srcSlice: very efficient and scalable forward static slicing

Hakam W. Alomari¹, Michael L. Collard², Jonathan I. Maletic³*,†, Nouh Alhindawi⁴
and Omar Meqdadi⁵

¹Faculty of Information Technology, Jerash University, Jerash, Jordan
²Department of Computer Science, The University of Akron, Akron, Ohio, USA
³Department of Computer Science, Kent State University, Kent, Ohio, USA
⁴Faculty of Sciences and Information Technology, Jadara University, Irbid, Jordan
⁵Department of Computer Science, University of Wisconsin-Platteville, Platteville, Wisconsin, USA

ABSTRACT

A highly efficient lightweight forward static slicing approach is presented and evaluated. The approach does not compute the program/system dependence graph but instead dependence and control information is computed as needed while computing the slice on a variable. The result is a list of line numbers, dependent variables, aliases, and function calls that are part of the slice for all variables (both local and global) for the entire system. The method is implemented as a tool, called srcSlice, on top of srcML, an XML representation of source code. The approach is highly scalable and can generate the slices for all variables of the Linux kernel in approximately 20 min on a typical desktop. Benchmark results are compared with the CodeSurfer slicing tool from GrammaTech Inc., and the approach compares well with regard to accuracy of slices. Copyright © 2014 John Wiley & Sons, Ltd.

Received 22 April 2013; Revised 18 November 2013; Accepted 20 February 2014

KEY WORDS: program slicing; software maintenance; impact analysis; forward decomposition slice

1. INTRODUCTION

Program slicing is a widely used, and well-known, approach for understanding and detecting the impact of changes to software. The idea is simple, given a variable and the location of that variable in a program, a slice produces the other parts of the program that are affected by the variable. Slicing has been used successfully for many years for a wide variety of maintenance tasks [1–12]. For example, slicing was used to help address the Y2K problem [13, 14] in identifying parts of a program that could be impacted by changes on date fields. The concept of program slicing was originally identified by Weiser [15, 16] as a debugging aid. He defined the slice as an executable program that preserved the behavior of the original program. Weiser’s algorithm traces the data and control dependencies by solving data-flow equations for determining the direct and indirect relevant variables and statements. Since that time, a large variety of different slicing techniques and tools have been proposed. This large body of literature is well documented in a detailed survey on the vocabulary of program slicing [17]. These techniques can be broadly distinguished according to the type of slices such as, static versus dynamic [7, 11, 12, 18, 19], closure versus executable [12], inter-procedural versus intra-procedural [20–22], and forward versus backward [23–26, 12].

The calculation of a program slice is, with few exceptions, based on the notion of a Program Dependence Graph (PDG) [27, 28] or one of its variants, for example, a System Dependence Graph...

*Correspondence to: Jonathan I. Maletic, Department of Computer Science, Kent State University, Kent, Ohio, USA.
†E-mail: jmaletic@kent.edu

Copyright © 2014 John Wiley & Sons, Ltd.
Unfortunately, building the PDG/SDG is quite costly in terms of computational time and space. As such, slicing approaches generally do not scale well, and while there are some (costly) workarounds, generating slices for a very large system can often take days of computing time. Additionally, many tools are strictly limited to an upper bound on the size of the program they can slice because of memory constraints.

Our approach [30] addresses this limitation by eliminating the time and effort needed to build the entire PDG. In short, it combines a text-based approach, similar to Cordy’s [31], with a lightweight static analysis infrastructure that only computes dependence information as needed (aka on-the-fly) while computing the slice for each variable in the program. The slicing process is performed using the srcML [32, 33] format for source code. srcML provides direct access to abstract syntactic information to support static analysis. While this lightweight approach will typically never match the accuracy of generating a complete PDG/SDG and doing full pointer analysis, it can provide a fairly accurate picture of a program slice in an extremely short time comparatively for large systems (i.e., we found a three orders of magnitude increase in speed for large systems).

A very fast and scalable, yet slightly less accurate, slicing approach is extremely useful for a number of reasons. Developers will have a very low cost and practical means to estimate the impact of a change within minutes versus days. This is very important for planning the implementation of new features and understanding how a change is related to other parts of the system. It will also provide an inexpensive test to determine if a full, more expensive, analysis of the system is warranted. Lastly, we feel a fast slicing approach will open up new avenues of research in metrics and the mining of histories based on slicing. That is, slicing can now be conducted on very large systems and on entire version histories in very practical time frames. This opens the door to a number of experiments and empirical investigations previously too costly to undertake.

We implemented our approach in a tool called srcSlice.‡ The approach was first introduced in [30], and there we conducted a comparison study to the CodeSurfer tool from GrammaTech Inc.§ A free academic version of CodeSurfer is used, as the commercial version was prohibitively expensive for this study. In the work presented here, we extend this evaluation to a total of 18 systems, along with making a number of enhancements to our algorithm and tool. The results of the comparison show that the slices produced by our approach are very reasonable with respect to accuracy. It is also shown to be very efficient with regard to computational time. To further demonstrate the scalability, we applied the tool to 17 years of versions of the Linux kernel and present the results.

The specific enhancements from our previous work include an improved algorithm to greatly enhance intra-procedural issues and pointer aliasing. The approach has also been extended to produce control-flow path information for each slice as that was not available previously. A new feature to extract line slices (versus variable slices) has also been added to the tool. The complete algorithm for intra-procedural slicing is now presented, and an algorithm for computing the control-flow information is included. The tool now supports computing slices on both C and C++. The evaluation has been extended and includes details of the feature benchmark results. Also, additional systems were added to the performance benchmarks, and times for different settings were added for a more complete comparison.

The remainder of the paper is organized as follows. Section 2 describes our slicing approach, the specifics of the algorithm, implementation details, and some limitations of the approach. A comparison with the CodeSurfer tool is presented in Section 3, and the study results are presented in Section 4. Section 5 demonstrates the scalability of the approach. Related work is then presented in Section 6, followed by conclusions in Section 7.

2. LIGHTWEIGHT FORWARD STATIC SLICING

Forward static program slicing [21, 25, 34] refers to the computation of program points that are affected by other program points. The forward slice from program point $p$ includes all the program

‡Available for download at www.srcML.org under General Public License.
points in the forward control flow affected by the computation at \( p \). Here, we use the initial variable declaration as the starting point. The approach varies from the traditional definitions in two ways. First, a PDG is not computed for the entire program. Second, the slicing criterion does not require a precise reference to a location in the source (only a variable). Specifically, the approach taken here computes a \textit{forward, static, non-executable (closure), inter-procedural program slice} for each variable in a system.

The conventional definition of a forward slice leads to slices that are not executable [21, 25]. The forward slice is defined as the set of statements that may be affected by the slicing criterion. However, Binkley et al. [35] state a formal definition of forward slicing that is faithful to the spirit of forward slicing and for which the constructed slices are executable. They start by establishing a formal link between a forward non-executable slice and what they call the complement executable forward slice. They prove that the formal definition of the complement forward executable slice is precisely the complement of the traditional non-executable forward slice as defined by [25]. That is, instead of defining a forward slice as those statements of the program that are affected by the slicing criterion, a complement forward slice is defined as a program whose execution is unaffected by the slicing criterion. Finally, they expressed the symmetry between forward and backward slicing as follows: `A complement (executable) forward slice is the largest sub-program unaffected by the initial values of variables in the slicing criterion, while the backward slice is the smallest sub-program affected by the final values of variables in the slicing criterion’.

Our approach follows the original definition of the forward slice as defined by [21]. Unfortunately, this definition of a forward slice leads to slices that are not executable. However, it is possible to define an executable form of forward slicing using the same approach followed by [35], and consequently, this is one way to extend our approach to computing backward slices. The empirical determination of how significant this difference is with respect to backward slicing remains a problem for future work, as the non-executable slices produced by our approach are beyond the scope of the theory introduced previously, while the executable slicing definitions can be investigated formally.

The approach relies on an underlying XML representation of the source code, namely, srcML [32, 33]. srcML augments source code with abstract syntactic information. This syntactic information is used to identify program dependencies as needed when computing the slice. srcML (SouRce-Code Markup Language) is an XML format used to augment source code with syntactic information from the AST to add explicit structure to program source code. The srcML format is supported with a toolkit, including src2srcml and srcml2src, which supports conversion between source code and the format. Multiple languages, including C, C++, and Java, are supported. Once in the srcML format standard XML tools can be used for analysis. This format has been previously used for lightweight fact extraction [32, 33], source-code transformation [32], and pattern matching of complex code [36]. Before presenting the static forward algorithms, we define our slicing criterion, and then show how a slice is computed using this approach.

2.1. \textit{Extended decomposition slicing criterion}

We define our slicing criterion to consist of a file name, a function name, and a variable name. This slicing criterion is the triple \((f, m, v)\) where \( f \) is a file in the system, \( m \) is a function/method in the file \( f \), and \( v \) is a variable in the given function \( m \). This definition of a slicing criterion does not require a precise reference to a statement number. This concept of slicing is used by Gallagher et al. [6, 37] and is referred to as a \textit{decomposition slice}. The definition includes all relevant computations involving a given slicing variable.

A decomposition slice can be viewed as a union of a collection of slices taken at individual statements on the given variable [22]. The example used by Gallagher does static backward slicing only. As defined by Gallagher, a decomposition slice with respect to a variable \( v \) is the union of static backward slices taken at a set of statements that output variable \( v \) in addition to the slice taken at the last statement in the program. The last statement is included so that a variable that is not part of the program output may still be used as a decomposition criterion.

Here, we extend Gallagher’s definition of decomposition slicing in two ways. The first is adapting it to forward static slicing and second is to slice at the set of statements that define a
given variable \( v \). This permits a maintainer to analyze all effects of proposed changes to a particular variable and to isolate the effect of those proposed changes. This choice is motivated by the example given in Figure 1. If we perform two forward slices of the variable \( c \) in this program starting at statements \( s_3 \) and \( s_5 \), that is, both statements that assign (redefine) a value to the variable \( c \), then the resulting slices include the statements \( \{ s_3, s_4 \} \) and \( \{ s_5, s_6 \} \), respectively, that is, statements impacted by the value of variable \( c \). Slicing from the assignment statement in \( s_3 \) is not sufficient to capture all the statements impacted by the value of \( c \), given that statement \( s_6 \) is not retrieved in the slice, because the value of \( c \) assigned in statement \( s_3 \) can never reach the use of \( c \) in statement \( s_6 \), as there is an assignment in statement \( s_5 \) that redefines \( c \). Therefore, the decomposition slice obtained by a forward slicing algorithm for the example in Figure 1 using the variable \( c \) is equal to slice \( (c, s_3) \cup slice (c, s_5) \).

From the perspective of data-flow analysis, the decomposition slice could be either backward-based or forward-based. That is, the backward-based decomposition slice is computed iteratively by propagating information from the outputs of variables to their inputs, and from inputs to outputs in the case of forward-based decomposition slice. Once a slice is obtained using any slicing algorithm, a decomposition slice may be computed [6, 37].

### 2.2. Slice profile and system dictionary construction

In the computation of a slice, certain dependence information is required. Unlike other slicing techniques, our algorithm does not fully rely on pre-computed data and control dependencies because they can require costly analysis, for example, constructing the def-use chains in the existence of pointers. Instead, needed dependencies for the slicing variable are calculated on the fly while constructing the slice. The approach computes a slice profile that contains all the relevant statements from all possible slices, over a given slicing variable \( v \). After the algorithm is applied, the slice profile associated with a variable \( v \) consists of the lines of code transitive affected by the value of \( v \) along control and data dependencies.

