A Qualitative Study of Two Whole-Program Slicers for C[*]

Leeann Bent[‡] Darren C. Atkinson[†] William G. Griswold[‡]

[‡]Dept. of Computer Science & Engineering
University of California San Diego, 0114
La Jolla, CA 92093-0114 USA
{lbent,wgg}@cs.ucsd.edu

[†]Department of Computer Engineering
Santa Clara University
500 El Camino Real
Santa Clara, CA 95053-0566, USA
datkinson@clarku.edu

ABSTRACT

Recently, a few whole-program static slicers for the C programming language have been developed, permitting a variety of hypotheses about time–precision tradeoffs in program analysis for software engineering to be tested. This paper reports an initial investigation into these claims through GrammaTech's CodeSurfer and UCSD's Sprite research prototype, which represent two very different approaches in the program analysis tool design space.

First, it was found that algorithmic superiority tended to provide large improvements in relative precision in select cases. Second, a number of non-algorithmic design choices had a substantial and sometimes unintuitive influence on slice results. Third, considerable expertise and time was required to discern the reasons why a particular statement appeared in a slice, diminishing the slice's probable usefulness. These results provide recommendations for future tool design.

1. INTRODUCTION

Program slicing and the techniques that support it have been under investigation for nearly two decades. A few whole-program static slicers for the C programming language have been developed recently, permitting a variety of long-held hypotheses abouting slicing (and program analysis for software engineering in general) to be tested. Among these are claims about time–precision tradeoffs and the importance of using precise algorithms to achieve the best results.

This paper reports our initial investigations into these claims through the use of two whole-program slicers, GrammaTech's CodeSurfer and our Sprite research prototype. These slicers represent two very different approaches in the design space of program analysis tools. CodeSurfer uses algorithms of high precision, using tunability and precomputation to control and amortize their cost. Sprite favors less precise (and more efficient) demand-driven algorithms complemented by tunability features that reduce or compensate for losses of precision.

We first gleaned a comprehensive set of "microbenchmarks", or feature benchmarks, from the research literature to help expose hidden contributors to slicing results. Second, we ran a smaller number of production-quality "macrobenchmarks", or performance benchmarks, to investigate the consequences of each slicer's design decisions and uncover further issues. A number of quantitative and qualitative insights arose from our investigations:

- Algorithmic precision, coming at reasonable cost and possible loss of scalability, had an unpredictable effect on slice results. In one case, context-sensitive data-dependencies created a 570% difference in slice size. In other cases, context-sensitive data-dependencies had no effect. Distinguishing fields in the pointer analysis, on the other hand, had very little effect. This is possibly due to the low quality of the points-to information computed by both tools.
- Several non-algorithmic design choices had a large and sometimes unintuitive influence on the slice results. The choice of what to highlight, what constitutes a dependence, implementation decisions, library modelling, and the handling of incomplete programs are examples. Such choices have a small performance impact, but may overwhelm a user with apparent dependences or fail to lead her to points of interest in the code.
- Although both tools possess considerable programmability, they lacked adequate control over what portions of the program were sliced or highlighted. This lack made it difficult for us to divine why a particular statement is in a slice. This is problematic, given that a purpose of slicing is to deduce the possible causes of a variable having a particular behavior.

These insights both suggest a number of directions for future

[*]This research supported in part by NSF grant CCR-9970985

UCSD Technical Report CS2000-0643, May 2000 (revised). An earlier version of this report was submitted to SIGSOFT FSE 2000.

investigation and provide lessons for tool designers.

The following sections first provide some background on slicing, describe our experimental setup, and describe the slicers. We then report the results of our microbenchmarks and macrobenchmarks, provide some design recommendations for future slicing efforts, and conclude.

2. PROGRAM SLICING

A static *program slice* or *backwards program slice* of a variable reference is the set of expressions and statements in a program that *may affect* the value of that reference [15]. A *forward slice* of a variable reference is the set of expressions and statements that *may be affected by* that variable's value.

Program slicing has a number of potential applications. In the past, most of these have been unrealized (in part) because of the lack of mature tools. Backwards slicing was originally proposed as an aid to debugging, helping to answer the question "How did this variable get the wrong value?" Forward slicing might be applied to program change tasks, helping to answer a question such as "How will my change affect downstream code?" However, either might be used for program understanding, code certification and inspection (i.e., in security applications), program restructuring, testing (to ensure coverage), program differencing and specialization, and optimization citeCsurfWhitePaper.

Program slicing is an attractive basis for a comparative assessment of claims about performance and precision, because it has become an archetype of program analysis for software engineering. Moreover, it places severe demands on a program analysis infrastructure and the computer system on which it runs because it computes a transitive closure over the predecessor semantic dependence relation until a fixed point is reached. This experiment focuses on backwards slicing, in deference to its historic roots.

3. RESEARCH QUESTION & DESIGN

Recent work on the effect of more precise algorithms suggests that the benefits realized on small examples are not realized on larger systems [4, 13]. This study goes a step further by looking beyond small variances of design choices within a single system to the combined impact of the collected techniques that embody an approach to the design of a complete tool.

The motivating research question of our experiment was *"For production-quality programs, how do choices about the precision of the whole-program analysis algorithms employed in a tool affect the precision of the resulting slices and performance, and why?"* The focus on production-quality programs addresses, to the extent possible, external validity of the results: slicing will be most useful on applications that users count on for their correct and improved functionality. Our focus on performance as well as precision reflects the importance of having not only a precise result, but also a timely one.

A more qualitative area of investigation regards the effectiveness of the slicers. A slicer's usefulness is determined by its ability to solve programmers' problems. For example, if a slice is performed to uncover the determinants of a variable's behavior for the purpose of software maintenance or enhancement, then a programmer may want to know *why* a statement is included in the resulting slice. The statement might be included due to a subtle pointer dependence or an artifact of the slicing algorithm; the cause could determine maintenance actions on the variable. Although our "task" was to understand why one slicer returned a different result than another, the same question underlies the investigation: why is this statement in the slice?

Size alone is not an adequate measure of precision: omitting just a couple of spurious but critical statements from a slice could save a programmer significant time, even if a few less critical statements slip into the slice. Consequently, qualitative analysis of slice differences can provide additional insights on the benefits of precision and also reveal the source of the differences. Both slicers provide significant programmability in their interfaces. These features can help explore these differences as well as the benefits of these techniques in improving time–precision tradeoffs.

