Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1651

srcSlice: very efficient and scalable forward static slicing

Hakam W. Alomari[1], Michael L. Collard[2], Jonathan I. Maletic[3,*,†], Nouh Alhindawi[4] and Omar Meqdadi[5]

[1]*Faculty of Information Technology, Jerash University, Jerash, Jordan*
[2]*Department of Computer Science, The University of Akron, Akron, Ohio, USA*
[3]*Department of Computer Science, Kent State University, Kent, Ohio, USA*
[4]*Faculty of Sciences and Information Technology, Jadara University, Irbid, Jordan*
[5]*Department 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], interprocedural 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

(SDG) [29]. 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.
[§]CodeSurfer is a produced of GammaTech Inc. www.grammatech.com.

DOI: 10.1002/smr

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 *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. *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 *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

DOI: 10.1002/smr

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 transitively 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

```
s1.  cin >> a;
s2.  cin >> b;
s3.  c = a + b;
s4.  cout << c;
s5.  c = a - b;
s6.  cout << c;
```

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

DOI: 10.1002/smr

(a)

```
1.  main(){
2.      int sum = 0;
3.      int i = 1;
4.      while (i<=10){
5.          sum = sum + i;
6.          i++;
7.      }
8.      cout<<sum;
9.      cout<<i;
10. }
```

(b)

Slice Profile(sum)= @index(1), slines={2,5,8},

Slice Profile(i)= @index(2), slines={3,4,5,6,9}, dvars={sum}

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.

each line starting from the beginning (line 1) and determining the forward slice. Definition-use chains are followed along with forward control dependencies. The profile for *sum* is created first as it is encountered in line 2 ($slines(sum) = \{2\}$). Then the profile for i is created in line 3 ($slines(i) = \{3\}$). The two profiles are updated as follows for the given line number:

4: $slines(sum) = \{2\}$; $slines(i) = \{3, 4\}$, $controledges(i) = \{(3,4)\}$
5: $slines(sum) = \{2, 5\}$, $controledges(sum) = \{(2,5)\}$; $slines(i) = \{3, 4, 5\}$, $dvariables(i) = \{sum\}$, $controledges(i) = \{(3,4),(4,5)\}$
6: $slines(sum) = \{2, 5\}$, $controledges(sum) = \{(2,5)\}$; $slines(i) = \{3, 4, 5, 6\}$, $dvariables(i) = \{sum\}$, $controledges(i) = \{(3,4),(4,5),(5,6)\}$
8: $slines(sum) = \{2, 5, 8\}$, $controledges(sum) = \{(2,5),(2,8),(5,8)\}$; $slines(i) = \{3, 4, 5, 6\}$, $dvariables(i) = \{sum\}$, $controledges(i) = \{(3,4),(4,5),(5,6)\}$
9: $slines(sum) = \{2, 5, 8\}$, $controledges(sum) = \{(2,5),(2,8),(5,8)\}$; $slines(i) = \{3, 4, 5, 6, 9\}$, $dvariables(i) = \{sum\}$, $controledges(i) = \{(3,4),(4,5),(4,9),(5,6),(6,9)\}$

These are the slice profiles for each variable, and the complete slice is then computed by finding the control-flow edges and then taking the union of the *slines* with the slice profiles of the *dvariables*, *cfunctions*, and *pointers*, minus any lines that are before the initial definition of the slice variable (i.e., the set $\{1, \ldots, def(v) - 1\}$). Thus, because *sum* is data dependent on i, the complete slice for $i = slines(i) \cup slines(sum) - \{1, 2\}$. This comes out to $\{3, 4, 5, 6, 8, 9\}$. This final computation can be carried out for all variables via a single pass through the dictionary.

2.3. Definition of the criterion

We now present a definition of our slicing criterion and how a slice is computed using the criterion.

Definition 1: Forward Decomposition Slice
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

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

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), and (F, M, V), where $F = \{f_1, f_2, \ldots, f_j\}$ is the finite set of files in p, $M = \{m_1, m_2, \ldots, m_y\}$ 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

DOI: 10.1002/smr

$$mds(f, m) = \bigcup_{i=1}^{d} ds(f, m, v_i).$$

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_3) includes the statements $\{s_3, s_4, s_6\}$. 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

DOI: 10.1002/smr

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 $stmt_i$ that has just been included in the slice profile of variable *v*, an immediate control predicate of $stmt_i$, 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*