Our approach allows decomposition slices to be constructed with respect to a set of slicing criteria. Rather than just a single variable of interest within the original program, our definition can retrieve the slices for all the variables inside a given function by modifying the slicing criterion to \((f, m)\). Moreover, the slicing criterion \((f)\) can be used to find all the slices of all variables in all functions in a given file. A system dictionary is built, referred to as \((F, M, V)\), and includes all files in the system, all functions in each file, all variables in each function, and all global variables in the system. Each entry of the system dictionary is a slice profile with the following structure:

- file, function, and variable names;
- @index, an index of each variable in the order that it was declared in the function;
- slines, a list of lines that comprise the slice;
- cfunctions, a list of functions called using the slicing variable;
- dvariables, a list of variables that are data dependent on the slice variable;
- pointers, a list of aliases of the slicing variable; and
- controledges, a list of all possible control-flow edges of the slicing variable.

Let us now look at a simple example. The approach works much like a programmer would compute a slice in their head. Figure 2 presents a small program (a) along with the final system dictionary (b). The dictionary includes two slice profiles, one for each of the variables sum and i. The @index represents the position of variables as declared in the function. In this way, we can deal with variables of the same name within the same scope. The slice profiles are computed by examining

```plaintext
1. cin >> a;
2. cin >> b;
3. c = a + b;
4. cout << c;
5. c = a - b;
6. cout << c;
```

Figure 1. Slicing motivation proposed by Gallagher [6].
Function

A forward decomposition slice $ds$ of a program $p$ is constructed with respect to a given file $f$, a given function $m$ in $f$, and a given variable $v$ in $m$. It consists of the union of the static forward slices (denoted by $sfs$) constructed for the criteria $\{(v), s_1\}, \ldots, \{(v), s_k\}$, where $\{s_1, \ldots, s_k\}$ is the set of statements in $p$ that assign to $v$. It is defined as

$$ds(f, m, v) = \bigcup_{s \in \{s_1, \ldots, s_k\}} sfs_{s(v)}(p).$$

This definition can be generalized to cater to a set of variables, functions, and files. This yields a definition for the general forward decomposition slice.

Definition 2: General Forward Decomposition Slice

A general forward decomposition slice of a program $p$ is constructed with respect to the following slicing criteria $(f, m), (f, M, V)$, where $F = \{f_1, f_2, \ldots, f_j\}$ is the finite set of files in $p$, $M = \{m_1, m_2, \ldots, m_k\}$ is the finite set of methods for each $f \in F$, and $V = \{v_1, v_2, \ldots, v_d\}$ is the finite set of variables for each $m \in M$. The general decomposition slice for all variables (i.e., set $V$) inside a given function $m$ is formed by

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>main()</code> {</td>
</tr>
<tr>
<td>2.</td>
<td><code>int sum = 0;</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>int i = 1;</code></td>
</tr>
<tr>
<td>4.</td>
<td><code>while (i&lt;=10) {</code></td>
</tr>
<tr>
<td>5.</td>
<td><code>sum = sum + i;</code></td>
</tr>
<tr>
<td>6.</td>
<td><code>i++;</code></td>
</tr>
<tr>
<td>7.</td>
<td><code>}</code></td>
</tr>
<tr>
<td>8.</td>
<td><code>cout&lt;&lt;sum;</code></td>
</tr>
<tr>
<td>9.</td>
<td><code>cout&lt;&lt;i;</code></td>
</tr>
<tr>
<td>10.</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

Figure 2. (a) Sample source code, (b) system dictionary with two slice profiles for the source code in (a). The final slice for $sum = \{2, 5, 8\}$ and the final slice for $i = \{3, 4, 5, 6, 8, 9\}$ after considering dependencies.
The general decomposition slice for all variables in a given file $f$ is given by

$$fds(f) = \bigcup_{i=1}^{y} mds(f, m_i).$$

The general decomposition slice for all variables in all the files $F$, and all global variables in the system is given by

$$gds(F, M, V) = \bigcup_{i=1}^{j} fds(f_i).$$

According to Definition 1, the decomposition slice can be viewed as a special case of simultaneous slicing [38, 3] in which the set of slicing criteria are derived from the program and the variable of interest. Alternatively, decomposition slicing can be viewed as a variation of the original definition proposed by Weiser. Whereas Weiser’s slicing criterion contains a set of variables and a single slicing point, the decomposition slice contains a set of points of interest (i.e., statements calculated from the program to be sliced) and a single variable. This definition suggests the possibility of slicing at an arbitrary number of points within the program, not only at a single point. Therefore, the decision of slicing using a single point can be generalized to simultaneous slicing.

2.4. Algorithm overview

The slice profiles for all variables are computed line by line as variables are encountered. After all the slice profiles are computed, then a single pass through this system dictionary is used to take into account dependent variables, function calls, control-flow edges, and pointer aliasing to generate the final slices.

The inter-procedural and intra-procedural dependencies are defined as follows. An intra-procedural data-dependence relation between two points exists if the first point may assign a value to a variable that may be used by the second point. An intra-procedural control-dependence relation between two points exists if the first point is a conditional predicate, and the execution at the second point directly depends on the result of the first point. In addition, there is an inter-procedural data-dependence relation between each function call argument and the corresponding parameter. Finally, there is an inter-procedural control-dependence relation from each call point of a function to its signature.

To extract the direct data-dependence relations between statements, we used the standard definition of def-use chains, except that the forward redefinition of the variable is allowed. For example from Figure 1, the returned slice using the criterion $c, s_j$ includes the statements $\{s_5, s_4, s_6\}$. If we allow the redefinition of variable $c$ in statement $s_5$, this is the decomposition slice of variable $c$. Let us assume that we are interested in the slice for variable $v$. We start with the first definition of variable $v$ in function/method $m$. Then all the expression statements where the slicing variable $v$ is referenced are recorded including assignments, function calls, and pointer aliases. The statements that reference pointer aliases are recorded as they are impacted indirectly by the slicing variable.

The algorithm computes direct data dependencies in two steps: definition detection and use verification. The output of definition detection is a set of pairs of the form $(v, Sp(v))$ where $v$ is the slicing variable and $Sp(v)$ is $v$’s slice profile that initially includes the statement that defines $v$. A new declaration statement for a variable, with the same name of the existing variable (e.g., due to scope), results in a new slice profile. Use verification ensures that there is a def-use chain from the declaration statement in $Sp(v)$ to other statements in the forward trace through which a definition of variable $v$ reaches. As a result, these use statements are included in $Sp(v)$.

A failure to find a def-use chain will result in an empty slice profile. To compute the closure over the data dependencies, all statements that include local or global variables affected by the value of the slicing variable are included. For example, the slice of an assignment statement $(a = c)$ with respect
to a variable $c$ will include the slice profile of variable $a$. Detecting such statements is important because the static slicing necessitates following the slicing variable over all its possible values.

In order to locate all statements relevant to the slicing variable $v$ across the boundary of the function $m$, we consider the following. Each called function in the set $cfunctions$ is mapped to its function signature, that is, the inter-procedural control-dependence relation between the call point of a function and its entry. All arguments in these function calls are mapped to the corresponding parameters in the function signature, that is, the inter-procedural data-dependence relation between each function call argument and the corresponding parameter.

Our algorithm for computation of a forward decomposition slice is presented in Figure 3. The algorithm traces the program forward statement by statement to determine data and control dependencies. The algorithm $ComputeSliceProfile$, shown in Figure 6, is the main algorithm for intra-procedural slicing. The $ComputeInterprocedural$ algorithm, shown in Figure 6, is used when slicing over the called functions from the slice profile of the slicing variable. Additionally, the control-flow edges for the slicing variable are computed by the $ComputeControlPaths$ algorithm that is presented in Figure 7.

The $ComputeSliceProfile$ algorithm performs forward propagation of variables whose definitions are being detected. Statements and parts of statements are evaluated in the order in which they occur. This algorithm implements the definition detection by analyzing the declaration statements and parameter statements (see line 3), and implements the use verification by analyzing expression statements (see line 11).

The global declaration statements are analyzed in the same way as the definition detection in the $ComputeSliceProfile$ algorithm. The $ComputeSliceProfile$ algorithm is repeatedly called for each function in file $f$ to compute the closure over data dependencies. The definition detection generates a set of variables. The immediate data dependencies corresponding to these variables are checked by the use verification, and the dependencies are included in the appropriate slice profiles. From the newly added statements, new sets of dependent variables are generated for the closure, and the aforementioned steps are repeated until no more statements are added to the slice.

The set $V$ ($VL$ or $VG$) is responsible for storing the slice profiles of the variables. The elements in the set are $(v, Sp(v))$ pairs. Defined variables are added to the set as they are encountered. For a variable used as an l-value, a slice profile is created (if not already present) and the statement line number is added. This is carried out while processing declaration statements. When a variable is used as an r-value in an assignment statement, the l-value variable of the assignment statement is added to the set $dvariables$ of the slice profile of the r-value variable (see lines 15 and 16). The set $cfunctions$ for the variable is computed while processing function calls (see line 33), making it possible to compute the closure across the system. Declared pointers and their associated variables are added to the set $VP$ as they captured (see line 24). The set $LP$ stores the boundaries of all loops in the system, where the loop boundaries are the conditional and the end statement of that loop (see line 45).

Intra-procedural control dependencies are computed as follows. Given a statement $stmt$, that has just been included in the slice profile of variable $v$, an immediate control predicate of $stmt$, say $stmt_j$, must be included in the slice profile of variable $v$. The main control predicates of interest are while, for, if, else, switch, case, and do. The return statements are not considered, as our algorithm captures the analogous effects of a return statement appearing before the function exit through slicing over all variables. By storing those control-flow statements (loop or condition) when $stmt_i$ is included, we check to see whether it is in the body of the block of a control-flow statement. In this case, it is added to the appropriate slice profile.

Finally, using the system dictionary and the computed slice profiles, the closure can be found with a single pass through the dictionary. For each variable, the sets $dvariables$, $cfunctions$, and $pointers$ contain the relevant line numbers for the forward slice of the variable. The complete slice for each variable is computed by taking the union of a variable’s $slines$ along with the slices of all the $dvariables$, $cfunctions$, and $pointers$, minus any lines before the definition of the slicing variable. This is carried out recursively so that a complete closure is computed across procedures and variables. This is a simple process, and a hash table makes it very efficient.

The $ComputeInterprocedural$ algorithm, given in Figure 6, performs the inter-procedural slicing. However, the first step of this process is accomplished by first mapping the indices of the variables in the argument list (see line 6 in Figure 3) to their corresponding indices detected in the calling statements (see...
Input: Slicing Criterion \( \{F, M, V\} \)
Output: System Dictionary - a table of slice profiles for all variables

**Sp** (Slice Profile) \( \forall v \in V \)

**VL** (Local variables) \( \{v_i, Sp(v_i)\} \mid \forall m \in M \)

**VG** (Global variables) \( \{v_i, Sp(v_i)\} \mid \forall f \in F \)

**VP** (Declared pointers) \( \{v_i, v_j\} \mid v_i \text{ is a pointer alias of } v_j, \forall m \in M \)

**LP** (Loop profile) \( \{\{s, e\} \mid s \text{ is starting line and } e \text{ is ending line, } \forall m \in M \}

**LS** (Stack of line numbers) for loop profiles

For each file \( f, \in F \), ComputeSliceProfile(\( m_f \)) for all methods \( m_f \in F \).