Because we are by necessity measuring the behavior of whole tools, not just their algorithms, there can be numerous intervening variables influencing what statements are returned in a slice and how long it takes. Additionally, because Sprite is a research prototype, while CodeSurfer is a commercial tool, there may be factors in implementation that are unaccounted for. Consequently, we allowed for the generation of additional hypotheses about determinants of precision and performance and also undertook additional measurements in order to tease out these extra variables.

To expose and control for intervening variables as much as possible, we wrote 37 small programs, many based on examples from the literature, that exercise a slicer in just one behavioral dimension. Running a slicer on these "microbenchmarks" distills its salient characteristics into a discrete set of micro-features that can help explain slicing results in larger programs.

For the subsequent "macrobenchmarks", we chose a small set of production-quality public domain programs of varying size that worked with both slicers. We performed three slices on each program, attempting to choose slicing criteria that both are dissimilar from each other and might be chosen by programmers in practice. Focusing on 18 slices permitted a detailed analysis of the underlying causes of the slice results.

Because Sprite's algorithms are generally less precise than

CodeSurfer's, settings were chosen for the slicers that ensured that CodeSurfer was at least as precise across all components of the analysis, permitting a clearer assessment of the impact of the algorithms that could not be controlled. Settings were also chosen to ensure safety (conservativeness) of the analysis (both systems possess settings that can result in an unsafe result). When time–precision anomalies emerged in the results, we changed the settings for either or both tools in an attempt to achieve a more desirable tradeoff, thus exposing the underlying cause of the anomaly. Additional details on the microbenchmark and macrobenchmark designs are provided in sections 5 and 6.

4. THE SLICERS

4.1 Sprite

Sprite v2.0b is a research prototype slicer for ANSI and K&R C that can be run from a basic GUI or the command line [4]. It favors efficient techniques at the expense of precision. To permit tool users to manage their own time–precision tradeoffs, Sprite provides a number of tunability features that either select among a number of algorithmic options or provide extra information about the program being sliced.

For pointer analysis Sprite uses Steensgaard's flow-insensitive, context-insensitive points-to algorithm, which is space-efficient and runs in nearly linear time [14]. It computes equivalence classes of variables that refer to the same (abstract) memory location.

Control flow is represented with intraprocedural control-flow graphs related by a separate call graph. Context sensitivity (distinguishing unique paths to a function call) is provided through a mechanism that permits the tool user to select the context depth upto recursion. The effects of function pointers are extracted from the points-to sets, which can be computed prior the call graph because the points-to analysis is flow-insensitive.

The data flow analysis itself is performed with bit-vector data flow methods. The method has been extended to correctly handle pointers to locals in the presence of recursion. To reduce space usage the variable vector has been factored to collapse a single sparse vector space into several denser spaces. These spaces themselves are represented with semi-sparse bit vectors.

To address scalability concerns, Sprite computes all structured program information on demand, and computes unstructured (i.e. hard to demand) information upon starting the first slice for a program. For example, CFGs of procedures and points to information is computed upon starting the first slice, while control dependencies and data-flow information is computed on demand during slicing.

The effects of libraries are modelled with skeletal function definitions provided with the tool. The included libraries are libc and libm.

Tunability parameters include: goto inclusion, context sensitivity, filtering of function pointer classes by ANSI or K&R type matching (this can be unsafe if the program is not type safe), distinguishing fields of structures as individual variables, slicing into the callers (or not), and declaration of additional memory allocators.

Sprite's limitations include a limited number of provided library models, and line-level highlighting of slicing results (rather than statements and expressions). Additionally, some features of Sprite are simply engineering decisions. For example, unless otherwise specified, gotos are processed in the slicing, but not included in the slice. Also, the implementor chose not to include variable declaration, and function header lines.

4.2 CodeSurfer

CodeSurfer 1.4p1 is a commercial slicing-based tool produced by GrammaTech. It employs a sophisticated user interface providing capabilities such as "surfing" the program's dependences, but can be run from a command line interpreter as well. Though CodeSurfer implements several different algorithms, the publicly available version favors precise, asymptotically expensive algorithms. CodeSurfer uses batch precomputation of program dependence information to amortize analysis costs during interactive use.

CodeSurfer uses a System Dependence Graph (SDG) [9] as the program representation, and a slice is computed as graph reachability by computing a predecessor closure over this graph. The SDG also provides full context sensitivity (upto recursion).

The points-to analysis for CodeSurfer is parameterizable according to precision (and consequently performance). The options are "a" or Andersen's cubic algorithm [2], which is potentially more precise than Steensgaard's algorithm [13]. "m" is minimal analysis, in which every pointer can refer to anything that has had it's address taken and data pointers can refer to all dynamic memory. Finally, no pointer analysis can be specified (via the "none" option). A version of Steensgaard's algorithm is available as well ("s"), but not publicly. CodeSurfer's points-to analysis does distinguish the fields of structures in some cases (see microbenchmarks).

Other pointer analysis options include whether to treat calls to `malloc` a different location at each call site, and whether strings are modelled with a single location or unique locations. When calls to `malloc` are not expanded into seperate heap values, dynamic storage is not included in pointer analysis.

Libraries are modelled with stub functions provided with the tool, designed to accurately reflect dependence. The libraries provided with CodeSurfer 1.4p1 are libc and libm. The libc libraries may be configured to model reads and writes to different discrete locations used as sources and sinks (called discrete), or to treat the file system as a monolithic entity. (Sprite's library models are monolithic.)

CodeSurfer's limitations include an inability to stop a slice from proceeding into the callers. In contrast to Sprite, it can highlight the exact expressions or statements on a line that are part of a slice.

5. MICROBENCHMARKS

The microbenchmark programs are designed to test an algorithm's handling of one aspect of a language feature, coding style, or feature interaction. Consequently, they can verify or ascertain which algorithm a slicer is using. Because of their small size, the results can be analyzed exhaustively, also helping to reveal bugs in a slicer or oversights in how we are using it. The benchmarks test the following properties, divided roughly according to correctness and features:

- *Correctness properties*: interprocedural effects [9], handling of unstructured control flow (`break`, `continue`, `goto`) [1, 6], array definitions are preserving definitions, assignment through pointers, correct handling of function pointers, tracking of pointers through casts, tracking of pointers through structure casts, symmetry of array indexing [4], correct handling of external (imported) variables and functions, parameter passing of a pointer to a local variable in the presence of recursion [4], pointer arithmetic within a structure variable, handling of uninitialized pointers, and modelling of effects of I/O (e.g., on file descriptors).

- *Features*: capturing the effects of embedded halts and the like on control dependences [8, 11],[1] slicing into callers, context sensitivity of pointer analysis, context sensitivity of flow analysis, flow-insensitivity of pointer analysis, Steensgaard versus Andersen flow-insensitive pointer analysis [13], distinguishing of structure (or union) fields as separate abstract memory locations [16].

