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

Mark Harman
King's College London
Strand, London
WC2R 2LS, UK.
Mark.Harman@kcl.ac.uk

Abstract

Stop list slicing modifies traditional program slicing by augmenting the slicing criterion to include variables that are not of interest (in addition to those that are of interest). The dependence analysis of slicing is terminated when it encounters an assignment to a variable in the stop list. Stop list slicing focuses on the computation of interest, while ignoring computation that is known to be uninteresting. This has the potential to reduce slice size when compared to traditional forms of slicing.

In order to assess stop list slicing, the paper reports the results of a large scale empirical study into the maximum slice size reduction that can be achieved. The study concerned a suite of 42 programs of approximately 800KLoC in total. Over 600K slices were computed. Using the maximal stop list reduced the size of the computed slices by about one third on average. The paper also presents results from five case studies into the effects of stop list slicing.

1 Introduction

Program, slicing has been shown to be an effective source code manipulation technique, that can isolate parts of a program relevant to a particular computation. The slicing process removes irrelevant computation. However, 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 a programmer? Could a programmer determine that a variable is *not* of interest and have computations that affect this variable 'sliced' away too?

This paper proposes stop list slicing. In stop list slicing, the slicing criterion is augmented to contain two sets of variables. The first set is the set of variables from the traditional slicing criterion; these are the variables of interest that capture the computation to be slice out of the whole program. The second set is the stop list variable set; the variables that are considered uninteresting.

The problem of finding variables of *non-interest* is a similar kind 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 those of 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.

Recent empirical studies of slice size [4] have indicated that the typical size of a static slice is about one third of the size of the program form which it is constructed. However, the result also indicate a high degree of variance in slice sizes. The size of the slices obtained by the slicing process is the driving force that underpins all applications of program slicing. The motivation for stop list slicing is to attempt to reduce the size and complexity of program slices, while maintain their utility.

The paper introduces stop list slicing and presents results from three approaches to the evaluation of the technique. These evaluation studies aim to address the following research questions:

1. Can stop list slicing produce large enough reductions in slice size to be worthy of further consideration?

2. How does stop list slicing perform on a typical stop list?

3. Are the slices that result from stop list slicing any use?

To answer these 3 questions that paper contributes three related studies, one a large scale empirical study concerned with research question 1 above, backed up by the second two smaller cases studies which concern typical behaviour and the more qualitative questions of usefulness. Specifically the contributions of the studies presented in the paper are as follows:

1. Results are presented from a large scale empirical study into the largest possible reduction that can be achieved using stop l;ist slicing. The motivation from this study is Amdahl's law [1]: if the largest reduction that can be achieved is small then the technique is not worthy of further consideration,. Fortunately the results reveal that large reductions i slice size are possible using maximal stop lists. The results of this study therefore suggest that stop list slicing is worthy of further investigation.

2. The second evaluation study considers a case study in stop list slicing applied to the European Space Agency program `copia`. The results for maximal stop lists slicing are compared to those for a more 'typical' choice of stop list. For this case study, stop list slicing with respect to a more typical stop list variable set reduced slice size by about one quarter.

3. The final case study considers four programs for which the slices are small enough to be presented in the. This sturdy seeks to present case study evidence for the more qualitative question :are stop list slices any use?

The rest of this paper is organised as follows: Section 2 introduces stop list slicing, while Sections 3, 4 and 5 evaluate it. Section 2 presents the results of a large scale empirical study into the maximum size reduction possible, using a maximal stop list, while Section4 considers the effect of a more 'typical' stop list on case study. Section 5 presents four smaller case studies that are sufficiently small to allow a more qualitative evaluation of the effect of stop list slicing in terms of the slices themselves. Section 5.1 presents threats to validity. Section 6 presents related work and Section 7 concludes.

2 Stop List Slicing

Stop-list slicing is the application of a stop list to the computation of backward program slices. 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.'

For example consider a statistics program that computes a number of statistics based on the sum of an input data set. If the average computation is incorrect, but the other statistics are not then the variables `sum` might be included on the stop list. In this case the results would be similar to the corresponding program dice [13].

In order to evaluate stop list slicing and top address the three research questions highlighted in the introduction, we implemented a stop-list slicer, as an extension to the popular sl;icing tool, CodeSurfer[7].