**Algorithm ComputeSliceProfile(\( m_f \))**

1. repeat
2. for each stmt, \( m \) in statement sequence do
3. case: stmt is a declaration-statement or a parameter-list
4. for each \( v \in \) stmt do
5. if \( v \in V_L \) then
6. \( Sp(v).\text{index} = \{\text{index of } v\} \)
7. end if
8. \( Sp(v).\text{slines} = Sp(v).\text{slines} \cup \{\text{stmt}, \text{linenumber}\} \)
9. \( V_L = (v, Sp(v)) \)
10. end for
11. case: stmt is an expression-statement
12. found := false
13. for each \( v \in \) stmt, \( v \in V_L \) do
14. found := true
15. if \( v \) is a r-value and \( v(1-value) \neq v(l-value) \) then
16. \( Sp(v).dvariables = \{v(l-value)\} \)
17. \( Sp(v).slines = Sp(v).slines \cup \{\text{stmt}, \text{linenumber}\} \)
18. if \( v \) \( \notin V_P \) then // retrieve "a", where \( v \) is a pointer to "a"
19. \( a = \text{Vp variable}(v) \)
20. \( Sp(a).slines = Sp(a).slines \cup \{\text{stmt}, \text{linenumber}\} \)
21. \( Sp(a).dvariables = Sp(a).dvariables \cup \{v(1-value)\} \)
22. \( V_L = (a, Sp(a)) \)
23. end if
24. else if \( v(1-value) \) is a pointer alias of \( v(r-value) \) then
25. \( Sp(v).pointers = \{v(1-value)\} \)
26. \( Sp(v).slines = Sp(v).slines \cup \{\text{stmt}, \text{linenumber}\} \)
27. \( V_P = \{v(1-value), v\} \), //update declared pointers
28. end if
29. \( V_L = (v, Sp(v)) \)
30. end for
31. if !found then // For a global variable, repeat lines 13 - 30 using the set \( V_G \) instead of \( V_L \)
32. end if
33. case: stmt is a function-call //Process all arguments, with their indices
34. for each \( v \) \in \) argument-list do
35. if \( v \in V_L \) then //local variables
36. \( Sp(v).slines = Sp(v).slines \cup \{\text{stmt}, \text{linenumber}\} \)
37. \( Sp(v).cfunctions = Sp(v).cfunctions \cup \{\text{name}\} \)
38. \( Sp(v).\text{index} = \{\text{index of } v\} \)
39. \( V_L = (v, Sp(v)) \)
40. else if \( v \in V_G \) then //global variables
41. // Repeat lines 38 - 41 using the set \( V_G \)
42. end if
43. if \( v \in V_P \) then //pointer variables
44. // Repeat lines 25 to 27
45. end if
46. end if
47. case: stmt is a control-flow-statement
48. \( L_S = \text{push}(\text{stmt}, \text{linenumber}) \) //Store the line of the condition
49. case: stmt is an end-statement of loop or condition then
50. end stmt.\text{linenumber} = \( L_S.\text{pop}() \)
51. end if
52. end for
53. until done with \( m \)

Figure 3. Algorithm to compute the intra-procedural slice for each variable in a system.
mapping is given in Figure 4. Here, regarding the function call in line 5 in Figure 4, the index mapping detects that the slice profile of parameter \( x \) must be included in the slice profile of variable \( \text{sum} \), and the slice profile for parameter \( y \) must be included in the slice profile of variable \( i \) (Figure 4).

Our inter-procedural algorithm is implemented recursively (see line 31 in Figure 6). Because it is possible that some variables may be visited several times, we must avoid re-computing the slice of

```
(a) int main() {
  int sum = 0;
  int i = 1;
  while (i <= 10) {
    foo(sum, i);
  }
  cout << sum;
}

(b) void foo(int x, int &y) {
  fun(x);
  y++;
}

void fun(int z) {
  z++;
}
```

---

Figure 4. (a) Sample source code, (b) system dictionary with two slice profiles for the source code in (a). The final slice for \( \text{sum} = \{2, 5, 7, 9, 10, 13, 14\} \) and the final slice for \( i = \{3, 4, 5, 9, 11\} \) after considering inter-procedural slicing.

---

Figure 5. Control-flow path computation for the code in Figure 2. (a) Control-flow edges (b) system dictionary with slice profiles including control-flow edges shown in (a).
those variables. To accomplish this, we assign the flag visited (initialized to false) in the slicing profile of each variable. Once a variable is visited from any iteration, this flag is set to true (see lines 13 and 35 in Figure 6). Afterwards, it is never touched again (see lines 3 and 23 in Figure 6). In essence, our inter-procedural algorithm is a depth-first search algorithm.

2.5. Computing control-flow paths

Because a number of program maintenance and understanding tasks need control-flow information along with the slice, we have additional functionality to produce this information. In addition, the

Figure 6. Algorithm to compute the inter-procedural slice for each variable in a system.

algorithm ComputeInterprocedural(f)
1. for each method m_i ∈ F do
2. for each v ∈ VL do
3. if (Sp(v).visited = false) then
4. for each called function e Sp(v).cfunctions do
5. function_name ← (called function name)
6. argument_index ← {index of v in call argument list}
7. Spi ← ArgumentProfile(function_name, argument_index)
8. Sp(v).slines ← Sp(v).slines ∪ Spi.slines
10. Sp(v).pointers ← Sp(v).pointers ∪ Spi.pointers
12. end for
13. Sp(v).visited = true
14. VL ← (v, Sp(v))
15. end if
16. end for
17. end for
18. for each v ∈ VG do
19. //For global variables, repeat lines 3 - 15 using the VG instead of VL
20. end for

algorithm ArgumentProfile(function_name, parameter_index)
21. for each v ∈ local variables of function_name do
22. if (Sp(v).w_index = parameter_index) then
23. if (Sp(v).visited = true) then
24. Spi ← Sp(v)
25. return Spi
26. else
27. for each called function e Sp(v).cfunctions do
28. new_function_name ← (called function name)
29. new_parameter_index ← {index of v in call argument list}
30. if (new_function_name ≠ function_name) then
31. Spi ← ArgumentProfile(new_function_name, new_parameter_index)
32. //repeat lines 8 - 11
33. end if
34. end for
35. Sp(v).visited = true
36. VL ← (v, Sp(v))
37. end if
38. end if
39. end for
40. return Spi
use of this information facilitates handling of unstructured control flow. If a PDG is generated, this information is directly available from the PDG. However, in our approach, control-flow information is not stored when producing a slice. Therefore, in order to include control-flow information as an output, we need to store this information while the slice is being computed. This additional step incurs computational overhead, and as such, we do not derive this information by default.

For a structured program, control-flow paths are determined by branching (e.g., loops and conditionals). Consider the effects of the statement at line 4 in Figure 2. This statement causes two-flow branches regarding the variable sum, namely, the true branch (from statement 2 to statement 5) and the false branch (from statement 2 to statement 8). Thus, we have two control-flow edges: (2, 5) and (2, 8). Figure 5 presents all control-flow edges for this problem.

We added the capability to generate control-flow paths by extending ComputeSliceProfile to keep track of sequence information between lines. That is, while computing the slice, we save the beginning and ending of each loop and condition (i.e., the pre and post dominators). This is carried out in the last two cases of the algorithm (lines 45–49) in Figure 3. With this information, we then apply the ComputeControlPaths algorithm given in Figure 7 to generate the control-flow path for the slice.

Let us consider each control-flow statement as a conditional statement that has one predecessor statement, one true-successor statement, and one false-successor statement. The predecessor statement is defined as the last statement executed before entering the control block. The true-successor statement represents the first statement that will be executed when the control-flow statement evaluates to true. Finally, the false successor is the first statement that will be executed when the control-flow statement evaluates to false. Note that a successor statement for both of these cases may not exist. Because our flow analysis is at the variable level, the control-flow edges differ

---

**Figure 7. Algorithm to compute the control-flow edges for each variable in a system.**

---

from one variable to another (e.g., values are extracted from the slicing lines of each variable separately). Because the control-flow statement is a two-way branch, we define two control-flow edges for each block: (predecessor, true successor) and (predecessor, false successor), see lines 15 and 16 in Figure 7.

To complete the computation of the paths from the control-flow edges, we create an edge between each two successive slicing lines (e.g., statements in the body of the same block). Special care has to be taken with statements that belong to an if-block and its associated else-block. In particular, if a slicing line is in a body of an if-block and the successor line is in an else-block, then no edge is created between these two successive lines (see lines 21 and 22 in Figure 7). Additionally, we connect each slicing line not belonging to any loop or condition with the first successor outer line (see lines 24–26 in Figure 7). This is to override the case of multi-successive loops or conditions, where there is a path that results from the false branch of all these successive loops or conditions.

The presence of goto statements, and its restricted forms (e.g., break and continue), complicated the construction of the control-flow information. This problem is considered [39] as one of the more demanding and critical problems when slicing is applied to maintenance problems over legacy systems, often written in old, unstructured programming styles. The problem is determining which predicates and relevant jump statements to include in the slice when the program contains unconditional jump statements. Clearly, goto statements need to be included in order for the slice to have the correct semantics.

We treat goto statements as any other control-flow statement (line 45 in Figure 3) and include the path to the label. If it happens to be guarded by a conditional, then both paths are included. That is, we considered that the true successor represents the first statement that will be executed when the jump statement is taken and the false successor is the first statement that will be executed when the jump statement is not taken. Representing a jump statement this way causes it to be the source of control-flow path.

2.6. Pointer analysis

We support limited pointer analysis by keeping track of aliases in the slice profile. Each variable that we slice \( v \) on may have a set of aliases \( VP \) (lines 25–27 in Figure 3) that are stored in the variable’s slice profile. We then include the slices of the aliased variables in \( VP \) into the slice of the variable \( v \) when doing the final computation of the slice.

Currently, we support direct pointer aliases such as pointer being assigned to a variable’s address (int *ptr; ptr = &x); that is, we keep track of local pointer variables. Additionally, we support reference variables in C++ so call-by-reference situations are supported as aliases. The fact that we keep a set of aliases allows support for some cases of indirection.

2.7. Implementation

We implemented our slicing approach as a tool called \textit{srcSlice}. It is written in C++ and uses srcML as input. The system to be sliced is first converted to srcML. The tool \textit{src2srcml} is applied to the source-code files of the system and produces a srcML archive, a single file that contains the entire system in the srcML format. The source code is unprocessed, that is, it is not run through the C-preprocessor before being converted. The conversion to the srcML format is very fast at a typical rate of 28 KLOC/s, and produces a reasonably sized XML file of three to four times that of the original source code.