For both the microbenchmarks and macrobenchmarks, both tools were used with their "default" settings unless otherwise noted (below and in Section 6). The Sprite default settings are safe and favor performance: minimum context sensitivity, no function pointer filtering, and not distinguishing structure fields. CodeSurfer's default settings are unsafe and generally favor precision: Points-to analysis is performed with the "a" algorithm, each malloc call site is modelled as a unique location, but string literals are modelled as a single entity; the file system is modelled non-monolithically. For CodeSurfer, we used the *non-default* monolithic file system modelling, and the *non-default* many-strings option. We chose monolithic I/O modelling because non-monolithic modelling is unsafe and Sprite models I/O with a monolithic file system. We chose many-strings because that is closer to what Sprite uses (Sprite models strings using a unique location for each unique string), ensuring that our use of CodeSurfer is at least as precise as Sprite, permitting the benefits of its context-sensitive data dependence analysis and superior points-to analysis to be clearly observed. The practical implications of using the non-default many-strings option is discussed in Section 6.

5.1 Sprite

In the category of features, Sprite behaved as expected. Sprite models unstructured control flow and weak semantic dependences due to embedded returns, but not program exits. However, the microbenchmarks revealed several surprising behaviors in the "gray area" between correctness and features. They are not bugs *per se*, but may have unintended consequences for a tool user.

One gray area was in the handling of undefined entities. For one, a call to an undefined function yields a warning of the missing function definition, but there is no attempt to model the likely effects of such a call. For example, the call `x = f(y)` for undefined function `f` when slicing on `x` will not include the call in the slice, and `y` is not added to the slicing criterion despite the likely (albeit not guaranteed) semantic dependence. Also, a pointer that is nowhere initialized will not be a member of any points-to class, so any effects through it are not tracked. For example, for uninitialized pointer `p`, the statements `*p = x; y = *p;` when slicing on `y` will not add `x` to the slicing criterion. Nor is there a warning. Sprite does not include the library models by default; they must be explicitly included in the list of files for the project.

Another gray area was a set of decisions about what to highlight in a slice. For one, Sprite does not highlight global variable declarations or declarations of uninitialized locals. Global declarations are always initialized and are an actual semantic effect that should seemingly be highlighted. Although uninitialized local declarations are technically not part of the slice, a programmer seeking a bug may wish to be directed to such a declaration as it might be the source of a bug. Sprite also chooses not to highlight control flow statements such as `else`, `case`, `break`, `continue`, `goto`, goto label and `return`, although it models their *effects* correctly. When the goto option is enabled, semantically significant goto's are highlighted as expected.

[1] To some users, handling weak semantic dependences—especially those due to an intentional call to the `exit()` function—could be considered essential. They largely represent the fact that a statement in the slicing criterion is *not* executed if the halt is executed. We include them under features because there is not a direct control flow or data flow from the exit to the dependent statement. Also, not all weak semantic dependences are easily computable; statements that infinitely loop or divide by zero are two examples.

5.2 CodeSurfer

In the category of correctness and features, CodeSurfer behaved as expected. It provided full context sensitivity of data dependences, distinguishing structure fields in variable references but not pointer references, and pointer modelling consistent with Andersen's algorithm. CodeSurfer models unstructured control flow and weak semantic dependences due to embedded returns, but not program exits. It does capture the analogous effects of a `return` statement appearing before the end of the function. As with Sprite, we encountered a few surprising behaviors in the gray area between correctness and features.

One gray area was in the handling of undefined entities. A call to an undefined function yields a warning of the missing function definition and partially models the possible effects of the call. For example, the call `x = f(y)` for undefined function `f` when slicing on `x` will include the call in the slice, and add `y` to the slicing criterion, presumably because it is likely that a function's return value is dependent on its input arguments. However, a call of the form `g(&x)` will not be treated as a possible definition of `x`. Also, a pointer that is nowhere initialized will not be a member of any points-to class, so any effects through it are not tracked. That is, for uninitialized pointer `p`, the statements `*p = x; y = *p;` when slicing on `y` will not add `x` to the slicing criterion. Nor is there a warning. Finally, CodeSurfer does not include the library models by default; they must be explicitly included in the list of files for the project.

In the gray area of what should be highlighted in a slice, CodeSurfer highlights code that is not only semantically related to the slicing criterion, but also syntactically related to the executable code in the slicing result. This produces something close to an "executable" slice, although the user interface permits some customization of what is highlighted. For example, CodeSurfer highlights variable declarations regardless of whether they are initialized. CodeSurfer also highlights all relevant statements that alter or determine control flow, with the exception of goto labels.

5.3 Discussion

Both slicers behaved largely as expected, although we were surprised at both the diversity of handling gray areas and the level of expertise required to use both tools.

Given that a slicer assists in changing code, a user desires to have all potentially relevant code highlighted without necessarily being overwhelmed with information. Sprite's uniform highlighting of declarations could be a problem in this regard. In comparison, CodeSurfer's approach of highlighting generously, but providing filters, allows a user to customize the highlighting to fit the task at hand.

Both tools print warnings about undefined functions to the command line, yet the rest of the information is displayed via a GUI. This bifurcation of displayed information means that a tool user could easily overlook warnings that might have dramatic effects on slices that mislead the user. This was the case early in our studies, before we became sensitized to the issue. In some respects CodeSurfer's approach could be misleading in a subtler way because its partial solution to missing function definitions could mask the fact that something is wrong. Many of the undefined functions we encountered were in fact from standard C libraries that are handled by each slicer's library models, yet it is also easy to forget to include these until the undefined function messages are noticed. A benefit of the library modelling mechanisms of both tools is that they are simply simplified C function definitions that approximate each function's effect on its parameters and return value: a (sophisticated) user can add their own models at incremental cost.

Program		LOC	Salient features
compress	1.29	842	Data transformer; no external libraries
wally	–	1519	Legacy AI go engine with backtracking
ispell	3.1.2	5794	Interactive, table-driven
ed	0.2	7084	Command interpreter; uses options libraries
diff	2.7	8584	Complex, tightly nested recursion; uses options libraries
enscript	1.6.1	14554	Data transformer

Figure 1: Macrobenchmarks. LOC **is non-blank non-comment source lines of code.**

6. MACROBENCHMARKS

6.1 Experimental Set-up

To ascertain the consequences of the algorithmic choices and other issues surfaced by the microbenchmarks, we selected six production-quality public-domain "macrobenchmarks". These programs are: compress (spec95 benchmark), wally, ispell (GNU software), ed (GNU software), diff (GNU software), and enscript (GNU software). They range range from 850 to 14500 lines of non-comment source lines of code (excluding header files) and embody a variety of applications and software architectures (Figure 1). These programs were run on a 440 mHz UltraSparc 10 with 640MB of real memory and 1.1GB of virtual memory running the Solaris operating system.