Before computing a slice using the standard graph reachability algorithm [11], all data dependences that originate from simple assignments to the stop list identifiers are removed from the dependence graph (the underlying representation used by the slicer [7]). The present system removes dependence for all definitions of each stop-list variable. A refinement would allows each top-list entry to include a scope in addition to an identifier name.

The goal of stop list slicing is to remove assignments to variables that are on the stop list and which cannot assign to any other variables. The current implementation of stop lists slicing considers only simple assignments. The kinds of assignment that qualify as 'simple' are described in Table 1. In essence, each allows the defined variable to be easily determined. More sophisticated analysis looking, for example, for singleton points-to sets, would allow further removal of dependences and thus a reduction in stop-list slice size.

As with traditional slicing, minimal stop list slicing (removing all computation on the stop list variables form the slice) can be shown to be uncomputable. The proof follows exactly the corresponding proofs structure used by Weiser [18] to show that statement minimal slices are not computable. Therefore, any attempt at removing stop list assignments will be necessarily a conservative approximation, in which some stop list assignments may remain in the slice.

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

Table 1. Stop List Statement Types

3 Quantitative Empirical Study

The results presented in this section address the research question:

> "what is the maximum reduction is slice size achievable using the proposed stop list technique?"

Notice that 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 subject programs used in the study are described in Table 2. The maximal reductions for each program, when the simple (see Table 1) 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 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%. The data from this table allows a simple application of Amdahl's Law [1] to indicate the potential benefit of stop-list slicing. For example, if the reduction were only 2 or 3 percent using full stop-lists, then the technique would be of little interest. The percentages in Table 3 represent significant enough reduction to warrant further study.

	Size (Loc)	
Program	`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 2. The subject programs with simple line counting metrics.

Program	*Slices Taken*	*Average Slice Size* Empty Stop-List	Full Stop-List	*Average as Percent* 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. Reductions possible using stop-list slicing

4 Quantitative Case Study

The previous section showed that stop list slicing can produce considerable reduction is slice size. However, the results concerned the largest possible stop list. This section presents a case study in the application of stop list slicing to a single program to obtain results that indicate the effects of stop list slicing for 'realistic' choices of stop list variables. For the case study the European Space Agency program, *copia*, was selected.

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*.

Two principal methods choosing stop words were used The first considers the identifiers in isolation. The stop list is constructed by including those identifiers that are obvious loop counters and temporaries. The second technique requires a not through examination of the code.

The first technique was used to identify the variables on the representative stop list as shown in Table 5. This list was chosen in a conservative fashion. The identifiers of copia were extracted and considered in isolation. Those picked for the representative stop-list (*e.g.*, errno and temp_optind) are unlikely to be of interested to the programmer slicing the program. Given a more focused task, additional variables would most likely be added to the stop list; thus, increasing the reduction for the 'reasonable' list.

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; however, at 25% it still represents a significant reduction in average slice size.

5 Detailed Qualitative Case Studies

The right of Figure 1 shows the corresponding stop-list slice computed using the stop list of `c`

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

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*

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>) { ...}`. 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*. A similar situation arises in information retrieval [14, 17]. If one is presented with a piece of prose from which 'stop words' have been removed, a reasonable *guess* at its sense can be obtained. The same argument applies to 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 programmer 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 assignments 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 data dependences for simple assignments.

Before presenting the second example, we switch to

This section presents four examples that illustrate the application of stop list slicing 'in the small' to give a more qualitative evaluation of its effect on slicing. We start with the program *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.

```
#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.**

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 is imprecise (*e.g.*, it is impacted by programming style changes); thus, in subsequent examples of this section and in the next section, we switch to using vertex counts from the System Dependence Graph [11] (SDG) for measuring sizes and thus 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, which writes out the binary representation of the value received as input. The first loop serves only to compute

```
 1 main()
 2 {
 3     char in_binary[32];
 4     int i;
 5     int number;
 6     int max = 0;
 7     scanf("%d", &number);

 8     while (1 << max <= number)
 9     {
10         max++;
11     }

12     for( i = max - 1; i >= 0; i--)
13     {
14       int current_digit = 1 << i;
15       if ( current_digit <= number)
16         in_binary[i] = '1';
17       else
18         in_binary[i] = '0';

19       if (current_digit <= number)
20           number = number - current_digit;
21     }
22     printf("%s\n", in_binary);
23 }
```