DOI: 10.1002/smr

```
Input:  Slicing Criterion (F, M, V)
Output:  System Dictionary - a table of slice profiles for all variables
Sp: Slice Profile ∀v ∈ V
VL: Local variables {(v_i, Sp(v_i))| ∀m ∈ M}
VG: Global variables {(v_i, Sp(v_i))| ∀f ∈ F}
VP: Declared pointers {(v_i, v_x)| v_i is a pointer alias of v_x, ∀m ∈ M}
LP: Loop profile {(s, e) | s is starting line and e is ending line, ∀m ∈ M}
LS: Stack of line numbers for loop profiles
For each file f_i ∈ F, ComputeSliceProfile(m_i) for all methods m_i ∈ F.
algorithm ComputeSliceProfile(m)
1.  repeat
2.    for each stmt_i ∈ m in statement sequence do
3.      case: stmt_i is a declaration-statement or a parameter-list
4.        for each v ∈ stmt_i do
5.          if (v ∉ VL) then
6.            Sp(v).@index ← {index of v}
7.          end if
8.          Sp(v).slines ← Sp(v).slines ∪ {stmt_i.linenumber}
9.          VL ← (v, Sp(v))
10.       end for
11.     case: stmt_i is an expression-statement
12.       found ← false
13.       for each v ∈ stmt_i and v ∈ VL do
14.         found ← true
15.         if v is a r-value and v(r-value) ≠ v(l-value) then
16.           Sp(v).dvariables ← {v(l-value)}
17.           Sp(v).slines ← Sp(v).slines ∪ {stmt_i.linenumber}
18.           if v ∈ VP then  // Retrieve "a", where v is a pointer to "a"
19.             a ← Vp.variable(v)
20.             Sp(a).slines ← Sp(a).slines ∪ {stmt_i.linenumber}
21.             Sp(a).dvariables ← Sp(a).dvariables ∪ {v(l-value)}
22.             VL ← (a, Sp(a))
23.           end if
24.         else if v(l-value) is a pointer alias of v(r-value) then
25.           Sp(v).pointers ← {v(l-value)}
26.           Sp(v).slines ← Sp(v).slines ∪ {stmt_i.linenumber}
27.           VP ← {v(l-value), v}              //update declared pointers
28.         end if
29.         VL ← (v, Sp(v))
30.       end for
31.       if !(found) then
           // For a global variable, repeat lines 13 - 30 using the set VG instead of VL
32.       end if
33.     case: stmt_i is a function-call  //Process all arguments, with their indices
34.       for each v ∈ argument-list do
35.         if v ∈ VL then //local variables
36.           Sp(v).slines ← Sp(v).slines ∪ {stmt_i.linenumber}
37.           Sp(v).cfunctions ← Sp(v).cfunctions ∪ {function name}
38.           Sp(v).@index ← {index of v}
39.           VL ← (v, Sp(v))
40.         else if v ∈ VG then  //global variables
              // Repeat lines 38 - 41 using the set VG
41.         end if
42.         if v ∈ VP then  //pointer variables
              // Repeat lines 25 to 27
43.         end if
44.       end for
45.     case: stmt_i is a control-flow-statement
46.       LS ← push(stmt_i.linenumber)  //Store the line of the condition
47.     case: stmt_i is an end-statement of loop or condition then
48.       end-stmt.linenumber ← LS.pop()
49.       LP ← (end-stmt.linenumber, stmt_i.linenumber)
50.   end for
51. until done with m
```

Figure 3. Algorithm to compute the intra-procedural slice for each variable in a system

line 38 in Figure 3) as the slice profile is computed. Then from this index mapping, we recover all statements that are included in the slice profile of the parameters (*see line 22 in* Figure 6), and take the union of a variable's *slines* along with the slices of *cfunctions* (*see lines 8 to 11 in* Figure 6). An example of index

DOI: 10.1002/smr

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 *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)	1.	`int main(){`
	2.	`  int sum = 0;`
	3.	`  int i = 1;`
	4.	`  while (i<=10){`
	5.	`  foo(sum,i);`
	6.	`  }`
	7.	`  cout<<sum;`
	8.	`}`
	9.	`void foo(int &x, int &y){`
	10.	`  fun(x);`
	11.	`  y++;`
	12.	`}`
	13.	`void fun(int z){`
	14.	`  z++;`
	15.	`}`
(b)	*Slice Profile(main/sum)= @index(1), slines={2,5,7,9,10,13,14}, cfunctions={(foo,1);(fun,1)}*	
	Slice Profile(main/i)= @index(2), slines={3,4,5,9,11}, cfunctions={(foo,2)}	
	Slice Profile(foo/x)= @index(1), slines={9,10,13,14}	
	Slice Profile(foo/y)= @index(2),slines={9,11}	
	Slice Profile(fun/z)= @index(1),slines={13,14}	