The \textit{srcSlice} tool is then run on the srcML archive using the Qt XML parser QXmlStreamReader. The srcML format preserves all original source-code text, and can easily be used for an identity transformation. This pull XML processor produces the srcML one node at a time in document order, that is, as they are listed in the source-code files. Because srcML wraps statements, declarations, expressions, and syntactic constructs with XML elements, analysis programs (including \textit{srcSlice}) can process each of these program elements as they reached, with an exact mapping back to the original source code. For \textit{srcSlice}, the iteration through the nodes is used to record the necessary data for our slicing approach. The advantages of using the pull approach are that traversing the code
is quite straightforward and memory conservative because there is no need to store the entire srcML document tree in memory, as in a document object model (DOM) approach.

The srcSlice tool generates the following:

- Slice profile: contains all the relevant inter- and intra-procedural statements, and control-flow paths from all possible slices, over a given slicing variable \( v \).
- System dictionary: includes all files in the system, with all functions in each file, all local variables in each function, and all the global variables in the system.

Change sets can be expressed as a set of lines. In considering the impact of a change set, it is useful to see the computation of a slice based on this set of lines. Therefore, we extended our tool with line slicing. This feature allows developers to compute the forward slice of a given line in a given file, and will be very useful for computing the impact of maintenance activities committed to a version-control system.

That is, instead of computing the forward slice of all variables in all the files, the slicing over the line granularity will only compute the slice profile for the variable(s) of interest, which occur (defined/declared) in a specific statement. This removes local lines and any intra-procedural lines that are not of interest. For example, if we are interested in the forward slice of line 6 in Figure 2, then we will retrieve only the slicing lines of variable \( i \).

To accomplish line slicing granularity, our approach looks up the specified file and finds the variables appearing (directly or indirectly using aliases) in the given line(s). Then, we traverse and filter out the slice profiles for those variables from the pre-computed slice dictionary. This is basically a simple traversal process because there is no need to re-compute the slice profile again. All the information necessary is already in the system dictionary.

Two options are provided for slicing lines. The first option prunes all lines prior to the specified line, in the given functional scope of the line, from the slice profiles of the variable(s). The second option does no such pruning. For example, line slicing of line 6 in Figure 2, with these two options is

- Without pruning: \( \text{Slice Profile}(i) = @index(2), \text{slines} = \{3,4,5,6,8,9\}, \text{dvars} = \{\text{sum}\} \)
- With pruning: \( \text{Slice Profile}(i) = @index(2), \text{slines} = \{6,9\} \)

This allows the user different options as to how many constraints to put on the slicing according to their needs.

We now analyze the time requirements of this algorithm. The system dictionary can be constructed in time \( O(cn) \), where \( n \) is the number of statements in the program, and \( c \) is the average number of variables per statement. The complete closure is determined in constant time if we use a hash table that maps sets of slicing variables to other relevant lines for the forward slice of the variable.

### 2.8. Limitations of the approach

This implementation of srcSlice works for both the C and C++ languages. It supports user-defined types, classes, methods, method calls, and limited inheritance. It is also very scalable with no inherent limitations on the size of the program being sliced. However, because of the more lightweight approach taken, there are a number of limitations (for C++) to our implementation of srcSlice.

First, the tool does not make any attempts to do analysis on dynamic binding through virtual functions or function pointers. Currently, in the case of virtuals, we take the method from the class of the variable as can be determined within its local scope. As we do not do full type resolution, this often resolves to a base class. Thus, we include only one method in the slice versus all possible virtuals that could be called. Because CodeSurfer does full name/type resolution, it should be including all possible virtual methods in the slice. We feel that some limited analysis of virtuals and function pointers could be supported in the future but nothing is being carried out currently. In order to derive a more complete solution, we would need to perform additional type/inheritance resolution.

Second, although srcSlice supports function name overloading between base and derived classes, our tool does not fully or correctly support all overloading situations properly. We currently do not distinguish between overloaded methods on the same class but with different parameter/return types.
While we do slice across each version of the overloaded method, any calls to an overloaded method are assumed to be associated with the initial occurrence/declaration of the overloaded method (physically in the file). This is our lightweight solution to the problem and will cause differences in intra-procedural analysis for the slice. Again, we feel that overloading handling could be improved with additional effort.

Third, support for unstructured control flow is not completely accurate, and we do not support exceptions at this time. Our approach for dealing with goto statements is akin to what is carried out by Gallagher [40, 6] and Jiang [41], but may not produce correct slices in all situations [42, 43]. While our approach is not completely accurate, and better more complete solutions do exist [44–46], given the overall difficulty of this problem [39], we feel it is a reasonable solution in the context of our goal of efficiency.

With regard to pointer analysis, we currently do not support full type resolution and will miss some pointer variables. Also, there are many complicated pointer aliasing situations that are extremely difficult to address even with very time consuming analysis approaches. For example, the flow-sensitive and context-sensitive analysis algorithms can produce precise results, but their complexity, at least \( O(n^3) \), makes them impractical for large systems [47]. While our approach addresses the simpler pointer aliasing situations [48, 49], we do not address the more complex cases as they are beyond the scope of this tool and its goal of efficiency versus accuracy.

3. COMPARATIVE STUDY

To assess our approach and the srcSlice tool, we conducted a comparative study with the academic license of CodeSurfer. The objective of this study is twofold. First, we want to determine if the slices produced from srcSlice are comparable to those produced by CodeSurfer in terms of the correctness and the size of the slices. That is, we compare how srcSlice's algorithm affects the size and the accuracy of the slices compared with a standard. The second objective is to demonstrate that our approach is highly scalable and efficient. Together, these objectives lead to two primary research questions this study tries to address:

**RQ1:** Does srcSlice produce accurate (compared with CodeSurfer) slices?

**RQ2:** Is srcSlice highly size-scalable and time-efficient?

The question of what is a perfectly accurate slice is somewhat open to interpretation [15, 50]. This is the case for many results of static analysis. For example, an empirical study of static call graph extractors by Murphy et al. [51] demonstrates that the call graphs extracted by several broadly distributed tools vary significantly enough to surprise many experienced software engineers. These differences are shown with nine different call graph extraction tools of C code from three software engineering systems. In particular, an evaluation and comparison of five different implementations of program slicing by Hoffner [52] showed that the resulted slices differ in their size and accuracy. His study covered three inter-procedural slicing tools.

In order to evaluate our slicing approach, we compare the results obtained by our tool to the results of CodeSurfer. The same benchmarks are given to both tools. We feel that comparing our results to that of a commonly used existing approach/tool will minimally provide a baseline with respect to the accuracy of the results. If our results are similar to that of CodeSurfer’s, we feel confident that it produces reasonably correct slices. In other words, the slices produced by CodeSurfer are used as a gold standard to which the approximate results of srcSlice can be compared. We first ran both slicers on a number of small programs (i.e., feature benchmarks, from 21 to 112 lines of code (LOC)). These results are used to determine the correctness of our results and help explain the slicing results in larger programs. Second, we ran both slicers on larger open-source programs (i.e., performance benchmarks) of varying size, (from 3 to 600 KLOC) that worked with both slicers.

3.1. Set-up and configurations

*CodeSurfer* is a commercial-based slicing tool for C/C++ programs. Produced originally as a research tool, it is now available from GrammaTech Inc. It is based on the slicing work carried out at the
University of Wisconsin surveyed in [53]. This tool starts by generating a control-flow graph (CFG) for each source file in the system, and then the SDG is constructed for the entire system. CodeSurfer views slicing as a graph reachability problem, either backward or forward with three options for dependencies: control-dependence edges only, data-dependence edges only, or both edges. There are several features provided by CodeSurfer to assist in the code analysis process of slicing including the extraction of return values, passed by reference parameters, and modified global variables for a given function scope. CodeSurfer allows the user to control the settings of the aforementioned features with five different static analysis parameters that affect the level of precision and consequently the build time. For the Super-lite setting, all expensive analyses are disabled including pointer analysis and no data or control dependencies. The Lite setting is the same as Super-lite except that the CFG is generated. For the Medium setting, the intra-procedural data dependencies are calculated, but no inter-procedural data dependencies, and imprecise, but more efficient, pointer analysis is performed. For the High setting, the full functionality is supported with high precision, except that dynamic storage is not included in the pointer analysis [54]. The Highest setting eliminates this last limitation.

In the context of this study, CodeSurfer has two main limitations. The first limitation is that CodeSurfer does not have the ability to slice incomplete and non-compile-able source code. While this may not be a major deficiency, our approach does not require the system to be compiled (or complete). The srcML parser ignores the missing parts but still generates a valid srcML file. The second limitation is that the academic license of CodeSurfer we used in this study was unable to slice programs larger than 200 KLOC with the Highest setting. The commercial version of CodeSurfer is quite expensive and beyond our means to purchase. We found that at the lowest settings (i.e., Super-lite), it can slice programs larger than 200 KLOC, but this provides no intra-procedural slicing, pointer analysis, or control-flow information.

For our study, the Highest setting for CodeSurfer is used to provide the most precise results. Because CodeSurfer is widely used, we are more interested in comparing the accuracy rather than exact timings. We do include a comparison of times at both the highest and lowest settings for completeness. Also, because the tool can be used for other tasks than computing slices, all features except the slicing results were turned off.

The output format of the two tools is somewhat different. In order to compare the results, some filtering and scripting was necessary. CodeSurfer has an API that allows users to build extensions to the capabilities of the tool via a Scheme programming interface. We developed a script to obtain the slice size data from CodeSurfer in the following manner:

- Get a list of all program points in the PDG.
- Perform slicing using the nodes from the list.
- Calculate the number of program points in the slice.
- Output the size of the slice into a specified file.
- Repeat from steps 2 to 4 until end of list.

CodeSurfer produces a slice that consists of a list of statement numbers for each file in the system. Any duplicate lines were removed before the slice size for each file is calculated.

3.2. Evaluation criteria

Here, we want to evaluate the slicing results of our tool to determine if correct slices are produced, and are produced efficiently. The time and cost it takes to generate the slice, including execution time and memory requirements, is of particular interest with respect to usability of the approach. In addition, we want to determine if the results obtained by srcSlice are comparable to CodeSurfer in terms of accuracy. However, because the implementations of these tools have so few aspects in common, it is not meaningful to compare all of the relevant aspects of the different implementations. Therefore, we focus our attention on evaluating slices of both tools, by taking into consideration the

---

3The version of CodeSurfer was obtained from GrammaTech in May 2013. It is a free academic license version.

correctness, size of the results, time and space efficiency, and the limitations of both tools. Finally, we investigate most of the language features supported, for example, is the tool able to handle pointers, call by reference, and so on?

First, we examine the slice size and quality. The correctness of the slice relates directly to its purpose [52]. A small slice that contains relevant parts of a program for a specific input could be used for locating bugs, but it might be too small for applications where we have to consider all possible inputs (e.g., overall program comprehension) [10]. The slice size (denoted by \( SZ \)) for both tools is measured by the SLOC, number of statement lines of code. The slice is safe if it contains every statement that is actually affected by the slicing criterion. A safe slice is conservative if it may be imprecise, that is, if it also contains statements that are not affected by the value of the variable in the slicing criterion (false positive). The minimal slice is a safe slice that contains no unnecessary statements [54], that is, no other slice for the same slicing criterion contains fewer statements. The slice precision factor can be measured by how close the resulting slice is to the minimal slice [52].

The problem of determining the minimal slice is not in general decidable [15, 54, 55]. In fact, such a set is un-computable because of the un-decidability of the required static analysis. However, as mentioned earlier, the definition of what is a minimal slice depends on the intended use. Therefore, even with the most precise slicer, the resulting slice is at best a conservative approximation of the minimal slice, that is, the resulting slice \( \supseteq \) minimal slice.