Choice of slicing criteria. To increase the external validity of our study, it is desirable to select slicing criteria that reflect those used in practice. Slicing, however, is an emerging technology, so there are no operational profiles for slicing or a characterization of a "typical" slice. Indeed, it may be infeasible to acquire valid profiles until tool designers better understand the performance–precision tradeoffs and feature choices that would likely impact the way a program slicer is used. A slow slicer, for example, could discourage speculative, exploratory uses; an imprecise slicer might lead a programmer to subset the program before slicing on it.

Consequently, we used our own experience as programmers to choose slicing criteria that we felt might be used in practice. We also tried to vary the slicing criteria to expose the effects of program structure and language features on each slicer's behavior. Of course, this collection of slicing criteria does not represent a typical, average, or complete profile, so any general conclusions we draw must be considered preliminary and hypothesis-generating for future studies.

Normalization of slice results. Because of behavioral differences revealed in the microbenchmarks, we took a couple of steps to insure that the slicers' results could be meaningfully compared. In drawing conclusions from the results, the effect of these normalizations must be kept in mind.

Because Sprite did not traverse into the callers, we had to ensure that CodeSurfer did not. [2] To achieve this effect, for the (eleven) slicing criteria not within function `main`, we modified the programs processed by CodeSurfer to remove those calls to the procedure containing the initial slicing criterion that could be reached from entry procedure `main`. Although this changes the nature of the program somewhat, we verified that it had no adverse impacts on slice results.

Because the slicers report their findings differently, we had to do postprocessing on the results returned from each slicer. Because Sprite reports line-based results, CodeSurfer's syntax-based results are mapped to line numbers. To remove statements that are artifacts of highlighting (e.g. uninitialized declarations, or syntactic statements unrelated to the slice) rather than the slicing algorithms themselves, declarations and control structure artifacts are filtered from the output. This post-processing has a possible side effect of removing relevant statements from the slices. However, this postprocessing was implemented (and checked) carefully, and we have verified that any side effects from this are minimal. Also, since this is a partly comparative study and both programs were post-processed, it will not affect the relative results.

6.2 Slice Size

A *conservative* or *safe* slice includes all statements that could possibly affect the slicing criterion (and perhaps some that do not). A *minimal* slice is a conservative slice that contains no unnecessary statements. Since it is impractical to compute a minimal slice for large programs, for the purposes of comparison we use the intersection of the two slices returned from Sprite and CodeSurfer, called the *intersected* slice. This is a safe approximation of the *minimal* slice. The *relative safety margin* of a slice is the size of a slice divided by the size of the intersected slice. The safety margin provides a measure of the *relative* quality of a slice.

The results of running the slicers on 18 criteria over the 6 programs are presented in Figure 2. Focusing first on the slices in which no options are set, no obvious pattern emerges. It is apparent that for some slices, Sprite's results are considerably larger. Neither slicer's results are consistently contained within the other, nor does one slicer produce consistently smaller slices. However, on average Sprite's slices are larger. Upon closer examination, we observe that Codesurfer produced a high (greater than 1.50) safety margin on one slice (discussed below); Sprite produces a high saftey margin on five slices. Codesurfer produced a safety margin of 12.79 on ed:sflags:1009. Sprite produces a safety margin of 2.56 on wally:x:1798, 177.22 on ispell:cflag:857, 5.87 on ispell:preftype:727, 571.80 on diff:switch_string:629, and 1.54 on enscript:token:1881. The average slice size is 11.78% of the given program, with many (eight) small (5% or less) slices, and no slice larger than 35% of the system. Sprite produces a smaller slice than Codesurfer on two criteria, but by a small margin on one of these. The slice sizes were approximately equal on three slices, and Codesurfer was smaller on the remaining 13.

A more interesting question is *why* the slicers produced the above results.

Sprite.

The discernible causes of Sprite's safety margins are varied. The most obvious cause of Sprite's imprecision is the lack of context. As Figure 6.1 shows, increasing the context for the five slices with large safety margins reduced them to within the 1.50 safety margin. An interesting thing to note is that while increasing the context improves Sprite's results, using the worklist algorithm (instead of the default iterative algorithm) gives similar results. This is because the worklist algorithm converges locally, instead of globally. This means that it considers fewer unreachable paths (or contexts) during Sprite's default one-pass setting. The timings for the worklist algorithm are discussed in the Timings section below.

Because of early results, and the close examination of slices (particularly compress:new_count:241 which contains many I/O calls) we also knew that imprecision in the libraries might be an issue. A look at the library models confirms that the Codesurfer libraries are far more detailed than Sprite's. For example, a typical Sprite function model might contain six statements, whereas the CodeSurfer equivalent might contain 40. Consequently, we resliced four slices with larger safety margins using CodeSurfer's libraries and infinite context depth. These further reduced both the slice size and the safety margin in all four slices (bottom of Figure 6.1)).

The slices that were affected by context depth often increased

[2] Sprite v2.0b does slice into the callers, however, Sprite v2.0b is a beta version and slicing into the callers is still an unsupported feature.