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

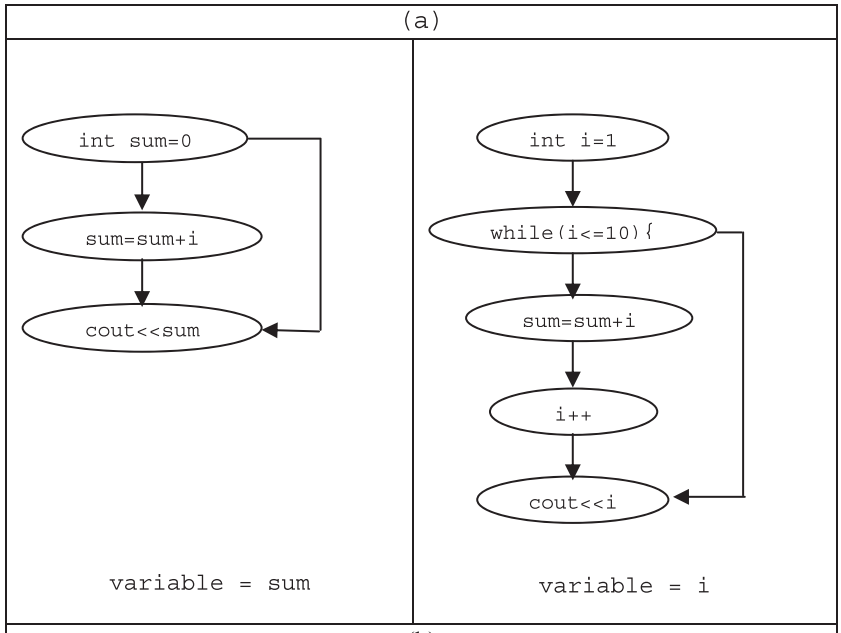

Figure 5. Control-flow path computation for the code in Figure 2. (a) Control-flow edges (b) system dictionary with slice profiles including control-flow edges shown in (a).

DOI: 10.1002/smr

```
Input:  Slicing Criterion (F, M, V)
Output:  System Dictionary - a table of slice profiles for all variables
Sp: Slice Profile ∀v ∈ V
Spi: an instance of Sp
VL: Local variables {(v_i, Sp(v_i)) | ∀m ∈ M}
VG: Global variables {(v_i, Sp(v_i)) | ∀f ∈ F}

For each file f_i ∈ F, ComputeInterprocedural(f_i)

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 ∈ 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
9.            Sp(v).cfunctions ← Sp(v).cfunctions ∪ Spi.cfunctions
10.           Sp(v).pointers ← Sp(v).pointers ∪ Spi.pointers
11.           Sp(v).dvariables ← Sp(v).dvariables ∪ Spi.dvariables
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).@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 ∈ 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
```

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

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

DOI: 10.1002/smr

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

```
Input:  Slicing Criterion (F, M, V)
Output:  System Dictionary - a table of slice profiles for all variables
Sp: Slice Profile ∀v ∈ V
VL: Local variables {(vi, Sp(vi))| ∀m ∈ M}
VG: Global variables {(vi, Sp(vi))| ∀f ∈ F}
LP: Loop profile {(s, e) | s is starting line and e is ending line, ∀m ∈ M}
For each file fi ∈ F, ComputeControlEdges(mi) for all methods mi ∈ F.

algorithm ComputeControlPaths(m)
1.    for each v ∈ VL do
2.      for each L ∈ LP do
3.        for each sl ∈ Sp(v).slines do
4.          predecessor ← 0
5.          false-successor ← 0
6.          if (sl <= L's control-flow-statement line number) then
7.            predecessor ← sl
8.          end if
9.          if (sl >= L's control-flow-statement line number) then
10.          false-successor ← sl
11.         end if
12.         true-successor ← first line of v inside L block-if it exists
13.       end for
14.       if (predecessor < false-successor) then
15.         Sp(v).controledges ← Sp(v).controledges ∪ {(predecessor, true-successor)}
16.         Sp(v).controledges ← Sp(v).controledges ∪ {(predecessor, false-successor)}
17.       end if
18.     end for
19.     for each sl ∈ Sp(v).slines do
20.       // edges between successive  lines  with care of if-else matching
21.       if (!(sl ∈ if-block And {next line after sl}∈ else-block)) then
22.         Sp(v).controledges ← Sp(v).controledges ∪ {(sl, next line after sl)}
23.       end if
                    // if this slicing line does not belong to any loop or condition block,
                    // then connect this line to the first successive slicing line that is also
                    // outer of any loop or condition
24.       if (!(sl ∈ control-block)) then
25.         Sp(v).controledges ← Sp(v).controledges ∪ {(sl, next outer slicing line)}
26.       end if
27.     end for
28.     VL ← (v, Sp(v))
29.   end for
```

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

DOI: 10.1002/smr

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 *srcSlice*. It is written in C++ and uses srcML as input. The system to be sliced is first converted to srcML. The tool *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 *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 *srcSlice*) can process each of these program elements as they reached, with an exact mapping back to the original source code. For *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

DOI: 10.1002/smr

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: *Slice Profile*$(i) = @index(2)$, $slines = \{3,4,5,6,8,9\}$, $dvars = \{sum\}$
- With pruning: *Slice Profile*$(i) = @index(2)$, $slines = \{6,9\}$

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.

DOI: 10.1002/smr

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

DOI: 10.1002/smr

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

[¶]The version of CodeSurfer was obtained from GrammaTech in May 2013. It is a free academic license version.
[‖]See the Wisconsin Program-Slicing Project http://www.cs.wisc.edu/wpis/html/, CodeSurfer.

DOI: 10.1002/smr

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 ~1 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* $\cong$ *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 slice time, the sum of individual slice times for each slicing criterion, is denoted by

$$TST(SC) = \sum_{i=1}^{n} ST(sc_i)$$

DOI: 10.1002/smr

Furthermore, the total time overhead for one build for both slicers is denoted by $T(SC) = BT + TST(SC)$. 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 $ST(sc_i) \cong 0.00$. 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.