A preferred decrease in the slice size is limited by the ability of the resultant slice to reflect all system behavioral aspects. Binkley et al. [56] observed that after studying 43 programs with \( \approx 1 \text{MLOC} \), the most precise program slicers had an average slice size equal to 30% of the original program. They studied five factors that may influence the slice size including the expansion of structure fields, the inclusion of calling context, the level of granularity of the slice, the presence of dead code, and finally the choice of points-to analysis.

For the purpose of comparison, we use the intersection of corresponding slices returned by both tools, called the intersected slice, following the same approach used in the qualitative study of Bent et al. [54]. That study compared a data-flow slicing approach (Sprite) [47, 57] and a PDG-slicing approach (CodeSurfer). Bent used the intersected slice as an approximation of the minimal slice. The relative safety margin (denoted by \( SM \)) of a slice (size of resultant slice divided by the size of the intersected slice) was used to provide a measure of the relative quality. Let us assume that the corresponding slices returned from both tools are correct with different contents. In that case, the differing statements are not required to be in the slice. Because the statements that are not included would be incorrect, this is a contradiction with the assumption that both slices are correct. Therefore, a smaller correct slice that does not include the differing statements must exist, that is, intersected slice. Hence, the intersected slice \( \supseteq \) minimal slice. However, as our performance benchmarks results demonstrate in the next section, we can obtain several hints that indicate an approximation of the minimal slice using the srcSlice tool. Our results indicate that the srcSlice’s slice \( = \) intersected slice fairly often, so that the srcSlice’s slice \( \supseteq \) minimal slice. In this paper, the slice size represents the total slice size (denoted \( TSZ \)), the sum of individual slice sizes for each slicing criterion. If there are \( n \) criterions (denoted by the set \( SC = \{sc_1, sc_2, \ldots, sc_n\} \)), then the total slice size is denoted by

\[
TSZ(SC) = \sum_{i=1}^{n} SZ(sc_i)
\]

The build time (denoted \( BT \)) reports the time required to build the SDG for CodeSurfer and the system dictionary for srcSlice. CodeSurfer does most of its work during the build phase where it pre-computes a large amount of data, primarily storing the SDG that contains data and control dependencies, and pointer information. Whenever CodeSurfer slices a program, it must first load that data from disk with slicing then perform any number of times. The slicing time (denoted \( ST \)) is the time it takes to handle a particular slicing request. If there are \( n \) slicing criterions (denoted by the set \( SC = \{sc_1, sc_2, \ldots, sc_n\} \)), then the total slicing time, the sum of individual slice times for each slicing criterion, is denoted by

\[
TST(SC) = \sum_{i=1}^{n} ST(sc_i)
\]
Furthermore, the total time overhead for one build for both slicers is denoted by $T(\text{SC}) = BT + TST(\text{SC})$. Previous studies on program slicing focus on individual slicers, and do not consider the build time [56, 58–60]. However, Bent et al. [54] verify that many slices using CodeSurfer take almost zero seconds once the load time is excluded, such that $ST(sc) \approx 0.00$. For comparison purpose with our tool, the build times are substantially larger than the total slicing time. Therefore, the time needed for retrieving the slice is ignored in our comparison because, as mentioned earlier, both tools do their slicing while constructing the system dictionary and SDG. We captured the build time for all of the slices using the UNIX built-in `time` command. The wall-clock time is reported because this represents the actual time a user waits for her results. The time to convert to srcML is also included. It took less than a second to generate the srcML for the feature benchmarks and close to 12 s for the largest program, Quantlib-0.9.7, in the performance benchmarks.

4. STUDY RESULTS

We now present data comparing slices from both tools. As mentioned previously, we use two benchmarks: (1) a feature benchmark to test a number of language features so the accuracy can be compared and (2) a performance benchmark so the run-time performance can be compared. These results are used to illustrate the first research question (RQ1) and partially address the second research question (RQ2). Finally, we ran srcSlice on the Linux kernel to fully address the second research question (RQ2).

For this study, both tools were run on the exact same machine (a standard desktop with 4 GB). The results and timings presented in the tables include all setup and parsing.

4.1. Feature benchmarks

First, we ran both tools on a set of small programs that exercised a representative set of language features and situations that slicing tools encounter. Besides various data and control-dependence situations, these feature benchmarks included situations such as function calls with control blocks, function calls within functions, nested function calls, the use of global and local variables, call by reference, pointer casting, and the use of external library calls. These are summarized in Table I along with statistics related to the programs and their slices. These statistics include three measures of program size: the size of each program in LOC as reported by `wc -l` utility and the size of the program as both file and function counts, in addition, slices taken, slicing time, slice size, and slice size relative to LOC. The complete source code for the benchmark programs are also posted on our website.**

The programs Information_flow, Sum, and Wc are provided with CodeSurfer as test cases. We developed the programs Pointer, Callofcall, and Testcases to assess additional critical test cases. The main language features addressed by each 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.
- **Testcases**: functions calls, local and global variables, call by reference, calling built-in/library 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 [54, 56, 61]. The purpose of these programs was to exercise the slicing behavior and for in-depth analysis.

Table I shows the results obtained by srcSlice and CodeSurfer for the feature benchmarks. The column Program lists the benchmarks used for comparison. The column Slicing Criterion contains the slicing inputs. For each program, we used our experience as programmers to select slicing

**See www.sdml.info/downloads/slice for the source code of the feature benchmark programs.
Table I. Feature benchmark results and comparison of CodeSurfer and srcSlice.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Files</th>
<th>Functions</th>
<th>Size</th>
<th>Slicing criterion</th>
<th>CodeSurfer</th>
<th>srcSlice</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Slices taken</td>
<td>Slicing time</td>
<td>Slice size</td>
</tr>
<tr>
<td>Information _flow</td>
<td>112</td>
<td>1</td>
<td>12</td>
<td>Main</td>
<td>hi</td>
<td>1</td>
<td>1.481</td>
</tr>
<tr>
<td>Sum</td>
<td>21</td>
<td>1</td>
<td>2</td>
<td>Main</td>
<td>All criterions</td>
<td>149</td>
<td>66</td>
</tr>
<tr>
<td>Wc</td>
<td>39</td>
<td>1</td>
<td>3</td>
<td>line_char_count, eof_flag, Scan_line</td>
<td>1</td>
<td>1.212</td>
<td>16</td>
</tr>
<tr>
<td>Pointer</td>
<td>36</td>
<td>1</td>
<td>5</td>
<td>Main</td>
<td>var1</td>
<td>46</td>
<td>24</td>
</tr>
<tr>
<td>Testcases</td>
<td>114</td>
<td>1</td>
<td>14</td>
<td>Main</td>
<td>All criterions</td>
<td>37</td>
<td>25</td>
</tr>
<tr>
<td>Callofcall</td>
<td>24</td>
<td>1</td>
<td>3</td>
<td>Main</td>
<td>All criterions</td>
<td>156</td>
<td>79</td>
</tr>
<tr>
<td>Total</td>
<td>346</td>
<td>6</td>
<td>39</td>
<td></td>
<td></td>
<td>444</td>
<td>15.78</td>
</tr>
<tr>
<td>Average</td>
<td>57.7</td>
<td>1</td>
<td>6.5</td>
<td></td>
<td></td>
<td>34.2</td>
<td>2.63</td>
</tr>
</tbody>
</table>

The slicing time measured in seconds and includes converting to srcML, slice size measured in number of statements. The percentage (%) columns are the slice size relative to LOC (lines of code).
criterion that we felt exposes 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 because 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 earlier. 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.* [56] reasoned that this case is due to the fact that the slice size in the SDG reports the global variables modeled as value-result parameters. Each global variable is counted twice as a node in the SDG at both the caller and procedure entry. In contrast, srcSlice ignores these variables in the returned slice. Table I 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 [52], given two correct slices, the preferable one should be the smallest as there are fewer lines to examine to address task such as debugging. However, this may not be the case for all tasks or contexts.

Manual checking of the slices produced by both tools showed that they were correct. This manual process involved inspecting these short programs and computing the slice by hand. We inspect the slices generated by both tools and determined that they were valid slices. Both tools provide the line numbers in the file that are in a given slice.

From Table I, 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 \((main, var1)\).

The slices produced using srcSlice for the programs Information_flow, Sum, Callofcall, and Wc are very similar to (i.e., a subset of) those produced by **CodeSurfer**. The difference in the results obtained by **CodeSurfer** was due to retrieving some less related statements; such as statements inside the blocks of *for* and *while* statements 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 [54]. For example, **CodeSurfer** returned all relevant statements that modify or determine control flow in the *else* part of an *if* statement whose body was not in the slice.

These small differences in the slices could be argued (either way) as being minor inaccuracies. Many software analysis tools produce minor variations due to small differences in interpretation [51] without having any real impact on the accuracy of the results. We make no claim that srcSlice produces a better slice for these benchmarks. However, it is clear that the slices produced are very close (almost the same) as those generated by **CodeSurfer** and do not appear to be missing any critical statements.

We note again that the settings used for **CodeSurfer** were to enhance accuracy and not performance for this benchmark.

