
      }

    }
   printf("%d "%d "%d \n",
              nl, nw, nc);
}
```

Figure 1. *Wordcount* **program on the left. The program slice on variable** `nw` **at the last statement of** *Wordcount* **program in the center. On the right a stop-list slice of the program with assignments to** `inword` **and** `c`**, and their respective declarations, removed.**

has had the stop words removed, a reasonable *guess* at its sense can be obtained. The same argument applies to a the fragment: we know it is a stop-list slice and in this context we can make some reasonable assumptions about the intent of missing variables, and the probable actions where the assignments are deleted.

Thus, when a comprehender knows that a stop-list slice is presented, we submit that eliminating *assignments* to the input variable, `c`, does not adversely affect comprehension of this slice. Nor does eliding references to the *assignments* to `inword` adversely affect comprehension of this program slice. That the loop depends on variable `c` is easily seen; likewise, the assignment to `nw` depends on `inword`. This is because control dependences are not removed from the stop list slice, just simple assignments.

Before presenting the second example, we switch to a more precise measure of slice size. To introduce the fundamental concepts of stop-list slicing and illustrate the sizes of reductions obtained, the proceeding example counted statements. This technique of text comparison and line counting does not extend to the larger programs analyzed. Thus, in subsequent examples of this section and in the next section, we will use vertex counts in the System Dependence Graph [8] (SDG) for measuring the percent reduction, rather than statement counts. For ease or presentation, we will continue to present snippets of code (rather than dependence graphs) to illustrate the stop-list slices.

The second example is shown on the left of Figure 2,

```
main()                                          main()
{                                               {
    int i;                                           int i;
    int number;
    int max = 0;                                     int max = 0;
    scanf("%d", &number);

    while (1 << max <= number)
    {
        max++;
    }

    for( i = max - 1; i >= 0; i--)                   for( i = max - 1; i >= 0; i--)
    {                                                {
      int current_digit = 1 << i;                      int current_digit = 1 << i;
      printf("%c",                                     printf("%c",
       current_digit <= number ? '1' : '0');            current_digit <= number ? '1' : '0');
      if (current_digit <= number)                     if (current_digit <= number)
          number = number - current_digit;                 number = number - current_digit;
    }                                                }
    printf("\n");                                    printf("\n");
}                                               }
```

Figure 2. The fragment to the right shows the stop-list slice of the fragment on the left using a stop-list of {`i, number`}.

which writes out the binary representation of the value received as input. The first loop serves only to compute the number of iterations of the second loop. Placing `i` on the stop list causes the stop-list slice to exclude the first loop; adding variable `number` to the stop-list removes the `scanf`. The stop-list slice of this fragment is shown on the right of Figure 2. The reduction in this instance is from 19 to 11, 42%.

The third example, shown in Figure 3, is from the utility *slowcat* [3]. It pauses while "cat"ing a file after a certain number of bits have been output; thus, "cat"ing the file slowly. Within the main loop pauses are inserted after a certain number of bits have been output. The main input-output loop is preceded by standard command line processing.

The main loop of *slowcat* is

```
while ((c = getc(infile)) != EOF) { ... } .
```

The slice on this loop includes 18 vertices while the slice on the counter increment "`bits_read += 8`" includes 21 vertices. The stop-list slice using `c` as the stop-list taken with respect to "`bits_read += 8`" includes only 9 vertices (a 57% reduction). What is being excluded here is opening the file, deciding the file name, etc. The reduction occurs because `c` has a data dependence on `infile` and data dependences on `c` are ignored.

```
int arg_ptr_index = 2;
 /* first arg is required infile name */

while (arg_ptr_index < argc)
{
  if (!strcmp(argv[arg_ptr_index], "-o"))
   {
     arg_ptr_index++;
     filename = argv[arg_ptr_index];
   }

   [[ check other arguments ]]
   ...
 }
```

Figure 4. Argument Processor

Finally, Figure 4 is shows a program fragment (extracted from a large system) which does some command line argument processing. The slice on the loop `while (arg_ptr_index < argc)` includes 50 vertices. The slice on `filename = argv[arg_ptr_index]` from

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define DEFAULT_RATE 14400
#define DELAY_POINT 256

void main(int argc, char* argv[]){
  FILE *infile;
  long rate = DEFAULT_RATE;
  int bits_read = 0;
  int c;
  unsigned long delay_time;

/*************************************************/
  if ((argc < 2) || (strcmp(argv[1],"-h") == 0)){
    printf("usage:\n   %s file_name [bits/sec]\n",argv[0]);
    printf("   The default time is 14400 bits/sec.\n");
    exit(0);
  }

  if (argc >= 3){
    rate = atol(argv[2]);
    if (rate <= 200){
       fprintf(stderr,"%s: illegal rate %s\n",argv[0],argv[2]);
       exit(-1);
    }
  }
  infile = fopen(argv[1],"r");
  if (infile == NULL){
    fprintf(stderr,"%s: unable to open %s for read\n",argv[0],argv[1]);
    exit(-1);
  }
/*************************************************/
  delay_time = DELAY_POINT * 1000000 / rate;

  while ((c = getc(infile)) != EOF){
    putc(c,stdout);
    fflush(stdout);
    bits_read += 8;
    if (bits_read > DELAY_POINT){
      usleep(delay_time);
      bits_read = 0;
    }
  }
  fclose(infile);
}
```