DOI: 10.1002/smr

Table I. Feature benchmark results and comparison of *CodeSurfer* and *srcSlice*.

	Size			Slicing criterion		*CodeSurfer*				*srcSlice*			
Program	LOC	Files	Functions	Method	Variable	Slices taken	Slicing time	Slice size	%	Slices taken	Slicing time	Slice size	%
Information _flow	112	1	12	Main	hi	1	1.481	32	28.6	1	0.978	27	24.1
				All criterions		149		66	58.9	22		48	42.9
Sum	21	1	2	Main	Sum	1	0.989	6	28.6	1	0.531	4	19.0
				All criterions		26		14	66.7	2		8	38.1
Wc	39	1	3	line_char_count	eof_flag	1	1.212	16	41.0	1	0.362	10	25.6
				Scan_line	i	1		7	17.9	1		4	10.3
				All criterions		46		24	61.5	9		19	48.7
Pointer	36	1	5	Main	var1	1	1.519	11	30.6	1	0.358	15	41.7
				All criterions		37		25	69.4	8		17	47.2
Testcases	114	1	14	Main	var1	1	7.662	50	43.9	1	0.641	44	38.6
				All criterions		156		79	69.3	24		56	49.1
Callofcall	24	1	3	Main	var1	1	2.921	4	16.7	1	0.411	4	16.7
				All criterions		23		13	54.2	7		10	41.7
Total	346	6	39			444	15.78	347		79	3.28	266	
Average	57.7	1	6.5			34.2	2.63	26.7	45.2	6.1	0.55	20.5	34.1

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).

DOI: 10.1002/smr

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.

DOI: 10.1002/smr

Table II. Performance benchmark results and comparison of *CodeSurfer* and *srcSlice*.

	Size			*CodeSurfer*					*srcSlice*				
Program	LOC	Files	Functions	Slices taken	Time highest	Time Super-Lite	Slice size	%	Slices taken	Time highest	Time Super-Lite	Slice size	%
ed-1.2	3087	10	126	4438	21	5	1782	57.7	420	3	1	1136	36.8
ed-1.6	3260	10	128	4527	19	5	1863	57.1	442	3	1	1199	36.8
which-2.20	3586	14	51	1429	15	9	736	20.5	196	2	1	754	21.0
wdiff-0.5	3874	13	56	1097	11	4	652	16.8	154	2	1	581	15.0
barcode-0.98	5205	18	74	4590	29	9	2177	41.8	392	3	1	1728	33.2
acct-6.5	8749	27	127	4983	47	11	2510	28.7	635	5	1	1767	20.2
enscript-1.4.0	18,162	52	180	9456	71	15	5916	32.6	920	7	2	3472	19.1
make-3.82	36,397	58	474	17,012	807	27	9446	26.0	2914	143	3	10,598	29.1
libkate 0.3.8	53,441	111	2210	28,017	109	33	12,846	24.0	4993	17	6	12,622	23.6
enscript-1.6.5.2	56,491	107	488	20,234	184	35	12,907	22.8	2580	90	5	10,635	18.8
enscript-1.6.5	56,494	107	488	20,252	184	35	12,913	22.9	2579	85	5	10,636	18.8
enscript-1.6.5.1	56,494	107	488	20,252	185	35	12,913	22.9	2579	85	5	10,636	18.8
a2ps-4.10.4	57,052	188	1104	24,493	393	116	14,249	25.0	3844	61	6	10,756	18.9
findutils-4.4.2	72,384	314	1141	23,641	215	94	13,689	18.9	5657	79	9	16,764	23.2
radius-1.0	82,029	196	1719	38,487	335	148	19,218	23.4	6997	170	11	21,905	26.7
dico-2.2	119,592	332	2504	52,297	1763	246	28,639	23.9	10,451	471	16	29,418	24.6
cvs-1.12.10	144,278	340	2027	74,328	286,328	371	40,869	28.3	9779	192	18	34,955	24.2
Clanlib -0.8.1	181,064	1137	5624	68,716	280,914	344	45,447	25.1	13,514	58	21	30,488	16.8
HippoDraw-1.21.3	248,592	1385	8627	—	—	377	—	—	12,400	66	29	25,976	10.5
Quantlib 0.9.7	599,802	3389	15,696	—	—	384	—	—	27,348	168	99	53,476	8.9
Total	1,810,033	7915	43,332	418,249			238,772		69,046			210,050	
Average	90,502	396	2167	23,236	31,757	115	13,265	28.8	3836	86	12	11,055	22.4

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*.

DOI: 10.1002/smr

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

DOI: 10.1002/smr

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.[††] 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 senario 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

[††]See http://www.dwheeler.com/sloccount/sloccount.html

DOI: 10.1002/smr

Table III. Comparison of *CodeSurfer* and *srcSlice* to the intersected slice over 13 files from enscript-1.6.5.

