A Very Efficient and Scalable Forward Static Slicing Approach

Conference Paper · October 2012

DOI: 10.1109/WCRE.2012.52

3 authors, including:

Hakam Alomari
Miami University
13 PUBLICATIONS 43 CITATIONS
SEE PROFILE

Michael Collard
University of Akron
62 PUBLICATIONS 1,525 CITATIONS
SEE PROFILE

A Very Efficient and Scalable Forward Static Slicing Approach

Hakam W. Alomari
Department of Computer Science
Kent State University
Kent, Ohio, USA
halomari@cs.kent.edu

Michael L. Collard
Department of Computer Science
The University of Akron
Akron, Ohio, USA
collard@uakron.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio, USA
jmaletic@kent.edu

***Abstract*—A highly efficient lightweight forward static slicing method is introduced. The method is implemented as a tool on top of srcML, an XML representation of source code. The approach does not compute the program dependence graph but instead dependency 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 a given variable. The tool produces the slice in this manner for all variables in a given system. The approach is highly scalable and can generate the slices for all variables of the Linux kernel in less than 13 minutes. Benchmark results are compared with the CodeSurfer slicing tool and the approach compares well with regards to accuracy of slices.**

***Keywords*- program slicing; software maintenance; impact analysis; minimal slice**

I. Introduction

Program slicing is a commonly used approach for understanding and detecting the impact of changes to software. The idea is quite simple, given a variable and the location of that variable in a program, tell me what other parts of the program are affected by this variable. The approach has been used successfully for many years for various maintenance tasks [1-3]. For example, slicing was used to help address the Y2K problem by identifying parts of a program that could be impacted by changes on date fields. The concept of program slicing was originally identified by Weiser [4, 5] 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 number of different slicing techniques and tools have been proposed and implemented. These techniques are broadly distinguished according to the type of slices such as: static versus dynamic [2, 3], closure versus executable [3], inter-procedural versus intra-procedural [6, 7], and forward versus backward [3, 8].

The calculation of a program slice is, with few exceptions, based on the notion of a Program Dependence Graph (PDG) [9] or one of its variants, e.g., a System Dependence Graph (SDG) [10]. 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.

The work presented here 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 [11, 12] format for source code. The srcML format provides direct access to abstract syntactic information to support static analysis. While this lightweight approach will typically never match the accuracy of generating a PDG/SDG and doing full pointer analysis, etc. it provides a fairly accurate picture of a program slice in an extremely short time comparatively for large systems (i.e., we found up to four 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 deep more expensive analysis of the system is warranted. Lastly, we feel a fast slicing approach could 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* and conducted a comparison study to the *CodeSurfer* [13] tool from GrammaTech Inc. The results show that the slices produced by our approach are very reasonable with respect to accuracy. To demonstrate the scalability, we applied the tool to 17 years of versions of the Linux kernel and present the results.

The remainder of the paper is organized as follows. Section II describes our slicing approach and implementation. A comparison with the *CodeSurfer* tool is presented in Section III and IV. Section V demonstrates the scalability of the approach. Related work is then presented followed by conclusions.

II. Lightweight Forward Static Slicing

Forward static program slicing [14] 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 points in the forward control flow affected by the computation at *p*. Here we use the initial variable declaration as the starting point. Specifically, the approach taken here computes a forward, static, non-executable, inter-procedural program slice for each variable in a system. 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 (just a variable).

The approach relies on an underlying XML representation of the source code, namely srcML [11, 12]. 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, *src2srcml* and *srcml2src*, that 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 [11, 12], source-code transformation [11], and pattern matching of complex code [15]. Before presenting the static forward algorithms, we define our slicing criterion, and then show how a slice is computed using this approach.

A. Extended 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 [1, 16] 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 [17]. The example used by Gallagher does static backward slicing only. The decomposition slice on variable *v* is the union of 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 which is not part of the program output may still be used as a decomposition criterion.