						Sizes						Times					
Program		Slicing Criterion			Slicer	Intersection		Sprite		CodeSurfer		Sprite			CodeSurfer		
Name	LOC	File	Variable	Line	Options	Lines	Frac	Lines	SM	Lines	SM	Bld	Ld	Slc	Bld	Ld	Slc
compress	842	compress95.c	block_compress	390	–	13	0.02	13	1.00	13	1.00	0.23	1.52	0.01	3.94	1.07	0.00
		compress95.c	fcode	496	–	122	0.14	123	1.01	123	1.01			0.04			0.00
		harness.c	new_count	241	–	261	0.31	287	1.10	261	1.00			0.17			0.00
wally	1519	wally.c	es	1593	–	173	0.11	241	1.39	208	1.20	0.11	0.67	0.29	5.18	1.05	0.00
		wally.c	rvalue	2055	–	492	0.32	492	1.00	502	1.02			0.69			0.25
		wally.c	x	1798	–	77	0.05	197	2.56	78	1.01			0.19			0.02
ispell	5794	ispell.c	cflag	857	–	9	0.00	1595	177.22	9	1.00	1.74	2.13	1.80	32.77	1.54	0.43
		ispell.c	hashheader	745	–	334	0.06	492	1.47	341	1.02			0.37			0.12
		ispell.c	preftype	727	–	79	0.01	464	5.87	79	1.00			0.45			0.08
ed	7084	main.c	buf	1160	–	127	0.02	142	1.12	129	1.02	2.65	2.50	0.33	58.09	1.66	0.00
		main.c	err_status	302	–	2500	0.35	2539	1.02	2538	1.02			5.67			0.00
		main.c	sflags	1009	–	168	0.02	180	1.07	2214	13.18			0.55			0.24
diff	8584	analyze.c	changes	1010	–	1065	0.12	1191	1.12	1071	1.01	2.44	2.69	1.75	73.13	1.59	0.16
		diff.c	switch_string	629	–	5	0.00	2859	571.80	5	1.00			6.13			0.02
		diff.c	val	1073	–	1837	0.21	1911	1.04	1844	1.00			2.82			0.00
enscript	14554	main.c	real_total_pages	1567	–	3269	0.22	3410	1.04	3274	1.00	3.24	5.13	6.85	670.11	2.20	0.00
		psgen.c	token	1881	–	327	0.02	504	1.54	338	1.03			0.85			0.00
		psgen.c	x	878	–	2045	0.14	2142	1.05	2047	1.00			3.82			0.43
wally	1519	wally.c	es	1593	CSLibs, CD	173	0.11	174	1.01			0.10	2.54	98.76			
wally	1519	wally.c	x	1798	CD	77	0.05	77	1.00			0.09	0.68	4.96			
wally	1519	wally.c	x	1798	Worklist	77	0.05	77	1.00			0.13	0.66	0.06			
ispell	5794	ispell.c	cflag	857	CD	9	0.00	10	1.11			1.74	2.14	84.44			
ispell	5794	ispell.c	cflag	857	Worklist	9	0.00	10	1.11			1.75	2.13	0.24			
ispell	5794	ispell.c	hashheader	745	CSLibs, CD	331	0.06	419	1.27			1.72	4.15	21.10			
ispell	5794	ispell.c	preftype	727	CSLibs, CD	79	0.01	80	1.01			1.75	4.04	52.64			
diff	8584	diff.c	switch_string	629	CD	5	0.00	5	1.00			3.34	2.74	191.24			
diff	8584	diff.c	switch_string	629	Worklist	5	0.00	5	1.00			2.33	2.75	0.01			
enscript	14554	psgen.c	token	1881	CSLibs, CD	323	0.02	455	1.41			4.47	7.41	6.09			
enscript	14554	psgen.c	token	1881	CD	327	0.02	462	1.41			3.35	5.20	2.59			
enscript	14554	psgen.c	token	1881	Worklist	327	0.02	462	1.41			4.23	5.52	0.80			
compress	842	harness.c	new_count	241	One	261	0.31			261	1.00				3.74	1.10	0.00
wally	1519	wally.c	x	1798	One	77	0.05			78	1.01				4.58	1.07	0.00
ispell	5794	ispell.c	cflag	857	One	9	0.00			9	1.00				26.02	1.54	0.02
ed	7084	main.c	sflags	1009	One	168	0.02			2214	13.18				55.07	1.76	0.00
diff	8584	diff.c	switch_string	629	One	5	0.00			5	1.00				39.95	1.62	0.03
enscript	14554	psgen.c	token	1881	One	327	0.02			338	1.03				98.61	1.98	0.16

Figure 2: Slice Sizes and Times. LOC **is non-blank non-comment source lines of code;** Size **is the size of the slice in lines;** Frac **is the slice size relative to** LOC**;** SM **is the relative safety margin for a slice. The** CSlibs **option implies that the sprite slices were performed with CodeSurfer libraries; the** CD **option implies that context depth was infinite for Sprite. The** One **option implies the one-string option was used for CodeSurfer. All times are in seconds. Build and load times are per-program, not per-slice, so the first values hold for all slices.** Slicer Options **are those slice-time or build-time options deviating from the settings described at the beginning of Section 5, when applicable.**

in precision tremendously. For example, the slice ispell:cflag:857 is decreased from 1595 lines to 10, with an intersection of nine lines using infinite context depth. The slice diff:switch_string:629 was reduced from 2859 statements to 5, with an intersection of 5. The slice wally:x:1798 was decreased from 197 to 78 lines, 77 of which are in the intersection. All of these become almost identical to their respective CodeSurfer slices.

For some slices the CodeSurfer libraries and the addition of context depth had similar effects. These slices were wally:es:1593, ispell:preftype:629, ispell:hashheader:745, and enscript:token:1881. The slice wally:es:1593, using CodeSurfer's libraries in conjunction with infinite context depth, reduces the slice to 174 statements with an intersection of 173 statements. Most of this gain is due to the use of CodeSurfer's libraries; using only CodeSurfer's libraries reduces the slice size to 180 statements, while using only context depth gives no reduction at all. [3] For the slice ispell:preftype:629 using both CodeSurfer's libraries and infinite context depth decrease the slice to 80 lines, with 79 statements in the intersection. Using infinite context depth alone only descreases the slice to 271 slices, while only using CodeSurfer's libraries decreases the slice to only 261. The slice ispell:hashheader:745 was decreased to 419 lines, with 331 lines in the intersection using both options. Only using context depth decreases the slice to 467, while the use of CodeSurfer's libraries only decreases the slice to 489.

The use of the structs option in Sprite had little to no effect on the slices it was used on (those in the Sprite customization section of Figure 6.1).

[3] Context depth and CodeSurfer library results are not necessarily additive. Additional context depth will also improve the precision of statements depending upon the library models.

Using customizations in Sprite permitted attributing a cause to dependences (e.g., unrealizable paths due to conflation of calling contexts) by changing options rather than by inspecting the code or Sprite's dependence information by hand. On the other hand, it requires expertise to get improved results quickly, since there are many plausible (combinations of) customization options to try. Turning on all the options, while affordable on these programs, does not scale [4].

CodeSurfer. Much of CodeSurfer's slice safety margin can be attributed to its choice of control dependences as affected by by embedded returns, breaks, and gotos. For example, ed:sflags:1009 appears in a case statement that can be approximated as follows (this code does not appear in a loop and its function has no callers):

```
[1]  switch (c = *ibufp++) {
[2]    case 'z':
[3]      if (display_lines(sa, min(al,sa+r), gflag))
[4]        return ERR;
[5]      gflag = 0;
[6]      break;
[7]    case '!':
[8]      sflags = get_shell_command();
[9]      break;
[10] }
```

Because of the way CodeSurfer computes control dependences [5], slicing on `sflags` on line 8 includes lines 3 and 4 in the slice, on the presumption that a true conditional in line 3 results in a return from the function, and line 8 is never executed. Because the conditional in line 3 contains a function call to `display_lines`, the slice also contains the lines from that function that may determine its return value, and so forth. However, lines 3 and 4 have no influence on `sflag`'s value. The reason is that the unconditional `break` on line 6 ensures that regardless of the return value from the `display_lines` call on line 3, line 8 can never be reached; thus there is no control dependence generated by lines 3 and 4.