	Size		Slice size						Intersection		
			CodeSurfer			*srcSlice*					
enscript-1.6.5 File name	LOC	SLOC	LOC	%	Safety margin	LOC	%	Safety margin	LOC	*CodeSurfer* intersection %	*srcSlice* intersection %
src/psgen.c	2860	1993	1351	67.8	1.75	863	43.3	1.12	771	57.1	89.3
src/util.c	2156	1623	1227	75.6	1.48	853	52.6	1.03	827	67.4	97.0
src/main.c	2660	1406	1178	83.8	1.59	768	54.6	1.04	739	62.7	96.2
src/mkafmmap.c	250	153	92	60.1	2.04	45	29.4	1.00	45	48.9	100.0
afmlib/strhash.c	386	268	145	54.1	1.36	145	54.1	1.36	107	73.8	73.8
afmlib/afmparse.c	1017	759	636	83.8	2.05	313	41.2	1.01	310	48.7	99.0
states/ex.c	2378	1536	813	52.9	3.35	279	18.2	1.15	243	29.9	87.1
states/gram.c	2408	1607	433	26.9	2.41	301	18.7	1.67	180	41.6	59.8
afmlib/afm.c	824	590	468	79.3	1.31	357	60.5	1.00	357	76.3	100.0
afmlib/afmtest.c	184	113	67	59.3	1.60	42	37.2	1.00	42	62.7	100.0
afmlib/deffont.c	379	323	311	96.3	8.18	38	11.8	1.00	38	12.2	100.0
afmlib/e_88594.c	284	261	190	72.8		0	0.0		0	0.0	0.0
afmlib/e_mac.c	284	261	219	83.9		0	0.0		0	0.0	0.0
Total	16,070	10,893	7130			4004			3659		
Average	1236	838	548	69.0	2	308	32.4	1	281	52.8	91.1
Min	184	113	67	26.9	1.31	0	11.8	1	0	12.2	59.8
Max	2860	1993	1351	96.3	8.18	863	60.5	1.67	827	76.3	100

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.

DOI: 10.1002/smr

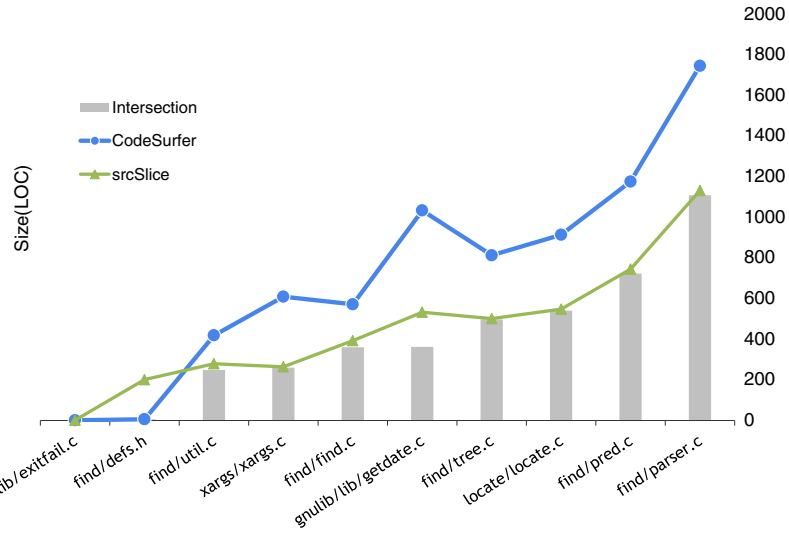

Figure 8. Comparison of *CodeSurfer*, *srcSlice*, and *slice intersection* over 10 files from the program findutils-4.4.2 ordered by intersection size. Except for a single file, *srcSlice* was much closer to the slice intersection.

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

DOI: 10.1002/smr

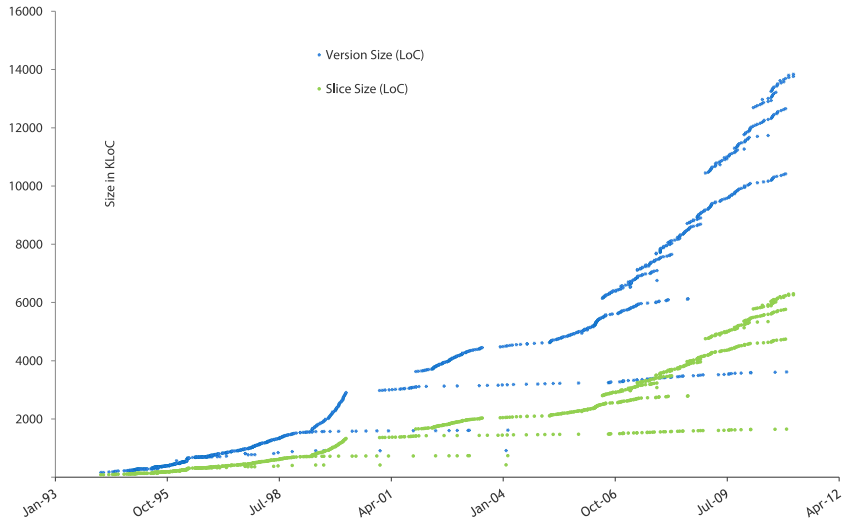

Figure 9. A comparison of the size of the *srcSlice*'s slices to the size of the Linux versions measured in MLOC, the *x*-axis is the version date.