### 4.2. Performance benchmarks

This initial comparison now leads us to a more comprehensive comparison. Table II shows the results of the performance benchmark along with statistics related to the programs and their slices. Again, these statistics include three measures of program size: the size of each program in LOC as reported by *wc -l* utility, and the size of the program as both file and function counts. In this table, the slices taken represent the number of forward slices over all possible criterions for each program. For **CodeSurfer**, this corresponds to slicing for each vertex in the SDG that represents executable code. In srcSlice, this number represents the number of variables in the program using the \((F, M, V)\) slicing criterion. Note that in Table II, the number of slices for both tools and the number of lines of code does not match, because in the PDG-based slicing approach, one line of code could be represented by multiple vertices [58]. In contrast, our slicing approach has variable granularity. Thus, one line of code may have several variables.
<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Files</th>
<th>Functions</th>
<th>Slices taken</th>
<th>Time highest</th>
<th>Time Super-Lite</th>
<th>Slice size</th>
<th>%</th>
<th>Slices taken</th>
<th>Time highest</th>
<th>Time Super-Lite</th>
<th>Slice size</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>ed-1.2</td>
<td>3087</td>
<td>10</td>
<td>126</td>
<td>4438</td>
<td>21</td>
<td>5</td>
<td>1782</td>
<td>57.7</td>
<td>420</td>
<td>3</td>
<td>1</td>
<td>1136</td>
<td>36.8</td>
</tr>
<tr>
<td>ed-1.6</td>
<td>3260</td>
<td>10</td>
<td>128</td>
<td>4527</td>
<td>19</td>
<td>5</td>
<td>1863</td>
<td>57.1</td>
<td>442</td>
<td>3</td>
<td>1</td>
<td>1199</td>
<td>36.8</td>
</tr>
<tr>
<td>which-2.20</td>
<td>3586</td>
<td>14</td>
<td>51</td>
<td>1429</td>
<td>15</td>
<td>9</td>
<td>736</td>
<td>20.5</td>
<td>196</td>
<td>2</td>
<td>1</td>
<td>754</td>
<td>21.0</td>
</tr>
<tr>
<td>wdiff-0.5</td>
<td>3874</td>
<td>13</td>
<td>56</td>
<td>1097</td>
<td>11</td>
<td>4</td>
<td>652</td>
<td>16.8</td>
<td>154</td>
<td>2</td>
<td>1</td>
<td>581</td>
<td>15.0</td>
</tr>
<tr>
<td>barcode-0.98</td>
<td>5205</td>
<td>18</td>
<td>74</td>
<td>4590</td>
<td>29</td>
<td>9</td>
<td>2177</td>
<td>41.8</td>
<td>392</td>
<td>3</td>
<td>1</td>
<td>1728</td>
<td>33.2</td>
</tr>
<tr>
<td>acct-6.5</td>
<td>8749</td>
<td>27</td>
<td>127</td>
<td>4983</td>
<td>47</td>
<td>11</td>
<td>2510</td>
<td>28.7</td>
<td>635</td>
<td>5</td>
<td>1</td>
<td>1767</td>
<td>20.2</td>
</tr>
<tr>
<td>enscript-1.4.0</td>
<td>18,162</td>
<td>52</td>
<td>180</td>
<td>9456</td>
<td>71</td>
<td>15</td>
<td>5916</td>
<td>32.6</td>
<td>920</td>
<td>7</td>
<td>2</td>
<td>3472</td>
<td>19.1</td>
</tr>
<tr>
<td>make-3.82</td>
<td>36,397</td>
<td>58</td>
<td>474</td>
<td>17,012</td>
<td>807</td>
<td>27</td>
<td>9446</td>
<td>26.0</td>
<td>2914</td>
<td>143</td>
<td>3</td>
<td>10,598</td>
<td>20.1</td>
</tr>
<tr>
<td>libkate 0.3.8</td>
<td>53,441</td>
<td>111</td>
<td>2210</td>
<td>28,017</td>
<td>109</td>
<td>33</td>
<td>12,846</td>
<td>24.0</td>
<td>4993</td>
<td>17</td>
<td>6</td>
<td>12,622</td>
<td>23.6</td>
</tr>
<tr>
<td>enscript-1.6.5.2</td>
<td>56,491</td>
<td>107</td>
<td>488</td>
<td>20,234</td>
<td>184</td>
<td>35</td>
<td>12,907</td>
<td>22.8</td>
<td>2580</td>
<td>90</td>
<td>5</td>
<td>10,635</td>
<td>18.8</td>
</tr>
<tr>
<td>enscript-1.6.5</td>
<td>56,494</td>
<td>107</td>
<td>488</td>
<td>20,252</td>
<td>184</td>
<td>35</td>
<td>12,913</td>
<td>22.9</td>
<td>2579</td>
<td>85</td>
<td>5</td>
<td>10,636</td>
<td>18.8</td>
</tr>
<tr>
<td>enscript-1.6.5.1</td>
<td>56,494</td>
<td>107</td>
<td>488</td>
<td>20,252</td>
<td>185</td>
<td>35</td>
<td>12,913</td>
<td>22.9</td>
<td>2579</td>
<td>85</td>
<td>5</td>
<td>10,636</td>
<td>18.8</td>
</tr>
<tr>
<td>a2ps-4.10.4</td>
<td>57,052</td>
<td>188</td>
<td>1104</td>
<td>24,493</td>
<td>393</td>
<td>116</td>
<td>14,249</td>
<td>25.0</td>
<td>3844</td>
<td>61</td>
<td>6</td>
<td>10,756</td>
<td>18.9</td>
</tr>
<tr>
<td>findutils-4.4.2</td>
<td>72,384</td>
<td>314</td>
<td>1141</td>
<td>23,641</td>
<td>215</td>
<td>94</td>
<td>13,689</td>
<td>18.9</td>
<td>5657</td>
<td>79</td>
<td>9</td>
<td>16,764</td>
<td>23.2</td>
</tr>
<tr>
<td>radius-1.0</td>
<td>82,029</td>
<td>196</td>
<td>1719</td>
<td>38,487</td>
<td>335</td>
<td>148</td>
<td>19,218</td>
<td>23.4</td>
<td>6997</td>
<td>170</td>
<td>11</td>
<td>21,905</td>
<td>26.7</td>
</tr>
<tr>
<td>dico-2.2</td>
<td>119,592</td>
<td>332</td>
<td>2504</td>
<td>52,297</td>
<td>1763</td>
<td>246</td>
<td>28,639</td>
<td>23.9</td>
<td>10,451</td>
<td>471</td>
<td>16</td>
<td>29,418</td>
<td>24.6</td>
</tr>
<tr>
<td>cvs-1.12.10</td>
<td>144,278</td>
<td>340</td>
<td>2027</td>
<td>74,328</td>
<td>286,328</td>
<td>371</td>
<td>40,869</td>
<td>28.3</td>
<td>9779</td>
<td>192</td>
<td>18</td>
<td>34,955</td>
<td>24.2</td>
</tr>
<tr>
<td>Clanlib -0.8.1</td>
<td>181,064</td>
<td>1137</td>
<td>5624</td>
<td>68,716</td>
<td>280,914</td>
<td>344</td>
<td>45,447</td>
<td>25.1</td>
<td>13,514</td>
<td>58</td>
<td>21</td>
<td>30,488</td>
<td>16.8</td>
</tr>
<tr>
<td>HippoDraw-1.21.3</td>
<td>248,592</td>
<td>1385</td>
<td>8627</td>
<td>—</td>
<td>—</td>
<td>377</td>
<td>—</td>
<td>—</td>
<td>12,400</td>
<td>66</td>
<td>29</td>
<td>25,976</td>
<td>10.5</td>
</tr>
<tr>
<td>Quantlib 0.9.7</td>
<td>599,802</td>
<td>3389</td>
<td>15,696</td>
<td>—</td>
<td>—</td>
<td>384</td>
<td>—</td>
<td>—</td>
<td>27,348</td>
<td>168</td>
<td>99</td>
<td>53,476</td>
<td>8.9</td>
</tr>
<tr>
<td>Total</td>
<td>1,810,033</td>
<td>7915</td>
<td>43,332</td>
<td>418,249</td>
<td>31,757</td>
<td>115</td>
<td>13,265</td>
<td>28.8</td>
<td>3836</td>
<td>86</td>
<td>12</td>
<td>11,055</td>
<td>22.4</td>
</tr>
</tbody>
</table>

The slicing time is measured in seconds and includes the conversion to srcML. The slice size is measured in number of statements. The percentage (%) columns are the slice size relative to LOC (lines of code). The times for CodeSurfer are given for both the Highest and Super-lite settings. At the highest setting, CodeSurfer was unable to slice programs in our study over 200 KLOC. All the CodeSurfer and srcSlice results, regarding slice taken and slice size, are from using the Highest setting because the Super-lite setting gives poor accuracy. The last two systems are not included in the totals and averages for srcSlice or CodeSurfer.
This performance study considers just over 1810 KLOC of C/C++ code from 20 open-source programs that range in size from approximately 3 to almost 600 KLOC. Table II shows the 20 programs along with the results obtained by srcSlice and CodeSurfer, where 18 programs can be sliced using the academic license of CodeSurfer at the Highest setting. All the results are for the highest settings for both tools. We also include the run times of the minimal setting for both tools.

The performance programs were chosen to cover a wide range of programming styles (e.g., acct contains different related computations; ed has a single purpose). Eight of the programs in Table II appear in Binkley’s studies [58, 56], although they may be different versions.

Each row in the table is a benchmark we used for the comparison. The slice was overall possible slicing criterions in each program. As seen in the last row of the table, the average slice size relative to LOC using both tools over all 18 programs included between 22.3% and 28.8% of the program source code. The range of the slice size coverage in the program for CodeSurfer is striking with an overall range from 16.8% for wdiff-0.5, to 57.7% for program ed-1.2. srcSlice had a narrower overall range from 8.9% for Quantlib 0.9.7 to 36.8% for ed-1.2 and ed-1.6.

Preliminary analysis does not indicate any trend relating program size and slice size using both slicers. Smaller LOC (ed-1.2 with 3087) gives high percentages (CodeSurfer = 57.7%, srcSlice = 36.8%), and the larger LOC (wdiff-0.5 with 3874) gives low percentages (CodeSurfer = 16.8%, srcSlice = 15.0%). The same thing occurs with programs ed-1.6 and which-2.2. The program size is only one of the program attributes that potentially affects slice size, as the programming style (i.e., number of methods, global variables, pointers, etc.) also affect slice size.

One way to see this is to take a look at the number of slices taken for both tools. For example, in the two programs, a2ps-4.10.4 and findutils-4.4.2, the slice size as a percentage is directly related to the number of slices taken by the CodeSurfer and srcSlice, as follows: in a2ps-4.10.4, the numbers of slices taken are 24,493 and 3844, with slice percentages equal to 25.0% and 18.9%, respectively. Moreover, in findutils-4.4.2, the numbers of slices taken are 23,641 and 5657, with 18.9% and 23.2%, respectively, even though the LOC of findutils-4.4.2 is larger. As discussed previously, the number of slices taken by CodeSurfer is related to the number of executable vertices in the SDG, for srcSlice it is related to the number of variables in the program.

In general, the slice size produced by srcSlice is smaller than the one produced by CodeSurfer; however, this is not the case in 5 out of 18 cases, that is, which-2.20, make-3.82, findutils-4.4.2, radius-1.0, and dico-2.2. The intersected slice results give us several indications as to why.

On a per-program and overall basis, srcSlice’s slicing time is very fast; the smaller programs took around four seconds (including the time to convert to srcML). This is an indication that the pre-computation strategy is successful at reducing slicing costs. The slicing times for CodeSurfer range from ~11 to ~286,000 s, with the highest settings for precision used and are on average ~370 times slower than srcSlice’s. However, CodeSurfer does produce a larger number of slices (~6 times more), which accounts for part of the slowdown. By excluding the larger programs (i.e., cvs-1.12.10, Clanlib-0.8.1), CodeSurfer’s slicing time reduces to ~3.5 times that of srcSlice.

As discussed previously, at the Super-lite setting, CodeSurfer disables all expensive analyses computation, including pointer analysis, inter-procedural slicing, and control dependencies. We include the slicing time of CodeSurfer in Table II (CodeSurfer/Time Lite) to give an idea of how the tool performs without the more complex analysis. While this drastically speeds up CodeSurfer, the performance is still slower than srcSlice. We can also turn off the intra-procedural slicing and pointer analysis in srcSlice and speed it up as seen in Table II (srcSlice/Time Super-Lite) but likewise decrease the accuracy of the slices produced. Notice that we were also able to give slicing times for CodeSurfer on the larger systems at the super-lite setting.

4.3. Slice intersection comparison

We use the intersected slice as a measure of the quality of a calculated slice. As explained in Section 3.2, we feel that intersecting our results with those of CodeSurfer will minimally give us a baseline with respect to accuracy of the results. That is, if our results are closer to that of the intersected slice, we feel confident that it produces reasonably correct slices. The slices of selected
files are generated using all possible slicing criterions with both tools, and then the intersection between corresponding slices is taken. The intersected slices are generated for two performance benchmarks from Table II: `enscript-1.6.5` and `findutils-4.4.2`. Those programs were particularly chosen to demonstrate the exceptions where the slice size differed drastically.