The above example results in a particularly large slice because the expressions in the included conditionals add variables to the slicing criterion that propagated into function calls controlling the conditionals. No other slice had this extreme behavior. When we removed all of the return statements in ed's case statement, all with `break`'s below them, CodeSurfer's slice dropped to around 200 lines. Although we are unsure of what control dependence algorithm CodeSurfer uses, Sprite computes control dependences from the dominance frontiers of the reverse control flow graph [7].

Besides this control-dependence issue, CodeSurfer, for the settings chosen, is generally more precise than Sprite, as expected. Given that this can come at a cost in build and slicing times (as discussed below), we reverted CodeSurfer to its default one-string, monolithic setting and reran a build and a slice from each program. The results lose no precision, and the builds are sometimes considerably faster, as discussed below in the timings section.

Program	LOC	CodeSurfer	Options
compress	842	1.34MB	1-string
wally	1519	2.19MB	1-string
ispell	5794	10.00MB	1-string
ed	7084	14.66MB	1-string
diff	8584	7.56MB	1-string
enscript	14554	27.39MB	1-string

Figure 3: Pre-built database sizes for CodeSurfer. LOC **is non-blank non-comment source lines of code.** 1-STRING **is the one-string build option (with monolithic libraries) for CodeSurfer.**

6.3 Timings and Database Sizes

Slicing can be broken down into the following phases of computation: a "build" phase which must happen each time the program is changed, a "loading" phase which must happen each time the tool is used, and a "slicing" phase, which must occur each time a slice is requested. While Sprite and Codesurfer both have these phases, they perform different behaviors during these phases. During Sprite's "build phase", the source files are parsed. These parsed files (.cpp files) are stored across tool invocations. This is represented as the "build" time in Figure 6.1. Precomputation for the program, including constructing the AST and CFG, and computing points-to information, is done on the first slice request. This is represented in Figure 6.1 as "load" time, because it must happen each time the tool is invoked for a particular program. Finally, Sprite computes control and data dependencies during "slicing". CodeSurfer, meanwhile, precomputes a large amount of information, storing a deep structure representation that contains a system dependence graph, data dependencies, control dependencies, and pointer information. Thus, the "build" phase for CodeSurfer consists of the time it requires to compute this information and write it to disk. When CodeSurfer slices a program, it must reload that information from disk (assuming the tool has been exited) before slicing. This is represented in Figure 6.1 as "load" time. Slicing may be performed any number of times for a particular load. Finally, "slicing" time is just the time it takes CodeSurfer to handle a particular slicing request.

Each of these phases has a potentially unique impact on a tool user, as the first only occurs when the program to be analyzed is changed, the second only occurs whenever a tool is restarted for a particular program, and the last will likely occur several times before the program is changed. Also, the runtime cost of these activities has a feedback effect that will influence frequency. If build or load times are slow, the programmer may attempt to use the slicer on stale versions of the program or hesitate to rebuild the database to change the tuning options to the build. Short slice times can encourage speculative use.

We captured build, load, and slicing time for all of the slices using TC-shell's built-in `time` command with the machine unloaded. Wall-clock time is reported since this represents the time a user is required to wait for her results (Figure 2). Although we modified each program slightly for CodeSurfer to prevent it from slicing into callers, we report build and load times for the unmodified program to permit comparing these numbers to Sprite's results. (These times are comparable for the modified programs, however.) Because, `time` commands on Solaris UltraSparc II's do not provide space usage, paging, or I/O data, we used CPU utilization and the sizes of the on-disk data produced by the two tools (Figure 3) as indicators of memory effects on time. On an unloaded machine, the amount of CPU utilization below 100% approximates the percentage of time lost to I/O events, including paging. Finally, all timings are taken without using the tools user interfaces; batch modes are used instead.

Builds. Build times for CodeSurfer range from 3.94 to 670.11 seconds for the settings we used. While these are up to 207 times slower than Sprite's build, CodeSurfer does most of it's work during build while Sprite does very little. In addition, excepting the enscript program, CodeSurfer's build times are on par with the full context depth slice times for Sprite (non-default setting). The outlying data point for build timings is enscript, taking 9 times longer than diff (a program half it's size). The many-strings, monolithic build for enscript required 130MB of memory. This is less than the system limits of 640MB for real memory and 1.1GB of swap. Sprite, meanwhile uses 23MB for the same slice (where Sprite's computation is performed). For CodeSurfer, CPU utilization figures indicate that 2.5% of the build time was lost to paging or I/O.

The CodeSurfer database for enscript at 27.39MB is the largest. The databases tend to grow with code size; enscript's database is almost twice as large ed's. However, diff's database is an exception. It is half the size of ed's database, even though ed has a smaller size. We are unsure why this is so.

The builds of the larger programs with the one-string option (CodeSurfer's default), show that some of the build cost is attributable to the many-strings option's more precise pointer modelling, which magnifies the cost of CodeSurfer's dependence and points-to analyses. Compared to the many-string builds, the diff and enscript one-string builds are 1.83 and 6.80 times faster respectively, whereas the ed build is only 1.05 times faster, suggesting the effects can be powerful but are highly dependent on the program.

While the one-string settings produce some noticable gains in build time, they produce little savings in database size. The one-string builds range 91.48% from 99.26% the size of the many-strings builds. The best improvement is seen in enscript, and a look at the enscript source confirms that it contains many strings used for character mapping.

An additional concern for CodeSurfer is the time to build the library. This must only be done (potentially) once for each setting of the library, however it does contribute additional cost to a build. For both settings (monolithic many-strings, and monolithic one-string) this build takes around 42 seconds.

Load time. Both slicers show fairly short load times that tend to scale with program size. Since, for both slicers, little is done in this phase, it is reasonable that these times should be short. For Sprite, this time includes PDG creation and pointer analysis; for CodeSurfer it seems to comprise file I/O to read portions of the SDG off of disk. Note that these times are fairly linear with respect to code size.

