FEATURE BENCHMARKS

This study concerns feature benchmarks chosen to evaluate various language features, with programs of different sizes and complexities. A list of these programs is given in Table 1.

The programs *Information_flow*, *Sum*, and *Wc* were those provided with *CodeSurfer* as test cases. The programs *Pointer*, *Callofcall*, and *Testcases* were written by the authors to assess additional critical test cases.

The main language features of the above programs are as follows:

- *Information_flow*: pointers, pointer casting, double pointers, data and control dependencies, global variables, function indirection,
- *Sum* and *Wc*: external libraries,
- *Pointer*: pointer flow,
- *Callofcall*: nested function calls, and
- *Testcases*: functions calls, local and global variables, call by reference, call built-in functions, dependence flow.

The programs we developed attempt to cover a range of test cases in C/C++ that are critical for most slicing methods [Binkley 1993; Bent, Atkinson, Griswold 2000; Binkley, Gold, Harman 2007]. The purpose of these programs was to exercise the slicing behavior and for in-depth analysis. In general, these language features included the following:

- Detecting function calls inside control blocks, such as *while, if, for*, etc. For example, function *f1* is called inside the *if-block* and *while-block* as shown:
    - *if (f1(v1, v2));*
    - *while (f1 (v1, v2) > 0);*
- Tracking multiple call depths, for instance the *f1* function calls the *f2* function with the slicing variable *v1*, and the *f2* function calls the *f3* function with the argument *v2* assigned to, and so on:
    - *void f1() { f2(v1); }*
    - *void f2(int v2) { f3(v2); }*
- Nested function calls, for instance where the function *f1* uses function *f2* as one of its parameters. We paid particularly close attention to this case, because most of the existing static slicing methods do not consider the types of parameters.

Frank Tip [Tip 1995] shows in his study of 22 static slicing approaches that the only approach that takes parameter aliasing into account was with Binkley [Binkley 1993]. Of the other 21 approaches only 9 could support inter-procedural slicing. For example, the intra-procedural algorithm produced by Weiser does not take into account which output parameters depend on which input parameters.

As we can see in the following example, the value returned from the function *f2* is used for the second argument of the function *f1*. The result is that the slice profiles of both functions are merged.

- *f1(v1, f2(v2), v3);*

- Distinguish between local and global variables having the same name, and detecting the flow of the data dependence between them. In addition, there are cases that include transitive dependence (*indirect dependence*).
- Call by reference parameter passing. This case supports pointer aliases.
    - *void f1(int &x, int y, int w);*
- Slicing over pointer variables. As shown below the pointer *p* is defined as a reference for the variable *v*. So the slice profile of pointer *p* should be part of the slice profile of *v*, since we can refer to *v* using the pointer *p*.
    - *int *p; p = &v; f(*p);*
- Detecting the calls of library functions whose implementation may not be available: for example calling function *abs ()* from the library *#include <cmath>*.

    In our approach, the code in external libraries is not analyzed as in the case of *CodeSurfer*. Specifically, we do not include any code in the analysis unless specifically provided. We try to keep the slice space at a minimum while still being useful in testing and maintenance tasks.

Table 1 shows the results obtained by *srcSlice* and *CodeSurfer* for the feature benchmarks. The *Program* column is the benchmarks used for comparison. The column *Slicing Criterion* contains the inputs used for the slicing process. For each program we used our experience as programmers to select slicing criterion that we felt expose the effects of the language features on each slicer's behavior. Additionally, in order to avoid any possible bias from our choices, we also computed the slice over all possible slicing criterions for each program.

CodeSurfer can take different combinations of slicing criterion including the *point* (line number), *variable name*, and *function name*. In order to unify the results obtained by both tools and since all feature benchmarks were in one source file, we adjusted the slicing criterion for *srcSlice* to use the criteria format (*f, m, v*). As seen in the last row of the table, for *CodeSurfer* the number of slices taken is 444. For *srcSlice* 79 slices were taken.

The program *Testcases* covers most of the language issues discussed above. The slices obtained by running both tools using the slicing criterion (*main, var1*) were observed to be correct; however, *CodeSurfer* included some global variables that did not have any dependence on the slicing variable.

Binkely et al [Binkley, Gold, Harman 2007] reasoned that this case due to the fact that the slice size in the SDG reports the global variables that modeled as a *value-result* parameters. Thus each global variable counts as a node in the SDG added at both the caller and procedure entry. In contrast, *srcSlice* ignores those variables in the returned slice. Table 1 demonstrates these results, as the slicing time and the slice size of *srcSlice* are smaller using both types of the slicing criterion. According to the definition by Hoffner [Hoffner 1995], the best slice should be the smallest correct slice. Manual checking of the slices produced by both tools showed that they were 100% correct; however *srcSlice* produced a smaller slice.

From Table 1 we can see that the slice size of *srcSlice* is consistently smaller than the ones produced by *CodeSurfer* (the average forward slice contained 45.2% of the program source using *CodeSurfer* and 34.1% using *srcSlice*) except for the program *Pointer* using the slicing criterion (*main, var1*).

A closer investigation of this program shows that for the sample code in Figure 1 (a), *CodeSurfer* has limitations in detecting the flow from pointer **p* in line 10 to the receiver argument *z* in function *f3*, which is assigned in the body of the function to pointer *zp* at line 3.