For our static forward decomposition slicing we modify Gallagher's definition by slicing at the set of statements that assign (or input) to the given variable *v*. 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, i.e., 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 impacted statements 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* on statement s_6, as there is an assignment that redefines *c* in statement s_5. 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 point of view 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. The quality of the decomposition slice is affected by the quality of the slice, since as shown the definition of decomposition slice is independent of any underlying slicing technique. Once a slice is obtained using any slicing algorithm, a decomposition slice may be computed [1, 16].

```
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 [1].

B. 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 rely fully on pre-computed data and control dependencies since they can require costly analysis, e.g., constructing the *def-use* chains in the existence of pointers. Instead, this is calculated as needed on the fly for the slicing variable 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.

By modifying the slicing criterion to (*f, m*), our approach can retrieve the slices for all the variables inside a given function. 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 the 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 as declared in order 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

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 consists of two slice profiles, one for each of the variables *sum* and *i*. The *@index* represents the count of variables in the function. This way we can deal with variables of the same name within the same scope. The slice profiles are computed by examining 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}*
5: *slines(sum)= {2, 5}*; *slines(i)= {3, 4, 5}, dvars(i)= {sum}*
6: *slines(sum)= {2, 5}*; *slines(i)= {3, 4, 5, 6}, dvars(i)= {sum}*
8: *slines(sum)= {2, 5, 8}*; *slines(i)= {3, 4, 5, 6}, dvars(i)= {sum}*
9: *slines(sum)= {2, 5, 8}*; *slines(i)= {3, 4, 5, 6, 9}, dvars(i)= {sum}*

These are the slice profiles for each variable and the complete slice is then computed by 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, ... ,def(v)-1}*. Thus, since *sum* is data dependent on *i*, the complete slice for *i = slines(i)* ∪ *slines(sum) - {1, 2}*. This comes out to *{3, 4, 5, 6, 8, 9}*. This final computation can be done for all variables via a single pass through the dictionary.

(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.

We now more formally define our slicing criterion and how a slice is computed using this criterion.

Definition: A *slicing criterion* is of the form (*f, m, v*), (*f, m*), (*f*), and (*F, M, V*), where $F = \{f_1, f_2, \ldots, f_j\}$ is the finite set of files in the system, $M = \{m_1, m_2, \ldots, m_y\}$ 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 $m \in M$.

Definition: A *forward decomposition slice* on variable *v* with respect to the slicing criterion (*f, m, v*) is of the form:

$$slice\,(f, m, v) = \bigcup_{n \in N} slice\,(v, n)\ ,$$

where *N* is a set of statements that assigns to the variable *v*. For the slicing criterions (*f, m*), (*f*), and (*F, M, V*) the slices consist of the union of static forward slices as follows:

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

$$slice\,(f) = \bigcup_{i=1}^{y} slice\,(f, m_i)$$

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

C. 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, and pointer aliasing and 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 to a variable *c* will include the slice profile of variable *a*. Detecting such statements is important due to the fact that 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, i.e., 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, i.e., 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 3, is the main algorithm for intra-procedural slicing. The same procedure is used when slicing over the called functions from the slice profile of the slicing variable.