Figure 3. Slowcat source. The stop-list slice eliminates the argument processing. The elided source is noted between the starred lines

the "`-o`" (output file name) option includes 52 vertices. Placing `arg_ptr_index` on the stop list (thus ignoring the dependence on `arg_ptr`, also essentially making the loop `while (<constant>) { ...}`, this slice includes 47 vertices. (The slice on the loop header contains only 6 vertices.) Thus, we get a 10% reduction from 52 to 47 vertices.

5 Results and Discussion

Having illustrated in the proceeding section that stop-list slicing is advantageous in real code, the next question to ask is what is the maximum reduction possible, which occurs when *all* variables are on the stop list. This is not the same as removing all data dependences as only data dependences associated with 'simple' assignments are removed. Those data dependences from non-simple assignments (*e.g.*, through pointers) and control dependences are not elided.

The maximal reductions for each program, when the simple (as noted in Table 2) are shown in Table 3. The table includes the number of slices taken and the average slice size using an empty stop-list and a "full" stop-list. The final column of the first presents the percent reduction for each program. Summary statistics over all program are presented in the last five rows of the table. Over all programs the reduction ranges from 8% to 77% with an average reduction of 34%.

Finally, the program *copia* was used as a case study to examine the expected reduction for a realistic stop-list (rather than all the variables in the program). Table 4 shows the average slice size computed using an empty stop-list, then with all variables on the stop-list, and finally with a representative stop-list as might be chosen by a software engineer studying *copia*. The variables on the representative stop list are shown in Table 5.

For *copia*, including all variables on the stop-list results in a 41% reduction in average size size. While, as expected, the average reduction obtained using the representative stop list was smaller, at 25% it still represents a significant reduction in average slice size. This indicates that presenting a stop-list slice to a programmer may be of interest.

5.1 Interpretation Context

There are two external threats to these results: program selection and slice selection. Most of the programs come from the open-source community. There are no event-driven, real-time or embedded systems in the sample. Thus, these results may not be extrapolated to these domains, without further analysis. The sample size assuages the concern that the sample does not represent "typical" programs. The slice selection threat is assuaged by taking *all slices*; this eliminates concerns that a bias may be introduced by a programmer selected criterion. However, it does raise the concern that computing all slices is not representative of engineering activity. In this case we argue that, as our sample comprises all slices, it would include *any* slice chosen at random.

The only internal threat to these results is errors that may be in in *CodeSurfer* itself, thereby compromising the data. To this we respond that *CodeSurfer* is a mature "industrial strength" tool.

6 Related Work

The program slicing tool *CodeSurfer*[4] has the ability to chase the data dependences of the system dependence graph. It does not have the ability, through the current user interface, to *ignore* selected dependences in the systematic way that stop-list slicing does. The *CodeSurfer* internal representation of the system dependence graph can be modified to ignore dependences by simply eliminating the selected dependences; this is how the data presented in Table 3 was collected. The *SeeSlice* [2] system has the ability to limit the graph edge distance considered by a slicer. The distance limitation permits the comprehender to "drill down" into a specific area (distance) of interest. Our work would integrate nicely into the *CodeSurfer* or *SeeSlice* environments. The only enhancement required would be to tell the underlying slicing engines to *ignore* selected data dependences.

The CANTO maintenance environment of Antoniol, et al. [1] uses an incremental technique to integrate software and architecture. CANTO can be used to *construct* stop-list slices, although it was not designed to do so. The construction of the slice is controlled by the comprehender. We just provide the stop-list slice.