```
main()
{
    char in_binary[32];
    int i;

    int max = 0;






    for( i = max - 1; i >= 0; i--)
    {
      int current_digit = 1 << i;
      if ( current_digit <= number)
        in_binary[i] = '1';
      else
        in_binary[i] = '0';

      if (current_digit <= number)
          number = number - current_digit;
    }
    printf("%d\n", in_binary);
}
```

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`} and slicing with respect to the value of in_binary at line 22.

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* [5]. A utility that 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.

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 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.

5.1 Threats to Validity

There are two external threats to these results: program selection and slice selection. Most of the pro-

```
#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

```
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

grams 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

Previous work has also considered ways to reduce and refine slices, this section considers previous approaches to the problem. First, the *SeeSlice* [3] system has the ability to limit the graph edge distance considered by a slicer. The distance limitation permits the programmer 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. [2] 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 programmer. We just provide the stop-list slice.

Orso, et al., [15] use an incremental technique to expand slices in steps by using types to elide subtle data 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 [13]. 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 [10]. 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.

Steindl's work on a data flow-aware programming environment, supports among other things, the ability for the programmers to 'disable' selected aliases if 'he knows (by some oracle) that two variables will never be aliases.' [16] Similarly, the set of potential dynamic types can be restricted. At the lowest level individual dependence can be excluded.

Steindl also describes a *bidirectional feedback* approach in which not only does the tool feed information in the programmer, but the programmers feed information back to the tool [16]. Such an approach would work well with stop-list slicing where, when considering a slice, a programmer may gain insights on variables that are not of interest.

The inspiration for stop list slicing comes from Information Retrieval[14, 17], in which a stop list of words is used to to reduce the size of an index.

The closest previous work to stop list slicing is Krinke's work on barrier slicing, which allows *stop points* to be specified [12] within the System Dependence Graph. Slicing stops when it reaches a stop-point or barrier. Stop list slicing can be implemented in terms of barrier slicing by placing barriers at all assignment to a variable.

Therefore stop list slicing can be viewed as high level a method for specifying barriers for the barrier slicing method. That is, the programmer simply specifies program variables of interest so that the stop list form a part of the slicing criterion, whereas with (pure) barrier slicing, the programmer has to consider the dependence graph and the location at which to introduce barriers.

7 Conclusions and Future Work

The central premise of slicing is that all variables are not of equal importance for all tasks. Stop list slicing develops this premise: 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 program command-line argument processing code.

The paper used this motivation to introduce an augmented slicing criterion and associated slicing technique that allows uninteresting computation to be identified and removed from a slice. This 'stop list slicing' approach was evaluated in three ways: through detailed qualitative case study, through quantitative case study for a 'typical' choice of stop list and by large scale empirical study for a maximal choice of stop list. The results indicate that large reductions in slice size are possible and that the slices that results may be useful.

Future work will consider generalisation of stop list slicing. For example, both stop list slicing and Krinke's barrier slicing suggest the possibility of '*predicate* slicing' in which a predicate P is used to determine is a slice should stop. This would require a combination of forward and backward conditioning [6, 8, 9] Predicate P might specify a single dependence to be ignored, the set of all dependence through a particular point, all dependences related to some variables, or a combination of these and other techniques. To be effective when working with large programs, a variety of level of specifying refinements allows both precision work to be performed, and also bulk as each is appropriate.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[2] 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.

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

[4] D. W. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

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

[6] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):595–607, 1998.

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

[8] S. Danicic, M. Daoudi, C. Fox, M. Harman, R. M. Hierons, J. Howroyd, L. Ouarbya, and M. Ward. Consus: A lightweight program conditioner. *Journal of Systems and Software*, 77(3):241–262, 2004.

[9] C. Fox, M. Harman, R. M. Hierons, and S. Danicic. Backward conditioning: a new program specialisation technique and its application to program comprehension. In 9^{th} *IEEE International Workshop on Program Comprenhesion*, pages 89–97, Los Alamitos, California, USA, May 2001. IEEE Computer Society Press.

[10] 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.

[11] 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.

[12] J. Krinke. Barrier slicing and chopping. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 81–87, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[13] 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.

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

[15] 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.

[16] C. Steindl. Benefits of a data flow-aware programming environment. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 105–109, 1999.

[17] C. J. Van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.