Slicing time. On a per-slice basis, CodeSurfer's slice time is incredibly small. In fact many slices come up with a 0.00 second slicing time, once load time is removed. This indicates that its precomputation strategy is successful at reducing slicing costs. This result also indicates that CodeSurfer probably does not read in much of its database before beginning graph traversal. The utilization numbers support this; they are generally between 45% and 70%. Thus, while I/O is a significant cost, it does not overwhelm the computation costs. The higher utilization for larger slices suggests that I/O costs could be amortized when many slices are performed in a single session The slice on ispell.c:cflag:857 is anomalous, with just 9 statements in it but taking 0.43 seconds (the measurements were repeated to ensure consistency). Given that this slice is derived from the program generating the second-largest database, there may be some non-locality in the slice that requires reading in a disproportionate amount of the SDG.

The timings of slices using CodeSurfer's default one-string setting in the build are similar to the many-strings option (with no change in precision). This is an indication that much of the work, and hence savings, can be seen in precomputation time. These results sugggest that slicing time can be reduced to almost constant, for programs of this size, with suitable investment in build time.

For Sprite, slicing time is closely related to slice size, probably due to the costs of data and control dependence computation during slicing. (While true for the programs here, [4] claims that this trend does not extrapolate to larger programs and slices.) The "tuned" slices appearing near the bottom of Figure 2 are considerably slower on the whole. Increasing context sensitivity to infinite depth potentially results in a quadratic increase in the context-graph (generated on demand) [4], largely explaining the increase. Additionally, enscript:token:1881 takes longer with the CodeSurfer libraries than with Sprite's. This is probably due to the Code-

Surfer libraries richer modelling. Using the worklist algorithm shortens these times considerably, while giving results idenical to infinite context depth. However, these results do not scale to larger programs. For diff:switch_string:629 the worklist algorithm slices in 0.01; for the enscript:token:1881 slice, Sprite runs in 0.80 seconds using the worklist algorithm. On ispell:cflag:857 the worklist algorithm takes 0.24 seconds, and on wally:x:1798 the worklist algorithm takes 0.39 seconds.

7. DISCUSSION

7.1 Tradeoff Analysis

Precision. For the settings chosen and given CodeSurfer's more precise data dependence and points-to algorithms, we had expected more of Sprite's relative safety margins to be larger, perhaps 2 and above on many slices. This turned out to be the case; in fact when Sprite's safety margins were larger, they were much larger (up to 572 times). However, all of Sprite's poor margins were reduced to within 1.5 by using CodeSurfer's library models and by increasing context sensitivity. (CodeSurfer's conservative handling of unconditional branches cannot be a contributor Sprite's low margins, as this affects only CodeSurfer's safety margin). The microbenchmarks reveal that CodeSurfer's context-sensitive data dependences and use of Andersen's points-to analysis can exhibit considerably greater precision. For example, Andersen's algorithm preserves the directionality of pointer assignments, whereas Steensgaard's algorithm maintains a symmetric relation. CodeSurfer's major compromise is the use of a context- and flow-insensitive points-to analysis that does not model structure fields.

Changing settings to increase Sprite's precision or decrease CodeSurfer's had varying effects on precision. Thus, the differences in precision we found can be attributed to CodeSurfer's conservative handling of unconditional control flow, use of infinite context depth, use of Andersen's superior points-to analysis, and its generally better library models.

A closer look at the data behind several slices revealed that points-to relations for several of the programs were large and implausible for both CodeSurfer and Sprite. A likely explanation for the largely similar slicing results once context sensitivity and CodeSurfer's libraries were used (and hence Sprite's small safety margins) is that collapsing of points-to sets due to pointer manipulations and the context insensitivity of the pointer analysis defeated the distinguishing of structure fields in Sprite's points-to analysis, the directionality of CodeSurfer's points-to analysis, and the context-sensitivity of data dependences in both. Further investigating the effects of context- and flow-sensitive pointer analyses might yield improved algorithmic precision. Recent work suggests that partial context sensitivity in pointer analysis can be achieved at low cost [10, 12].

Performance. The precision results above suggest that in addition to the performance-management techniques employed by Sprite and CodeSurfer, performance can be managed by "balancing" precision across the algorithms employed in an approach, thus avoiding the cost of algorithms whose benefits cannot be realized due to compromises elsewhere in the analysis.

CodeSurfer's precomputation of costly data structures reduces slicing costs to roughly constant on these slices. Sprite's demand-driven analysis avoids unnecessary computation at both build and slicing time. However, CodeSurfer's high-precision build costs remain high, as do Sprite's high precision slices. Both tools also provide significant tunability. Unfortunately, tunability is compromised by both the precomputation and demand-driven approaches. Tuning parameters to achieve high-precision in CodeSurfer sacrifices scalability because increased space requirements translate into increased time in builds. That is, precomputation significantly amortizes per-slice costs, but it does not insulate slicing from increases in slice time as precision is increased (and perhaps as program size is increased). Using tuning parameters to achieve high-precision in Sprite is costly because super-linear costs are pushed into slicing time. The precomputation and demand-driven approaches both have advantages, but they cannot be straightforwardly combined: changing Sprite, for example, to save all costly data structures on disk would significantly raise the build-time cost of tuning parameters to increase precision.

Non-algorithmic effects. An unexpected result is that non-algorithmic decisions tended to have as great an impact on the results as algorithmic ones, with modest impact on performance. The tools' designs differ regarding what should be highlighted (e.g., declarations), whether to slice into callers, or how to handle incomplete or erroneous programs (e.g., behavior for a missing function definition). The reasonable decision to highlight declarations often doubles the size of a slice. Using CodeSurfer's library models on several slices brings slice size down at modest cost. Better library models are not a substitute for context sensitivity, but library improvements allow increased precision at low a cost.

Additionally, for Sprite, using the worklist algorithm could improve the results to the same level as context sensitivity at a fraction of the cost. For Sprite, the use of the worklist versus iterative algorithm is an implementation issue, not an algorithmic one; the algorithm is essentially the same for both implementations. Using the default iterative algorithm yields up safety margins up to 572, while using the worklist algorithm gives results similar to maximum context sensitivity. This result does not scale to larger programsxi, however.

There is no "right" answer in handling some of these issues. The particular task and working style of the programmer influence what is the best choice. Highlighting more statements is safe in that it is a more complete enumeration of the contributors to the slice and hence to the behavior of the initial slicing criterion. Yet highlighting statements that are irrelevant to the task slows down and frustrates the programmer, perhaps causing the programmer to use the tool less.