Orso, et al., [11] use an incremental technique to expand slices in steps by using types to elide subtle data

Stop-list Size	Average Slice Size	Reduction in Percent
Empty	13,035	
All variables	7,723	41%
"Reasonable"	9,810	25%

Table 4. Average size of slices of program *copia* with various stop-lists. The "Reasonable" stop-list is given in Table 5

		Average Slice Size		*Average as Percent*		
Program	*Slices Taken*	*Empty Stop-List*	*Full Stop-List*	*Empty Stop-List*	*Full Stop-List*	*Reduction*
a2ps	58,280	26,937	21,747	46%	37%	19%
acct	7,250	826	498	11%	7%	40%
barcode	3,908	1,700	1,080	44%	28%	37%
bc	5,132	3,827	2,755	75%	54%	28%
byacc	10,150	2,407	1,346	24%	13%	44%
cadp	15,672	1,906	1,337	12%	9%	30%
compress	1,084	315	140	29%	13%	56%
copia	4,686	2,113	1,449	45%	31%	31%
csurf-pkgs	43,044	11,122	8,773	26%	20%	21%
ctags	20,578	12,427	9,762	60%	47%	21%
cvs	103,264	75,247	58,784	73%	57%	22%
diffutils	17,092	4,894	3,592	29%	21%	27%
ed	16,532	11,001	8,698	67%	53%	21%
empire	120,246	56,279	44,582	47%	37%	21%
EPWIC-1	12,492	1,817	419	15%	3%	77%
espresso	29,362	12,917	8,950	44%	30%	31%
findutils	14,444	5,369	3,698	37%	26%	31%
flex2-4-7	11,104	3,885	2,258	35%	20%	42%
flex2-5-4	14,114	3,996	2,367	28%	17%	41%
ftpd	25,018	12,630	7,174	50%	29%	43%
gcc.cpp	7,460	4,442	2,750	60%	37%	38%
gnubg-0.0	9,556	3,372	2,491	35%	26%	26%
gnuchess	15,068	8,084	4,759	54%	32%	41%
gnugo	68,298	33,331	29,205	49%	43%	12%
go	35,862	28,803	18,917	80%	53%	34%
ijpeg	24,028	9,734	7,019	41%	29%	28%
indent-1.10.0	6,748	3,496	2,129	52%	32%	39%
li	13,690	8,292	5,514	61%	40%	33%
named	106,828	58,939	44,675	55%	42%	24%
ntpd	40,198	16,026	12,234	40%	30%	24%
oracolo2	11,812	2,161	1,036	18%	9%	52%
prepro	11,744	2,110	989	18%	8%	53%
replace	1,734	162	104	9%	6%	36%
sendmail	47,344	22,792	16,406	48%	35%	28%
space	11,276	2,239	1,080	20%	10%	52%
spice	212,620	67,515	41,932	32%	20%	38%
termutils	3,112	1,136	575	37%	18%	49%
tile-forth-2.1	12,076	6,653	6,105	55%	51%	8%
time-1.7	1,044	165	113	16%	11%	31%
userv-0.95.0	12,516	3,515	2,441	28%	20%	31%
wdiff.0.5	2,420	373	240	15%	10%	36%
which	1,162	474	175	41%	15%	63%
wpst	20,888	3,547	2,702	17%	13%	24%
sum	626,646					
average	29,759	13,925	10,124	40%	28%	34%
max	212,620	75,246	58,783	80%	57%	77%
min	1,044	162	104	9%	3%	8%
stdev	40,339	19,530	14,545	19%	15%	13%

Table 3. The Data

RAND_SEED	_ALARM_CLOCK
_FILE_SYSTEM	_HEAP
_PROCESS_UMASK	adx
ady	dot_dot_dot
errno	fp
i	j
m	max
min	n
p	p1
ptr	q
q1	seed
temp_FILE_SYSTEM	temp__ALARM_CLOCK
temp__FILE_SYSTEM	temp__HEAP
temp_dot_dot_dot	temp_optind
temp_star_stderr	temp_star_stdin
temp_star_stdout	temp_star_stream
temp_star_strm	vm
y	z

Table 5. The "Reasonable" Stop-List for program *copia*

dependences and statements. The contribution of this work is a more accurate slice that regards the semantic information contributed by the data types of the variables under consideration. Our distinction from it is that we are not refining the slice to be more accurate; we are eliminating information to assuage information overload.

Program dicing uses the information that some variables fail some tests, whilst other variables pass all tests, to automatically identify a set of statements likely to contain the bug [9]. A program dice is obtained using set operations on backward program slices. Dices relate to this work insofar as they eliminate statements from program slices.

Decomposition slice equivalence can be used to significantly reduce the number of slice a programmer needs to comprehend, by forming equivalence classes of slices that were exactly the same, regardless of the slice criteria [5]. The slices computed by this technique are still large. The current work enhances the reduction by further reducing the size of the slice that must be comprehended.

7 Future Work

The results presented here are promising and have led us to formulate a research plan.

7.1 Control Stop-List Slicing

As noted in Section 3 this paper focuses on *data* stop-list slicing. It is also possible to consider *control* stop list slicing. This subsection lays out a conceptual framework for control stop-list slicing.