(a)	1.	`f3 (int z) {`
	2.	`int *zp;`
	3.	`zp = &z;`
	4.	`zp++;`
	5.	`}`
	6.	`main () {`
	7.	`int var1 = 1;`
	8.	`int *p;`
	9.	`p = &var1;`
	10.	`f3 ( *p);`
	11.	`}`
(b)		*file/f3/z/@index (1), slines {1,3}, pointers {zp}*
		file/f3/zp/@index (2), slines {2,3,4}, dvariables {zp}
		file/main/var1/@index (1), slines {7,9}, pointers {p}
		file/main/p/@index (2), slines {8,9,10}, cfunctions {f3@ (1)}

Figure 1 (a) Sample source code from *Pointer* program, (b) System dictionary with four slice profiles for the source code in (a).

Bent et al [Bent, Atkinson, Griswold 2007] describe this case as a gray area in the *CodeSurfer* algorithm, and defined it as "*handling undefined entities*". In particular a call of the form *f3 (&var1)* will not be treated as a possible definition of *var1*. Also, an uninitialized pointer will not be a member of any points-to set so any effects through it are not tracked. That is, the statements **p = var1; z = *p;* when slicing on *var1* will not add *z* to the slicing criterion, nor is there a warning.

However, this is not the case when slicing over all possible criterions, i.e., the number of slices taken is equal to 37. The slice size is equal to 25, and manual checking of the returned slice showed that *CodeSurfer* detects the lines from 1 – 4 using the slicing criterion (*f3, *p*) in line 10.

In contrast, *srcSlice*, as shown in Figure 1 (b), captures this case and included in the slice profile for each variable. This inability to track the chains of pointers in this particular example in *CodeSurfer* results in a slice with missing critical statements, especially when the slice includes aliases of the original variable.

The accuracy of the slices produced using *srcSlice* for the programs *Information_flow*, *Sum*, *Callofcall*, and *Wc* was identical to *CodeSurfer*. The slices produced using *srcSlice* was manually checked and found to be correct. The difference in the results obtained by *CodeSurfer* was due to retrieving unrelated statements; such as statements mentioned inside the blocks of *for* and *while* predicates and standard libraries. That is, *CodeSurfer* highlights statements that are not only semantically related to the slicing criterion but also syntactically related to the executable slice [Bent, Atkinson, Griswold 2007]. For example, *CodeSurfer* returned all relevant statements that modify or determine control flow statement in the *else* part of an *if* statement whose body was not in the slice.

As shown, for the settings chosen, *CodeSurfer* provides a correct slice with regards to data and control dependencies. The results also show that *srcSlice* produced accurate slices when compared to *CodeSurfer*. We note again that the settings used for *CodeSurfer* were to enhance accuracy and not performance.

Table 1.Feature Benchmarks results and comparison of *CodeSurfer* and *srcSlice*, time measured in seconds, slice size measured in number of statements, (%) columns are the slice size relative to LOC, (F) = number of files, (M) = number of functions, LOC = lines of code.

Program	Size			Slicing Criterion		CodeSurfer				srcSlice			
	LOC	**F**	**M**	**Method**	**Variable**	**Slices Taken**	**Slicing Time**	**Slice Size**	**%**	**Slices Taken**	**Slicing Time**	**Slice Size**	**%**
Information _flow	112	1	12	main	hi	1	1.481	32	28.6	1	0.978	27	24.1
				All Possible Criterions		149		66	58.9	22		48	42.9
Sum	21	1	2	main	sum	1	0.989	6	28.6	1	0.531	4	19.0
				All Possible Criterions		26		14	66.7	2		8	38.1
Wc	39	1	3	line_char_count	eof_flag	1	1.212	16	41.0	1	0.362	10	25.6
				Scan_line	i	1		7	17.9	1		4	10.3
				All Possible Criterions		46		24	61.5	9		19	48.7
Pointer	36	1	5	main	var1	1	1.519	11	30.6	1	0.358	15	41.7
				All Possible Criterions		37		25	69.4	8		17	47.2
Testcases	114	1	14	main	var1	1	7.662	50	43.9	1	0.641	44	38.6
				All Possible Criterions		156		79	69.3	24		56	49.1
Callofcall	24	1	3	main	var1	1	2.921	4	16.7	1	0.411	4	16.7
				All Possible Criterions		23		13	54.2	7		10	41.7
Total	**346**	**6**	**39**			**444**	**15.784**	**347**		**79**	**3.281**	**266**	
Average	**57.7**	**1**	**6.5**			**34.2**	**2.630**	**26.7**	**45.2**	**6.1**	**0.546**	**20.5**	**34.1**

REFERENCES

[Bent, Atkinson, Griswold 2000] Bent, L., Atkinson, D. C., and Griswold, W. G., (2000), "A Qualitative Study of Two Whole-Program Slicers for C", University of California at San Diego. A preliminary version appeared at FSE '00, Technical Report CS20000643.

[Bent, Atkinson, Griswold 2007] Bent, L., Atkinson, D. C., and Griswold, W. G., (2007), "A qualitative study of two whole-program slicers for C", pp.

[Binkley 1993] Binkley, D., (1993), "Slicing in the Presence of Parameter Aliasing", In Proceedings of the Third Software Engineering Research Forum, Orlando, Florida, pp. 261-268.

[Binkley, Gold, Harman 2007] Binkley, D., Gold, N., and Harman, M., (2007), "An Empirical Study of Static Program Slice Size", ACM Transactions on Software Engineering and Methodology, vol. 16, no. 2, pp. 1-32.

[Binkley, Gold, Harman 2007] Binkley, D., Gold, N., and Harman, M., (2007), "An empirical study of static program slice size", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 16, no. 2, pp. 8.

[Hoffner 1995] Hoffner, T., (1995), "Evaluation and comparison of program slicing tools. Technical report": D. o. C. a. I. Science, Department of Computer and Information Science, Linkping University.

[Tip 1995] Tip, F., (1995), "A Survey of Program Slicing Techniques", Journal of Programming Language, vol. 3, no. 0, pp. 121-189.