```
Input: Slicing Criterion = (F, M, V)
Output: System Dictionary – a table of slice profiles for all variables
Sp: Slice Profile ∀v ∈ V
VL: Set of local variables ∀m ∈ M - elements are the pairs (v_i, Sp (v_i))
VG: Set of global variables ∀f ∈ F - elements are the pairs (v_i, Sp (v_i))
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:    if stmt_i is a parameter-list then // Detection Step
4:      for each v ∈ stmt_i do
5:        if (VL = Ø) □ (v ∉ VL ) then
6:            Sp (v).@index ← {index of v}
7:            Sp (v).slines ← {stmt_i line number}
8:            VL ← (v, Sp (v))
9:        else if v ∈ VL then // update Sp(v)
10:           Sp (v).slines ← Sp (v).slines □ {stmt_i line number}
11:           VL ← (v, Sp (v))
12:       end if
13:     end for
14:   else if stmt_i is a declaration statement then
        // Detection Step - repeat the lines 4-13
15:   else if stmt_i is an expression statement then // Verification Step
16:     found ← false
17:     for each v ∈ stmt_i , st. v ∈ VL do
18:       found ← true
19:       if v is a (r-value) , st. v (r-value) ≠ v (l-value) then
20:          Sp (v).dvariables ← { v (l-value) }
21:          Sp (v).slines ← Sp (v).slines □ {stmt_i line number}
22:       else if v (l-value) is a pointer alias of v (r-value) then
23:          Sp (v).pointers ← { v (l-value) }
24:          Sp (v).slines ← Sp (v).slines □ {stmt_i line number}
25:       end if
26:       VL ← (v, Sp (v))
27:     end for
28:     if !(found) then /* for global variable, repeat lines 17 – 27
    using the VG set instead of VL set */
29:     end if
30:   else if stmt_i is a function call then
        // extract all the arguments, with their indices
31:     for each v ∈ argument-list do
32:       if v ∈ VL then
33:         Sp (v).slines ← Sp (v).slines □ {stmt_i line number}
34:         Sp (v).cfunctions ← Sp (v).cfunctions □ {function name}
35:         Sp (v).@index ← {index of v}
36:         VL ← (v, Sp (v))
37:       else if v ∈ VG then
            // repeat lines 33 – 36 using the set VG
38:       end if
39:     end for
40:   end if
41: end for
42: until end of m
end
```

Figure 3. Algorithm to compute the slice profile for each variable in a system. The final forward static slice is computed after including dependencies found in each slice profile of the system dictionary.

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 (*see line 14*) and parameter statements (*see line 3*), and implements the uses verification by analyzing expression statements (*see line 15*).

The global declaration statements (algorithm omitted for brevity) are analyzed in the same way as the definition detection in the algorithm *ComputeSliceProfile*. 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 above 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 done 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 20 and 21*). The set *cfunctions* for the effective variable is filled while processing function calls (*see line 30*), making it possible to compute the closure across the system.

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, since 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. Inter-procedural slicing is accomplished by mapping the indices of the variables in the argument list (*see line 6*) to their corresponding indices detected in the calling statements (*see line 35*). From this we recover all statements that are included in the slice profile of the parameters.

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 indexes to other relevant lines 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 done 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.

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 variable to other relevant lines for the forward slice of the variable.

We implemented our slicing approach as a tool called *srcSlice*. It uses srcML as input and the Qt XML parser class *QXmlStreamReader*. The main advantage of using the pull approach is the ability to construct a recursive parser making traversing the code quite simple. In addition, this approach is memory conservative since there is no need to store the entire srcML document tree in memory, as in a DOM approach.

III. Comparative Study

To assess our approach and the *srcSlice* tool we conducted a comparative study with the academic version 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 to a standard. The second objective is to demonstrate that our approach is highly scalable and efficient. Together these two objectives lead to three primary research questions this study tries to address:

RQ1: *Does srcSlice produce accurate slices?*
RQ2: *Is there an unacceptable level of inaccuracy?*
RQ3: *Is srcSlice highly scalable and efficient?*

The question of what is a perfectly accurate slice is somewhat open to interpretation [5]. This is the case for many results of static analysis. For example, an empirical study of static call graph extractors by Murphy et al [18] 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 [19] 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 by comparing our results to that of a commonly used existing approach/tool, will minimally give us a baseline with respect to the accuracy of the results. That is, if our results are similar to that of *CodeSurfer*'s we feel confident that it produces reasonably correct slices. We first ran both slicers on a number of small programs (aka *feature benchmarks*). These results are used to determine the correctness of our results and help explain slicing results in larger programs. Second, we ran both slicers on larger open-source programs (aka *performance benchmarks*) of varying size that worked with both slicers. These results are used to illustrate the first and the second research questions and partially address the third research question. Finally, we ran *srcSlice* on the Linux kernel to answer the third research question.