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, 25, 38, 40, 55, 56, 80] versus *Dynamic* [7, 11, 12, 18, 19, 35, 62, 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

DOI: 10.1002/smr

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

DOI: 10.1002/smr

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

1. Binkley D. The application of program slicing to regression testing. *Information & Software Technology* 1998; **40**(11-12):583–594.
2. Binkley D, Harman M. A survey of empirical results on program slicing. *Advances in Computers* 2004b; **62**:105–178.
3. De Lucia A, Fasolino AR, Munro M. Understanding function behaviors through program slicing. In P Proceedings of the 4th International Workshop on Program Comprehension (WPC), 1996; 9–18.

DOI: 10.1002/smr

4. De Lucia A. Program slicing: methods and applications. In Proceedings of Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation, 2001; 142–149.
5. Faragó C. Union slices for program maintenance. In Proceedings of the International Conference on Software Maintenance (ICSM), 2002; 12.
6. Gallagher KB, Lyle JR. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering (TSE '91)* 1991; **17**:751–761.
7. Gupta R, Harrold MJ, Soffa ML. An approach to regression testing using slicing. In Proceedings of the IEEE conference on software maintenance, 1992; 299–308.
8. Harman M, Danicic S. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability* 1995; **5**:143–162.
9. Lyle JR, Weiser M. Automatic program bug location by program slicing. In Proceedings of Second international conference on computers and applications, 1987; 877–882.
10. Pan K, Kim S, Whitehead JE, Jr. Bug classification using program slicing metrics. In Sixth IEEE International Workshop on Source Code Analysis and Manipulation, (SCAM '06), 2006.
11. Tip F. A survey of program slicing techniques. *Journal of Programming Language* 1995; **3**:121–189.
12. Xu B, Qian J, Zhang X, Wu Z, Chen L. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 2005; **30**(2):1–36.
13. Reps T, Ball T, Das M, Larus J. The use of program profiling for software maintenance with applications to the Year 2000 Problem. In Proceedings of ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, Sept. 22-25, 1997; 432–449.
14. Tsai W-T. Application of data-centered approach to year 2000 problem. In Proceedings of COMPSAC '97, Aug. 11-15, 1997; 287–288.
15. Weiser M. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. University of Michigan, Ann Arbor, MI, USA, Ph.D. Dissertation Thesis, 1979.
16. Weiser M. Program slicing. In Proceedings of the International Conference on Software Engineering (ICSE '81). San Diego, California, USA, 1981; 439–449.
17. Silva J. A vocabulary of program slicing-based techniques. *ACM Computing Surveys* 2012; **44**(3):1–41.
18. Agrawal H, Horgan JR. Dynamic program slicing. *SIGPLAN Not* 1990; **25**(6):246–256.
19. Korel B, Laski J. Dynamic program slicing. *Information Processing Letters* 1988; **29**(3):155–163.
20. Gallagher K, Binkley DW. Program slicing. In *Frontiers of Software Maintenance (FoSM '08)*. Beijing, China, 2008; 58–67.
21. Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. *SIGPLAN Not.* 1988; **23**:35–46.
22. Weiser M. Program slicing. *IEEE Transactions on Software Engineering (TSE)* 1984; **10**(4):352–357.
23. Binkley D, Harman M. Forward slices are smaller than bacward slices. In Proceedings of Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation, 2005; 15–24.
24. Fox C, Harman M, Hierons R, Danicic S. Backward conditioning: a new program specialisation technique and its application to program comprehension. In Proceedings of 9th IEEE InternationalWorkshop on Program Comprenhesion, 2001; 89–97.
25. Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 1990; **12**(1):26–60.
26. Kumar S, Horwitz S. Better slicing of programs with jumps and switches. In Proceedings of Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE), Springer-Verlag, London, UK, April 08–12, 2002b; 96–112
27. Ferrante J, Ottenstein KJ, Warren JD. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 1987; **9**(3):319–349.
28. Kuck DJ, Kuhn RH, Padua DA, Leasure B, Wolfe M. Dependence graphs and compiler optimizations. In Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1981; 207–218.
29. Liang D, Harrold MJ. Slicing objects using system dependence graphs. In Proceedings of the International Conference on Software Maintenance (ICSM), 1998; 358–367.
30. Alomari HW, Collard ML, Maletic JI. A very efficient and scalable forward static slicing approach. In Proceedings of IEEE International Working Conference on Reverse Engineering (WCRE'12), Kingston, Ontario, Canada, October 15-18, 2012; 425–434.
31. Cordy JR, Eliot NL, Robertson MG. turingtool: a user interface to aid in the software maintenance task. *IEEE Transactions on Software Engineering (TSE)* 1990; **16**(3):294–301.
32. Collard ML, Maletic JI, Robinson BP. A lightweight transformational approach to support large scale adaptive changes. In Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM). Timisoara, Romania, 2010; 1–10.
33. Collard ML, Decker M, Maletic JI. Lightweight transformation and fact extraction with the srcML toolkit. In Proceedings of 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11), Williamsburg, VA, USA, Sept 25-26, 2011; 10.
34. Bergeretti J-F, Carre' BA. Information-flow and data-flow analysis of while-programs. In ACM Trans. Program. Lang. Syst., (ACM), 1985; **7**(1):37–61.