In the SDG, for the code fragment "`if (p) a = 1`", the assignment "`a = 1`" is control dependent on `if (p)`. Furthermore, a procedure is also control dependent on each call-site to the procedure. Thus, if `p` were a control stop word then the slice of

```
if (p > 0)
   a = 1;
```

would stop at "`if (p > 0)`" and not look for definition of `p`. Similarly, in the code fragment

```
a() { b(); }

b() { c(); }

c() { x = 1; }
```

the slice on the assignment "`x = 1`" includes the entry points for `c`, `b`, and `a` because of control dependences. Using a control stop-word list of `b`, the slice would stop at `b`.

7.1.1 Output Statements

In the system dependence graph, "`printf("%d", a)`" is represented by a call vertex (to `printf`), parameter vertices for "`%d`", `a`, and a vertex for the return value. The call vertex is only involved in control dependence, so having it on a stop list would not change the (data) stop-list slice as defined in Section 3. Thus, an application of control stop-list slicing is the `printf` statement. The variables referenced in a `printf` do not represent data stop words because the call to `printf` is actually a control point in the system dependence graph. Output statements have always caused difficulties in computing program slices. Output statements do not contribute to the value of the variable in a slice, but they certainly are of interest to a programmer considering a program slice. For output statements, we use the same approach as decomposition slicing [7], in which output statements are not added to the slice computation, per se, but added in at the behest of the programmer.

7.2 Locating Stop Words

We have deliberately avoided the question of finding candidate stop-words. Now that we have some data that shows that stop-list slicing is worth the effort, we will have to spend some time determining what variables are likely candidates. One incipient idea is using unchangeable variables from the decomposition slice on the variable of interest[6, 7]. The unchangeable variables are used in other computations and thus cannot be changed in the decomposition slicing software evolution model. Unchangeable variables certainly contribute to the computation in question, but their data and control flows are beyond the focus of interest. Eliminating the data (and perhaps control) dependences on these variables is a reasonable place to start.

A second approach is to use techniques from information retrieval. A simple enumeration of variable uses could be a starting place. For instance, in text processing one could create a list words ordered the number of occurrences as a likely list of stop words. Applying the same technique to program source may be fruitful.

7.3 User Study

We will consider a user study. The goal of such a study would be to determine the "comprehension loss," if any, that might occur when a user is presented with a program slice and with a stop-list slice on the same criteria. An empirical evaluation would be difficult for the usual reasons that threaten any user study: caliber of subjects; size of sample; size of program used; etc.

8 Conclusion

Program slicing was devised to assist program comprehenders and maintainers in their difficult task. The central premise of this work is that all variables are not of equal importance to a software maintainer or comprehender. There are certain idioms and patterns, that are repeatedly used and can be considered as background noise in a comprehension environment. Examples of these are `for`-statements and their associated counter and argument processing code. To further assist comprehenders and maintainers, we have shown how the size of the slice itself can be reduced without significant loss of information. Moreover, this lost information can be easily retrieved by returning to a typical program slice.

References

[1] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei, and E. Merlo. Program understanding and maintenance with the canto environment. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 72. IEEE Computer Society, 1997.

[2] T. Ball and S. Eick. Visualizing program slices. In *Proceedings of the Tenth International Symposium on Visual Languages*, 1994.

[3] R. W. Buccigrossi and E. P. Simoncelli. EPWIC: Embedded Predictive Wavelet Image Coder. http://www.cns.nyu.edu/ eero/EPWIC/.

[4] CodeSurfer. GrammaTech, Inc. http://www.grammatech.com/products/codesurfer.

[5] K. Gallagher and D. Binkley. An empirical study of computation equivalence as determined by decomposition slice equivalence. In *Proceedings of the 10th Working Conference on Reverse Engineering, WCRE–03*, 2003.

[6] K. Gallagher, M. Harman, and S. Danicic. Guaranteed inconsistency avoidance during software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 2003. To appear Dec. 2003.

[7] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):35–46, January 1990.

[9] J. R. Lyle and M. D. Weiser. Automatic program bug location by program slicing. In *Proceeding of the Second International Conference on Computers and Applications*, pages 877–882, Peking, China, June 1987.

[10] T. Pedersen. www.d.umn.edu/˜ tpederse/Group01/wordnet.html.

[11] A. Orso, S. Sinha, and M. J. Harrold. Incremental slicing based on data-dependence types. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 158–167, Firenze, Italy, november 2001.

[12] M. Weiser. Programmers use slices when debugging. *CACM*, 25(7):446–452, July 1982.

[13] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

[14] D. A. Wheeler. SLOC count user's guide, 2005. http://www.dwheeler.com/sloccount/sloccount.html.