A. 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 [13]. It is based on the slicing work done at the University of Wisconsin surveyed in [20]. 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 dependences: 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 above 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 control-flow graph 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 [21]. 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 the free academic version of *CodeSurfer* used in this study is unable to slice programs larger than 200 KLOC [22]. Thus, all of the benchmark cases used here are under 200 KLOC.

For our study, the Highest setting for *CodeSurfer* was used to provide the most precise results. We expect to obtain a *safe* or *conservative* (definitions introduced in next section) slicing results at the expense of longer build times. We note that we are not explicitly comparing times of the two tools but the accuracy, hence using the high setting for *CodeSurfer*. However, we found that even on the lowest setting *CodeSurfer* runtime performance to be much slower than *srcSlice*. Also, since the tool can be used for other tasks than computing slices, all features except the slicing results were turned off.

B. 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 method. In addition, we want to determine if the results obtained by *srcSlice* are comparable to *CodeSurfer* in terms of accuracy. However, since 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 correctness, size of the results, time and space efficiency, and the limitations of both tools. Finally, we investigate most of the language features supported, e.g., is the tool able to handle pointers, call by reference, etc.

First we examine the slice size and quality. The correctness of the slice relates directly to its purpose [19]. 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) [23]. The slice size (denoted by *SZ*) for both tools is measured by the SLOC, number of statement lines of code. The best slice for a given slicing criterion should be the smallest correct slice. 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, i.e., 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 [21], i.e., 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 [19]. The problem of determining the minimal slice is not in general decidable [5, 21, 24]. 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, i.e., 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 [25] observed that after studying 43 programs with ~1 MLOC that the most precise program slicer had an average slice size equal to 30% of the original program. He 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 the both tools, called the *intersected slice* following the same approach used in the qualitative study of Bent et al [21]. That study compared a dataflow-slicing approach (*Sprite*) 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 must exist that does not include the differing statements, i.e., *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 *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 the slicing then performed 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:

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

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 [25-28]. However, Bent et al [21] 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 since as mentioned early; 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 since 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 11 seconds for the largest program, *cvs-1.12.10*, in the performance benchmarks.

IV. Study Results

We now present data comparing slices from both tools. 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 dependency situations they included such things 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. Due to space limitations we omit the details of this comparison. However, the benchmark programs are posted on our website (www.sdml.info/downloads/slice) along with the full results of this comparison.

In short we found that *srcSlice* produced equal or smaller slices than those produced by *CodeSurfer* in all cases but one.

TABLE I. PERFORMANCE BENCHMARKS RESULTS AND COMPARISON OF THE CODESURFER AND SRCSLICE, SLICING TIME MEASURED IN SECONDS, SLICE SIZE MEASURED IN NUMBER OF STATEMENTS, (%) COLUMNS ARE THE SLICE SIZE RELATIVE TO LOC, (F) = NUMBER OF FILES, (M) = NUMBER OF METHODS, LOC = LINES OF CODE.

Program	Size			CodeSurfer				srcSlice			
	LOC	**F**	**M**	**Slices Taken**	**Slicing Time**	**Slice Size**	**%**	**Slices Taken**	**Slicing Time**	**Slice Size**	**%**
ed-1.2	3087	10	126	4438	21	1782	57.7	516	4	1146	37.1
ed-1.6	3260	10	128	4527	19	1863	57.1	560	4	1241	38.1
which-2.20	3586	14	51	1429	15	736	20.5	203	4	465	13.0
wdiff-0.5	3874	13	56	1097	11	652	16.8	136	4	561	14.5
barcode-0.98	5205	18	74	4590	29	2177	41.8	451	4	1647	31.6
acct-6.5	8749	27	127	4983	47	2510	28.7	868	4	2193	25.1
enscript-1.4.0	18162	52	180	9456	71	5916	32.6	1139	6	3073	16.9
make-3.82	36397	58	474	17012	807	9446	26.0	3459	8	10703	29.4
enscript-1.6.5.2	56491	107	488	20234	184	12907	22.8	3043	11	10119	17.9
enscript-1.6.5	56494	107	488	20252	184	12913	22.9	3050	11	10126	17.9
enscript-1.6.5.1	56494	107	488	20252	185	12913	22.9	3050	10	10126	17.9
a2ps-4.10.4	57052	188	1104	24493	393	14249	25.0	4119	14	9035	15.8
findutils-4.4.2	72384	314	1141	23641	215	13689	18.9	7229	18	17298	23.9
radius-1.0	82029	196	1719	38487	335	19218	23.4	7822	16	18287	22.3
dico-2.2	119592	332	2504	52297	1763	28639	23.9	13012	22	30703	25.7
cvs-1.12.10	144278	340	2027	74328	286328	40869	28.3	10116	26	30310	21.0
Total	**727134**	**1893**	**11175**	**321516**	**290584**	**180479**		**58773**	**173**	**157033**	
Average	**45447**	**118**	**698**	**20095**	**19372**	**11280**	**29.3**	**3673**	**10**	**9815**	**23.0**