The results of running the slicers on all possible criterions over 13 files of the program `enscript-1.6.5` are presented in Table III. In order to provide a better estimate of file size, the third column reports the number of statement lines of code SLOC as reported by `sloc-count` utility.\footnote{See http://www.dwheeler.com/sloccount/sloccount.html}  

Focusing first on the slice sizes, it is apparent that for all slices, srcSlice’s results are consistently smaller. The average slice size for CodeSurfer and srcSlice is 69.0% and 32.4%, respectively. Upon closer examination, we observe that CodeSurfer produced a higher safety margin (SM) on all slices than those produced by srcSlice. CodeSurfer produced a maximum SM of 8.18% on the slice of `afmlib/deffont.c` file, and a minimum (close to the intersected slice) of 1.31% on the slice of `afmlib/afm.c` file. In contrast, srcSlice produces a maximum SM of 1.67% on the slice of `states/gram.c` file, and a minimum SM of 1% on four files. As shown, the slice size produced by srcSlice is consistently closer to the intersected slice. The intersected slice size produced by srcSlice and CodeSurfer is equal to 91.1% and 52.8% on average, with a maximum of 100% and 76.3%.

The size of the intersected slice for the file `afmlib/deffont.c` is small (38 lines). In addition, the intersected slice size on files `e_88594.c` and `e_mac.c` from the same directory is zero. A closer examination of the slices, particularly the two files `e_88594.c`, and `e_mac.c` with the same size of 261 SLOC, shows that both files contain 258 SLOC of array initialization values of the form `{0x00, AFM_ENC_NONE}`. This indicates that imprecision with regard to large array initialization might be an issue. Because the CodeSurfer algorithm treats each element of an array as a distinct variable, the slice sizes from CodeSurfer for these files were 72.8% and 83.9%, respectively. This more precise approach requires complex dependence analysis; however, it leads to unnecessarily large slices [12]. In contrast, the srcSlice algorithm treats the entire array as a single variable, and the declaration of the array is detected and processed the same as a scalar variable. That is, if the array is not referenced inside the file, then the slice size is zero. The same scenario occurs in the `deffont.c` file, which contains a 262 SLOC array declaration.

The results of comparing the slices of srcSlice, CodeSurfer, and the intersection of these slices on 10 files from the program `findutils-4.4.2` are shown in Figure 8. As can be seen, the srcSlice results are consistently closer to the intersected slice, except for the file `find/defs.h`. In this case, the CodeSurfer slice size is only 1.7% of a 348 SLOC file, and we are unsure of the cause of the imprecision in CodeSurfer.

4.4. Control-flow accuracy and performance

To evaluate the accuracy of our tool in recovering control-flow paths, we ran srcSlice (highest setting) on the set of small feature programs, used in Section 4.1. A manual check of the control-flow paths produced by our tool showed that they were correct for these slices. Because this exercised many of the more complicated programming language features, we feel quite confident that the approach produces accurate results.

Comparison with CodeSurfer slicing time measurements (Time Highest column), as shown in Table II, shows our time to compute the slice along with the control-flow information, at the highest setting of srcSlice, remains drastically faster for all systems. The range of the computed flow edges using srcSlice tool was from 3095 edges for `ed-1.2` to 92,665 edges for `Quantlib 0.9.7`.

5. SCALABILITY OF SRCSLICE

We now demonstrate the scalability of our lightweight slicing approach. We ran srcSlice over the Linux kernel to demonstrate that the approach is effective and scalable for large-scale systems. For a recent version of the Linux kernel, srcSlice computed slices for the slicing criterion (F, M, V) in 748 s and produced a system dictionary of 1,334,504 slice profiles with a total slice size equal to ~4 MLOC. The data used in this section originate from slicing 974 versions of the Linux kernel that
Table III. Comparison of CodeSurfer and srcSlice to the intersected slice over 13 files from enscript-1.6.5.

<table>
<thead>
<tr>
<th>File name</th>
<th>LOC</th>
<th>SLOC</th>
<th>LOC</th>
<th>%</th>
<th>Safety margin</th>
<th>LOC</th>
<th>%</th>
<th>Safety margin</th>
<th>LOC</th>
<th>%</th>
<th>Safety margin</th>
<th>LOC</th>
<th>%</th>
<th>Safety margin</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/psgen.c</td>
<td>2860</td>
<td>1993</td>
<td>1351</td>
<td>67.8</td>
<td>1.75</td>
<td>863</td>
<td>43.3</td>
<td>1.12</td>
<td>771</td>
<td>57.1</td>
<td>89.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src/util.c</td>
<td>2156</td>
<td>1623</td>
<td>1227</td>
<td>75.6</td>
<td>1.48</td>
<td>853</td>
<td>52.6</td>
<td>1.03</td>
<td>827</td>
<td>67.4</td>
<td>97.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src/main.c</td>
<td>2660</td>
<td>1406</td>
<td>1178</td>
<td>83.8</td>
<td>1.59</td>
<td>768</td>
<td>54.6</td>
<td>1.04</td>
<td>739</td>
<td>62.7</td>
<td>96.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src/mkafmmmap.c</td>
<td>250</td>
<td>153</td>
<td>92</td>
<td>60.1</td>
<td>2.04</td>
<td>45</td>
<td>29.4</td>
<td>1.00</td>
<td>45</td>
<td>48.9</td>
<td>100.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>afmlib/strhash.c</td>
<td>386</td>
<td>268</td>
<td>145</td>
<td>54.1</td>
<td>1.36</td>
<td>145</td>
<td>54.1</td>
<td>1.36</td>
<td>107</td>
<td>73.8</td>
<td>73.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>afmlib/afmparse.c</td>
<td>1017</td>
<td>759</td>
<td>636</td>
<td>83.8</td>
<td>2.05</td>
<td>313</td>
<td>41.2</td>
<td>1.01</td>
<td>310</td>
<td>48.7</td>
<td>99.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>states/ex.c</td>
<td>2378</td>
<td>1536</td>
<td>813</td>
<td>52.9</td>
<td>3.35</td>
<td>279</td>
<td>18.2</td>
<td>1.15</td>
<td>243</td>
<td>29.9</td>
<td>87.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>states/gram.c</td>
<td>2408</td>
<td>1607</td>
<td>433</td>
<td>26.9</td>
<td>2.41</td>
<td>301</td>
<td>18.7</td>
<td>1.67</td>
<td>180</td>
<td>41.6</td>
<td>59.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>afmlib/afm.c</td>
<td>824</td>
<td>590</td>
<td>468</td>
<td>79.3</td>
<td>1.31</td>
<td>357</td>
<td>60.5</td>
<td>1.00</td>
<td>357</td>
<td>76.3</td>
<td>100.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>afmlib/afmtest.c</td>
<td>184</td>
<td>113</td>
<td>67</td>
<td>59.3</td>
<td>1.60</td>
<td>42</td>
<td>37.2</td>
<td>1.00</td>
<td>42</td>
<td>62.7</td>
<td>100.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>afmlib/deffont.c</td>
<td>379</td>
<td>323</td>
<td>311</td>
<td>96.3</td>
<td>8.18</td>
<td>38</td>
<td>11.8</td>
<td>1.00</td>
<td>38</td>
<td>12.2</td>
<td>100.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>afmlib/e_88594.c</td>
<td>284</td>
<td>261</td>
<td>190</td>
<td>72.8</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>afmlib/e_mac.c</td>
<td>284</td>
<td>261</td>
<td>219</td>
<td>83.9</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>16,070</td>
<td>10,893</td>
<td>7130</td>
<td></td>
<td></td>
<td>4004</td>
<td></td>
<td></td>
<td>3659</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1236</td>
<td>838</td>
<td>548</td>
<td>69.0</td>
<td>2</td>
<td>308</td>
<td>32.4</td>
<td>1</td>
<td>281</td>
<td>52.8</td>
<td>91.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td>184</td>
<td>113</td>
<td>67</td>
<td>26.9</td>
<td>1.31</td>
<td>0</td>
<td>11.8</td>
<td>1</td>
<td>0</td>
<td>12.2</td>
<td>59.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>2860</td>
<td>1993</td>
<td>1351</td>
<td>96.3</td>
<td>8.18</td>
<td>863</td>
<td>60.5</td>
<td>1.67</td>
<td>827</td>
<td>76.3</td>
<td>100.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The percentage (%) in the CodeSurfer and srcSlice columns (under slice size) is the slice size relative to the individual lines of code (LOC). The intersection percentage (%) is the slice size for each tool relative to the intersection.
have been released over a period of 17 years with a total of ~4.4 billion LOC. The studied Linux versions are identified and ordered by their release date and sequence number. The dataset ranges from the first version 1.0.0 released in 1994 to version 2.6.37.1 released in 2011. The total slice size is ~2 billion LOC, with an average slice size relative to LOC of 46.0%.

`srcSlice` builds a slice profile for each individual variable and then combines the output into a complete system dictionary. This allows for efficient use of memory and computation; thus, many scalability issues are avoided. Additionally, the parsing of the code from srcML further avoids computationally intensive searches, as the stream reader pulls tokens from input srcML one after another as needed. As such, very large systems can be sliced in a very reasonable amount of time. In other words, large increases of system size do not cripple our tool. The first version of the kernel with ~166 KLOC takes 7 s. Version 2.6.37.1 with ~13 MLOC takes approximately 13 min. These timings do not include conversion to srcML, which takes approximately 7 s for the first version and approximately 7 min for Version 2.6.37.1.

We now examine the slice size of our results, as this is considered to be a crucial issue [56], and therefore determines the main aspect of the quality of the generated slices. Ideally, we want to generate the smallest correct slice. Any unrelated statements or variables avoided improve the quality of the slice. Because the average slice size relative to LOC is 46.0%, we feel that our results are in a reasonable margin, based on the work by Binkley et al. [56, 59] and the results obtained in the previous sections. Furthermore, the results given in Figure 9 represent the difference between the system size and the slice size, both measured in LOC, over 974 versions of the Linux kernel. As expected, slice size increases proportionally with the system size.

6. RELATED WORK

Program slicing has a long and rich history. The idea was first proposed by Weiser [15] as an aid to debugging. However, since that time, program slicing has been applied to almost all aspects of software engineering including testing [1, 7, 8], maintenance and debugging [6, 9, 62], reverse engineering and reuse [63–66], comprehension [3, 67, 68], automatic parallelization [69], refactoring [70], and measurements [71, 72, 10].

These various applications of program slicing require different properties; thus, a number of different definitions have been proposed. These definitions are covered in detail in various surveys of the slicing literature [4, 11, 12, 17, 73–79]. Interestingly, each survey presents the definitions from a slightly different perspective. For example, references [4, 11, 12, 74, 76–78] focus mainly on the applications of program slicing techniques. Binkley et al. [2] compares different implementations...
and classifies them according to their empirical results. Harman et al. [75] compared and classified the techniques in order to predict future techniques and applications. Silva [17] and Harman et al. [75] compare and classify slicing techniques in order to identify relations between them. Most recently, Androutsopoulos et al. [73] present a detailed review of existing work on slicing at the level of finite state machine-based models.