DOI: 10.1002/smr

35. Binkley D, Danicic S, Gyimothy T, Harman M, Kiss A, Ouarbya L. Formalizing executable dynamic and forward slicing. In Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop, 2004a; 43–52.
36. Dragan N, Collard ML, Maletic JI. Reverse engineering method stereotypes. In Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, Pennsylvania USA, September 25-27, 2006; 24–34.
37. Gallagher KB. Some notes on interprocedural program slicing. In Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), 2004; 36–42.
38. Danicic S, Harman M, Sivagurunathan Y. A parallel algorithm for static program slicing. *Information Processing Letters* 1995; **56**(6):307–313.
39. Harman M, Lakhotia A, Binkley D. Theory and algorithms for slicing unstructured programs. *Information & Software Technology* 2006; **48**(7):549–565.
40. Gallagher KB. Using Program Slicing for Program Maintenance, Dissertation, University of Maryland, Maryland, 1990.
41. Jiang J, Zhou X, Robson DJ. Program slicing for C – the problems in implementation. In Proceedings of Proceedings of the Conference on Software Maintenance, 1991; 182–190.
42. Agrawal H. On slicing programs with jump statements. *SIGPLAN Not.* 1994; **29**(6):302–312.
43. Ball T, Horwitz S. Slicing programs with arbitrary control-flow. In Proceedings of the First International Workshop on Automated and Algorithmic Debugging, 1993; 206–222.
44. Harman M, Danicic S. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance Research and Practice* 1998a; **10**(6):415–441.
45. Kumar S, Horwitz S. Better slicing of programs with jumps and switches. In Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, 2002a; 96–112.
46. Ottenstein KJ, Ottenstein LM. The program dependence graph in a software development environment. *SIGSOFT Software Engineering Notes* 1984; **9**:177–184.
47. Mock M, Atkinson DC, Chambers C, Eggers SJ. Program slicing with dynamic points-to sets. *IEEE Transactions on Software Engineering* 2005; **31**(8):657–678.
48. Shapiro M, Horwitz S. Fast and accurate flow-insensitive points-to analysis. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages, 1997; 1–14.
49. Steensgaard B. Points-to analysis in almost linear time. In Proceedings of the 23rd ACM Symposium on Principles of Programming Languages, 1996; 32–41.
50. Acharya M, Robinson B. Practical change impact analysis based on static program slicing for industrial software systems. In Proceedings of the 33rd International Conference on Software Engineering (ICSE), 2011; 746–755.
51. Murphy GC, Notkin D. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1998; **7**(2):158–191.
52. Hoffner T. Evaluation and comparison of program slicing tools. Technical report. Sweden: Department of Computer and Information Science, Linkping University, 1995.
53. Horwitz S, Reps T. The use of program dependence graphs in software engineering. In Proceedings of the 14th International Conference on Software Engineering (ICSE), 1992; 392–411.
54. Bent L, Atkinson DC, Griswold WG. A qualitative study of two whole-program slicers for C. Technical Report CS20000643, University of California at San Diego, 2000.
55. Danicic S, Fox C, Harman M, Hierons R, Howroyd J, Laurence MR. Static program slicing algorithms are minimal for free liberal program schemas. *Computer Journal* 2005; **48**(6):737–748.
56. Binkley D, Gold N, Harman M. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2007; **16**(2):8.
57. Atkinson DC, Griswold WG. Effective whole-program analysis in the presence of pointers. *SIGSOFT Software Engineering Notes* 1998; **23**(6):46–55.
58. Binkley D, Harman M. A large-scale empirical study of forward and backward static slice size and context sensitivity. In Proceedings of the International Conference on Software Maintenance (ICSM), 2003; 44–53.
59. Binkley D, Gold N, Harman M, Li Z, Mahdavi K. An empirical study of executable concept slice size. In Proceedings of the 13th Working Conference on Reverse Engineering, 2006; 103–114.
60. Binkley D, Harman M, Hassoun Y, Islam S, Li Z. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Software Systems (JSS)* 2010; **83**(1):96–107.
61. Binkley D. Slicing in the presence of parameter aliasing. In In Proceedings of the Third Software Engineering Research Forum. Orlando, Florida, 1993; 261–268.
62. Kamkar M. Interprocedural dynamic slicing with applications to debugging and testing, Linköping University: Sweden, 1993.
63. Canfora G, Cimitile A, De Lucia A. Conditioned program slicing. *Information & Software Technology* 1998; **40**(11):595–607.
64. Lakhotia A, Deprez J-C. Restructuring programs by tucking statements into functions. *Infromation and Software Technology* 1998; **40**(11-12):677–689.
65. Liu L, Ellis R. An approach to eliminating COMMON blocks and deriving ADTs from Fortran programs. In *Technical Report*. UK: University of Westminster, 1993.