7.2 Usability

Although both tools provide customization and query capabilities, neither adequately helped us to customize what is highlighted to fit the task, avoid common mistakes, or answer questions that we believe many programmers using a slicer would ask. Only with considerable effort could we determine why a statement or variable reference was included in a slice, even though we had an extra slicer for comparison. As a result, we often resorted to modifying the program (e.g., removing a `return` or a pointer assignment) just to see how the slice changed. If a programmer is using a slicer to guide complicated code changes, she may want to know not only *what* needs to be changed, but *how*. In our experience from this study, getting the answer using a slicer requires knowing a great deal about both the semantic characteristics of the program and how the slicer operates. A complex interaction of pointer characteristics, data dependences, control dependences, and highlighting rules can be at work. Furthermore, the possible loss of precision due to any number of factors—say, imprecision in the pointer analysis—means that the tool user must remain constantly aware of what the program's dependences actually are *and* how the slicer is modelling and displaying them. Unfortunately, the straightforward solution to increasing confidence in the trueness of dependences—increasing precision—comes at considerable cost.

7.3 Recommendations and Open Questions

Our results suggest a few issues that tool designers might take into consideration when designing a program slicer or other interactive data-flow analysis tool:

- To provide beneficial time–precision tradeoffs, an approach should use (or permit choosing) algorithms of complementary and balanced precision. For example, context-sensitivity in data dependences should be complemented by context-sensitivity in pointer modelling.

- Greater attention should be given to non-algorithmic design choices and how they affect a user's interpretation and use of a slice. Behaviors that can lead to confusion, such as handling of undefined functions and uninitialized pointers deserve special attention. Improved library modelling might provide substantial precision benefits at modest cost.

- For the purposes of understanding *why* certain statements are in a slice, support should be provided for significant customization of what dependences or portions of the program are included in the slice. For example, it should be possible to control not only the inclusion of control versus data dependences, as CodeSurfer can, but also distinguish (and/or filter) weak semantic dependences, loop-carried dependences, etc. Because many of the pointer aliases we saw were highly implausible, removal of aliases according to declared program types or declarative methods could permit exploring these properties. We would also like to see what assignment in the program brought about such an alias.

Three research questions arose during our study that could not be answered adequately with the tools in their current instantiation:

- How might a context-sensitive (and even flow-sensitive) pointer analysis improve the precision of slices?

- Do the results we report here hold on larger programs?

- How can a program slicer or similar tool be designed to hide the complex details of its algorithms from users, yet provide insight into the legitimate (versus illegitimate) causes of dependences?

8. CONCLUSION

This paper reports on the first comparative studying using two whole-program static slicers, GrammaTech's CodeSurfer and our Sprite research prototype. CodeSurfer favors precision and uses precomputation to amortize slicing time costs, whereas Sprite favors performance and uses demand-driven algorithms to reduce slicing costs. Examining the net impact of a set of complementary techniques in the design of a complete tool provides insights for the efficacy of an overall approach, rather than individual algorithms. We first ran a set of microbenchmarks to ascertain the feature sets of the two slicers and expose hidden experimental variables, and then ran a set of macrobenchmarks to determine the broader implications of each tool's approach.

For the programs and slicing criteria we chose, we found that the relationship between the cost and the precision of an approach was unpredictable. The use of precise algorithms incurred a high runtime cost, yet only sometimes provided large benefits in reducing slice size. It appears that the precision provided by context-sensitive dependence analysis and a superiority of Andersen's points-to analysis was compromised by the context-insensitive points-to analysis. Distinguishing structure fields in Steensgaard's points-to analysis in Sprite was also ineffective. Subtleties in the control-dependence analysis also increased the size of slice results, sometimes dramatically. The preliminary implication is that achieving good time–precision tradeoffs benefits from consistent precision across the chosen algorithms comprising an approach, although the results are highly dependent on the exact nature of the program and the slicing criterion.

An unexpected discovery in this study was that non-algorithmic design decisions had a significant impact on the results, often similar to the impact of algorithmic choices. For example, in trying to improve Sprite's results on some slices, using CodeSurfer's I/O library models had as much impact as adding context sensitivity. Using the Sprite worklist algorithm had as much impact as context sensivity in all other cases. However, this algorithm may have a performance impact on larger programs [3]. Given the modest cost and high impact of non-algorithmic decisions, compromises are unwarranted, but good choices require attention to the tasks to which a user may apply the tool.

Our pained attempts to understand why a particular statement appears in a slice suggest that tools like slicers need support for customizing what constitutes a dependence, post-filtering of dependences, and classifying the remaining dependences in a convenient fashion. The implausibility of many pointer aliases computed by both tools suggests that users be allowed to customize these as well. Ultimately, it would be ideal to be able to learn the "cause" of a dependence without requiring detailed knowledge of how the tool operates.

Much work remains to be done in understanding the various tradeoffs in the design of a program analysis tool such as a static program slicer. Due to the large number of variables that can influence slicer behavior and the difficulty of understanding why a particular statement is in a slice, we have only scratched the surface. With the improvements likely to come, it should soon be possible to assess the impact of tools like program slicers on program development and maintenance activities. Such contextualization will shed more light on a number of issues that we have just begun to explore here, such as control over precision and performance, customization of the slicing computation, improved query capabilities, and strategies that tradeoff batch precomputation with on-demand computation.

Acknowledgements

We are very greatful to GrammaTech, in particular Paul Anderson, Tim Teitelbaum, and Jolene O'Connor, for their quick and thoughtful help with CodeSurfer throughout this project.

REFERENCES

[1] H. Agrawal. On slicing programs with jump statements. In *Proceedings of the SIGPLAN '94 Conference on Programming Languages Design and Implementation*, pages 302–312, June 1994. SIGPLAN Notices 29(6).

[2] L. O. Andersen. *Program Analysis and Speicialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[3] D. Atkinson. Personal Communication, 2000.

[4] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, November 1998.

[5] T. Ball and S. Horwitz. Slicing program with arbitrary control flow. In *First International Workshop of Automated and Algorithmic Debugging (AADEBUG '93)*, pages 206–222. Springer–Verlag, May 1993.

[6] J. D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 1998 International Conference on Software Engineering*, April 1998.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[10] D. Liang and M. J. Harrold. Towards efficient and accurate program analysis using light-weight context recovery. In *Proceedings of the 2000 International Conference on Software Engineering*, June 2000.

[11] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, SE-16(9):965–979, 1990.

[12] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, June 2000.

[13] M. Shapiro and S. Horwitz. The effects of precision on pointer analysis. In *Fourth International Symposium on Program Analysis*, September 1997.

[14] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, FL, January 1996.

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

[16] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the SIGPLAN '99 Conference on Programming Languages Design and Implementation*, May 1999.