Program slicing techniques can be broadly distinguished according to the type of slices such as the following:

- **Static** [6, 9, 22, 38, 40, 55, 56, 80] versus **Dynamic** [7, 11, 12, 18, 19, 35, 79, 81–89],
- **Closure** [18, 81, 34, 84, 25, 62, 46] versus **Executable** [19, 22, 35, 40, 59, 74, 85, 90],
- **Inter-procedural** [81, 34, 37, 91, 92, 21, 25, 41, 62, 93, 88, 22] versus **Intra-procedural** [6, 16, 18, 19, 40, 42, 43, 46, 84],
- **Forward** [23, 25, 35, 58, 68, 86] versus **Backward** [6, 19, 22–24, 26, 58, 63, 67, 68, 93–95],
- **Conditioned** [63, 90, 24, 94, 95], **Amorphous** [67, 91, 96, 97], **Union** [5, 82, 90], and **Quasi-static** slicing [95].

The differences between several definitions of program slice can be further explained in terms of slicing criterion. Harman et al. [75] survey several different slicing definitions in terms of slicing criteria. In general, slicing criterion are categorized into three types: static criterion [9, 22, 25, 40, 80], dynamic criterion [18, 19, 81, 84, 87], and hybrid criterion [4, 79].

One thing all these approaches have in common is that the slices are computed using pretty much the same underlying information. That is, all of these approaches use the PDG [27, 28], SDG [29], or a subset (e.g., control flow and data flow) to compute the slice. Furthermore, these approaches require that the entire PDG/SDG (or large portion) is pre-computed before computing the slice. This directly implies that the scalability and speed of any of these slicing approaches are limited by the ability to compute the PDG/SDG. Unfortunately, building the PDG/SDG is quite costly in terms of computational time and space.

A number of approaches have attempted to address the issues of scalability and time efficiency in program slicing. Reps et al. [98] define a new algorithm for inter-procedural slicing using the SDG that is asymptotically faster than the algorithm given by [25]. They claim that the new algorithm is significant providing roughly a sixfold speedup on examples of 348 LOC to 757 LOC. A parallel approach to compute static slices has been proposed by Danicic et al. [38]. There, the CFG is converted into a network of simultaneous processes whose parallel execution produces the slice. The notion of simultaneous dynamic slicing introduced by [85] incrementally builds the simultaneous slice using an iterative algorithm for all given test cases. Another SDG-based incremental slicing

![Figure 9. A comparison of the size of the srcSlice’s slices to the size of the Linux versions measured in MLOC, the x-axis is the version date.](image-url)
technique is proposed in [99]. The scope of a slice can be increased in steps by considering additional types of data dependencies at each step. Initially, the slice starts small and each increment increases its size. This concept of incremental slicing is used by [50]. The authors designed and implemented a static change impact analysis framework for large industrial software systems, with a codebase with 10 years of development at ABB Inc. In this study, a commercial version of CodeSurfer is used. They address the unique challenges in designing a static slicing-based change impact analysis framework for systems with over a million lines of code. Beszedes et al. [82] proposed the concept of union slices. They define the union slice as the union of dynamic slices for a finite set of test cases. The idea of union slicing is to approximate static slice, cheaply and effectively using unions of dynamic slices. A conditioned slicing algorithm for computing executable union slices is presented by Daninic et al. [90].

A lightweight slicing approach for object-oriented programs using dynamic and static analysis, called dependence-cache slicing, is proposed in [100]. This approach is based on dynamic data-dependence analysis and static control-dependence analysis. In the context of maintaining large-scale systems, another lightweight maintenance tool, called TuringTool, was proposed by Cordy et al. [31] and was designed to support several maintenance tasks using elision symbols. These symbols are used for viewing large source programs on a small screen by providing source-code projection. The importance of this hierarchy view is clear as the user can focus at some point of interest inside the code to any required level of detail. For example, if the debugger is interested only in those statements that affected by the value of a given variable, then only those statements are displayed on the screen. This is the same concept behind using slicing tools.

Because our approach is scalable in terms of time and program size as shown previously, the need was to evaluate the correctness of our results. Our evaluation criteria are based on the study proposed by Hoffner [52] in which he discussed several possible aspects to evaluate the performance of proposed slicing approaches. These aspects are the slice size compared with the original size, and the time and space complexities. The author compares two sets of dynamic slicing tools, and evaluation criteria were established for comparing them. In the evaluation, slice size was measured using either the number of retrieved statements or the number of vertices in the PDG when comparable approaches are applied with similar languages. Conversely, the author suggests that the code size is the best metric when these approaches handle similar languages. In the context of complexity, the difficulty of the approach is determined by the number of vertices in the intermediate representation models, and as a result the required execution time to complete the slicing process.

Our slicing approach presented in this paper computes a forward, static, non-executable (closure), inter-procedural program slice for each variable in a system. The approach differs drastically in that it does not depend on the creation of a fully computed SDG/PDG. Instead the dependence information is retrieved as needed while computing the slice for each variable in the program. Our approach is distinguished from this related work in multiple ways. The method used is not SDG/PDG-based. There is no graph to traverse or data-flow equations to be solved. Dependence information is computed as needed. Unlike most of the others, we slice over all the variables inside the system. That is, we compute the slice for every variable in the program including local and global. As new variables are encountered, they are added to the slice profile. However, we collect enough information to produce slices based on a given line number. Because our approach uses srcML (and the program being sliced is not compiled), it can also be applied to incomplete programs (e.g., missing libraries, missing includes, or a single file). Other tools require the system to be completed and compiled.

7. CONCLUSIONS AND FUTURE WORK

An approach and tool, srcSlice, for efficient and scalable slicing was presented and compared with an existing tool, namely, CodeSurfer from GrammaTech Inc. The approach extends Gallagher’s decomposition slice by adapting it to forward static slicing and by slicing at the set of statements that define a given variable. The results demonstrated that the approach produces fairly accurate slices as compared with CodeSurfer and is highly scalable. The srcSlice tool was shown to work on
a variety of C/C++ programs and language features. The lightweight approach taken is shown to be able to extract both inter-procedural and intra-procedural slices relative for each variable, which in turn allows line slices to be computed. Furthermore, control-flow information for each slice can be optionally computed. The limitations, with regard to accuracy, of the approach are related to deep pointer aliasing, certain array index analysis, analysis of virtual functions, and overloading.

The tool leverages the srcML format and toolkit and as such, it can be applied to incomplete and non-compiling programs. This is particularly useful when external libraries need to be excluded to reduce complexity or when they are not available (e.g., undergoing an API migration). The srcML toolkit allows srcSlice to be applied to very large software systems in a very efficient and scalable manner. The tool is demonstrated to be very efficient computationally. Compared with CodeSurfer, it is between three and four orders of magnitude faster for the Highest setting of CodeSurfer and the Highest setting of srcSlice. Furthermore, srcSlice at the Highest setting is approximately one-order of magnitude faster than CodeSurfer at the Super-Lite setting. However, CodeSurfer at the Super-lite setting does no intra-procedural data or control-flow analysis and as such produces very inaccurate slices.

In practice, we do not foresee a lightweight approach such as srcSlice being replacement for more heavyweight slicing tools like CodeSurfer. Rather, developers/organizations can use our approach to judge if it is prudent to expend the time and money to run a more rigorous analysis on a large software system. The difference in run time allows decisions to be made during a short meeting rather than waiting a week for results.

However, we feel the impact of this work on how developers and researchers leverage program slicing to address various software engineering tasks could be much broader than heavyweight approaches. srcSlice is freely available and as such represents one of the few slicing tools that researchers can take advantage of as part of their investigations. Because of its efficiency, very large systems can now be sliced in a very short time, opening new avenues for research. For example, one can now compute the slices for all variables on all versions of a system and examine how slices of a system change over time. This is impractical to do with current SDG-based tools. The approach can also allow for (near) real-time slicing within a development environment (especially on smaller portions of code).

This could assist in re-engineering and refactoring situations as slices seem to reflect different types of changes occurring in a system, possibly identifying refactoring changes [10, 89]. Tools that support comprehension/debugging tasks would not need to pre-compute slices, as they could be easily generated as needed by a developer exploring a code base. Also, slices could be regenerated as the developer updates and makes changes to the code enhancing the feedback time and benefits of slicing information. Applications for testing may also benefit from the fast computation time. Paths could be quickly determined, and possible suggestions/recommendations for new or missing test cases could be made. Clearly, one can currently define metrics based on slices, but now, computing slice-based metrics is feasible and may provide practical means to compute more accurate coupling and cohesion measures.

In the near future, we will be providing srcSlice as part of our srcML infrastructure and suite of tools. This will be freely available and open source under GNU General Public License (GPL). This will allow developers to customize some of the underlying analysis (e.g., pointer, virtuals) as they see necessary.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers of this manuscript (and earlier versions submitted to WCRE) for their very helpful comments. We also thank Keith Gallagher for his very insightful comments on an early version of this work. This work was supported in part by a grant from the US National Science Foundation CNS 13-05292/05217.

REFERENCES

SRCSLICE: VERY EFFICIENT AND SCALABLE FORWARD STATIC SLICING


64. Lakhotia A, Deprez J-C. Restructuring programs by tucking statements into functions. *Information and Software Technology* 1998; 40(11-12):677–689.

86. Korel B. Computation of dynamic slices for unstructured programs. *Journal of Auto-
AUTHORS’ BIOGRAPHIES

Hakam W. Alomari is an Assistant Professor in the Faculty of Information Technology at Jerash University, Jordan. His research is focused on developing and constructing methods for lightweight static program analysis. The objective is to develop new analysis methods that are highly scalable for application on very large software systems. He completed the PhD in Computer Science at Kent State University in Ohio, USA, in August 2012. He received his MS in Computer Science from Jordan University of Science and Technology, Jordan, in 2006 and his BS in Computer Science from Yarmouk University, Jordan, in 2004.

Michael L. Collard is an Assistant Professor in the Department of Computer Science at The University of Akron in Ohio, USA. His research interests focus on software evolution with 35 refereed publications, including a Most Influential Paper Award, in the areas of source code representation, analysis, transformation, and differencing. He is currently funded by a grant from the National Science Foundation to support his work on the srcML project. He received the PhD, MS, and BS in Computer Science from Kent State University.

Jonathan I. Maletic is a Professor in the Department of Computer Science at Kent State University in Ohio, USA. His research interests are centered on software evolution, and he has authored over 100 refereed publications in the areas of analysis, manipulation, transformation, comprehension, traceability, and visualization of software. The National Science Foundation has awarded Prof. Maletic’s several research grants, including a current award to support the srcML project. He received the PhD and MS in Computer Science from Wayne State University and the BS in Computer Science from The University of Michigan-Flint.

Nouh Alhindawi is an Assistant Professor in the Department of Software Engineering at Jadara University in Irbid, Jordan, since 2013. He obtained his PhD in Computer Science from Kent State University, USA, in 2013 with Dr. Jonathan Maletic as his advisor. He obtained the master’s degree from Al-Balqa Applied University, Jordan, in 2006 and the BS degree from Yarmouk University, Jordan, in 2004. His research interests are in software engineering, information retrieval, and using information retrieval approaches for improving software comprehension.

Omar Meqdadi is an Assistance Professor in the Department of Computer Science and Software Engineering at University of Wisconsin-Platteville in Wisconsin, USA. His research interests are in software engineering with focus on software evolution, machine learning, program slicing, and using information retrieval for mining software repositories during the software evolution. He obtained his PhD in Computer Science from Kent State University. He has prior degrees in Computer Engineering from Jordan University of Science and Technology, Jordan.