DOI: 10.1002/smr

66. Simpson D, Valentine S, Mitchell R, Liu L, Ellis R. RECOUP\-Maintaining Fortran. *SigPLAN Fortran Forum* 1993; **12**(3):26–32.
67. Harman M, Danicic S. Amorphous program slicing. In Proceedings of the 5th International Workshop on Program Comprehension (WPC), 1997; 70.
68. Jackson D, Rollins EJ. Chopping: a generalization of slicing. Technical Report, Carnegie Mellon University, 1994.
69. Weiser M. Reconstructing sequential behaviour from parallel behaviour projections. *Information Processing Letters* 1983; **17**(10):129–135.
70. Komondoor R, Horwitz S. Semantics-preserving procedure extraction. In Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2000; 155–169.
71. Bieman JM, Ott LM. Measuring functional cohesion, *IEEE Transactions on Software Engineering (TSE)* 1994; **20**(8):644–657.
72. Ott LM, Thuss JJ. Slice based metrics for estimating cohesion. In Proceedings of IEEE-CS International Metrics Symposium, 1993; 78–81.
73. Androutsopoulos K, Clark D, Harman M, Krinke J, Tratt L. State-based model slicing: a survey. *ACM Computing Surveys* 2013; **45**(4):1–36.
74. Binkley D, Gallagher KB. 1996. Program slicing. *Advances in Computers* **62**; 105–178.
75. Harman M, Danicic S, Sivagurunathan Y, Simpson D. The next 700 slicing criteria. In Proceedings of the 2nd UK Workshop on Program Comprehension, 1996; 1–16.
76. Harman M, Gallagher KB. Program slicing. *Information and Software Technology* 1998b; **40**:577–582.
77. Harman M, Hierons, R. An overview of program slicing. *Software Focus* 2001a; **2**(3):85–92.
78. Kamkar M. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software* 1995; **31**(3):197–214.
79. Venkatesh GA. The semantic approach to program slicing. In Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, 1991b; 107–119.
80. Lanubile F, Visaggio G. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering* 1997; **23**(4):246–259.
81. Agrawal H, DeMillo RA, Spafford EH. Dynamic slicing in the presence of unconstrained pointers. In Proceedings of the symposium on Testing, analysis, and verification, 1991; 60–73.
82. Beszedes A, Gyimothy T. Union slices for the approximation of the precise slice. In Proceedings of IEEE International conference on software maintenance (ICSM), 2002; 12–20.
83. Chen TY, Cheung YY. Dynamic program dicing. In Proceedings of the Conference on Software Maintenance, 1993; 378–385.
84. Gopal R. Dynamic program slicing based on dependence relations. In Proceedings of the Conference on Software Maintenance, 1991; 191–200.
85. Hall RJ. Automatic extraction of executable program subsets by simultaneous program slicing. *Journal of Automated Software Engineering* 1995; **2**(1):33–53.
86. Korel B, Yalamanchili S. Forward computation of dynamic program slices. In Proceedings of the 1994 international symposium on software testing and analysis, 1994; 66–79.
87. Korel B. Computation of dynamic slices for unstructured programs. *IEEE Transactions on Software Engineering* 1997; **23**(1):17–34.
88. Mund GB, Mall R. An efficient interprocedural dynamic slicing method. *Journal of Systems and Software* 2006; **79**(6):791–806.
89. Zhang X, Gupta N, Gupta R. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering* 2007; **12**:143–160.
90. Danicic S, De Lucia A, Harman M. Building executable union slices using conditioned slicing. In Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC), 2004; 89–98.
91. Harman M, Hu L, Munro M, Zhang X, Danicic S, Daoudi M, Ouarbya L. An interprocedural amorphous slicer for WSL. In Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), 2002; 105–114.
92. Harrold MJ, Rothermel G, Sinha S. Computation of interprocedural control dependence. In Proceedings of the ACM International Symposium on Software Testing and Analysis, 1998; 11–21.
93. Lakhotia A. Improved Interprocedural Slicing Algorithm. In Technical Report: University of Southwestern Louisian, 1992.
94. Harman M, Hierons R, Fox C, Danicic S, Howroyd J. Pre/post conditioned slicing. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM), 2001b; 138–147.
95. Venkatesh GA. The semantic approach to program slicing. *SIGPLAN Not.* 1991a; **26**(6):107–119.
96. Harman M, Binkley D, Danicic S. Amorphous program slicing. *Journal of Systems and Software* 2003; **68**(1):45–64.
97. Ward M, Zedan H. Slicing as a program transformation. *ACM Transactions on Programming Languages and Systems* 2007; **29**(2):7.
98. Reps T, Horwitz S, Sagiv M, Rosay G. Speeding up slicing. *SIGSOFT software Engineering Notes* 1994; **19**(5):11–20.
99. Orso A, Sinha S, Jean H. Incremental slicing based on data-dependences types. In Proceedings of IEEE International Conference on Software Maintenance (ICSM), 2001; 158–167.
100. Ohata F, Hirose K, Fujii M, Inoue K. A slicing method for object-oriented programs using lightweight dynamic information. In Proceedings of the Eighth Asia-Pacific on Software Engineering Conference (APSEC). Macau, China, 2001; 273–283.

DOI: 10.1002/smr

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.

DOI: 10.1002/smr