The average forward slice contained 45.2% of the program source using *CodeSurfer* and 34.1% using *srcSlice*. The slices produced using *srcSlice* were manually checked (by the authors) and found to be correct when compared to *CodeSurfer*. The difference in the results was because *CodeSurfer* retrieved some unrelated statements; such as statements inside the blocks of *for* and *while* predicates and standard libraries. *CodeSurfer* highlights statements that are not only semantically related to the slicing criterion but also syntactically related to the executable slice [21]. For example, *CodeSurfer* includes all relevant statements that modify or determine control flow statement in the *else* part of an *if* statement whose body was not in the slice. For the settings chosen, *CodeSurfer* provides a correct slice with regards to data and control dependencies. The results also show that *srcSlice* produced accurate slices when compared to *CodeSurfer*. We note again that the settings used for *CodeSurfer* were to enhance accuracy and not performance.

A. Performance Benchmarks

This initial comparison now leads us to a more comprehensive comparison. Table I shows the results of the performance benchmark 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 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 I, the number of slices for both tools and the number of lines of code do not match, since in the PDG-based slicing approach one line of code could be represented by multiple vertices [27]. In contrast, our slicing approach has variable granularity. Thus, one line of code may have several variables.

This performance study considers just over 700 KLOC of C code from 16 open-source programs that range in size from approximately 3 to almost 150 KLOC. Table I shows the 16 programs along with the results obtained by *srcSlice* and *CodeSurfer*. 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 I appear in Binkley's studies [25, 27] although they may be different versions.

Each row in the table is a benchmark we used for the comparison. The slice was over all possible slicing criterions in each program. As seen in the last row of the table, the average slice size using both tools over all 16 programs included between 23.0% and 29.3% 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 13.0% for *which-2.2* to 38.1% for *ed-1.6*. In general, the slice size produced by *srcSlice* is smaller than the one produced by *CodeSurfer*; however this is not the case in 3 out of 16 cases, i.e., *make-3.82*, *findutils-4.4.2*, and *dico-2.2*. The intersected slice results give us several hints why this is so.

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 seconds, with the highest settings for precision used and are up to 19,000 times slower than *srcSlice's*. However, *CodeSurfer* does produce a larger number of slices (~5.5 times more), which accounts for part of the slow down. By excluding the larger program (*cvs-1.12.10*), *CodeSurfer's* slicing time reduces to ~126 times that of *srcSlice*.

B. Slice Intersection Comparison

We use the intersected slice as a measure of the quality of calculated slice. As explained in Section III B, we feel that by intersecting our results to that of a *CodeSurfer* will minimally give us a baseline with respect to accuracy of the results. That is, if our results are closer to that of the intersected slice we feel confident that it produces reasonably correct slices. The slices of selected files are generated using all possible slicing criterions with both tools, and then the intersection between corresponding slices is taken. The intersected slices are generated for two performance benchmarks from Table I *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 program *enscript-1.6.5* are presented in Table II. 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[1]. 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 relative to the *srcSlice's* and *CodeSurfer's* sizes, on average are equal to 91.1% and 52.8%, with a maximum of 100% and 76.3%,.

TABLE II. INTERSECTED SLICE OVER 13 FILES FROM ENSCRIPT-1.6.5, WHERE (%) IN THE CODESURFER (CS) AND SRCSLICE (SS) COLUMNS IS THE SLICE SIZE RELATIVE TO LOC, (%) IN THE INTERSECTION COLUMN IS THE INTERSECTED SLICE RELATIVE TO BOTH TOOLS SLICE SIZE, (SM) IS THE RELATIVE SAFETY MARGIN FOR A SLICE.

enscript-1.6.5	Size		Slice Size						Intersection		
	LOC	SLOC	CodeSurfer (CS)			srcSlice (sS)			Lines	CS	sS
File Name			**Lines**	**%**	**SM**	**Lines**	**%**	**SM**		**%**	**%**
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	**16070**	**10893**	**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 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 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 regards to large array initialization might be an issue. Because the *CodeSurfer* algorithm treats each element of an array as a distinct variable, [13], 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 [3]. 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 **Error! Reference source not found.**. 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*.

V. 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 seconds and produced a system dictionary of 1,934,557 variables. The data used in this section originates from slicing 974 versions of the Linux kernel that have been released over the last 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, since 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 seconds. Version 2.6.37.1 with ~13 MLOC takes approximately 13 minutes.

We now examine the slice size of our results, as this is considered to be a crucial issue [25], 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 will improve the quality of the slice. Since the average slice size relative to LOC is 46.0%, we feel that our results are in a reasonable margin, based on the

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

work by Binkley et al [25, 26] and the results obtained in the previous sections. Furthermore, the results given in Figure 5 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.

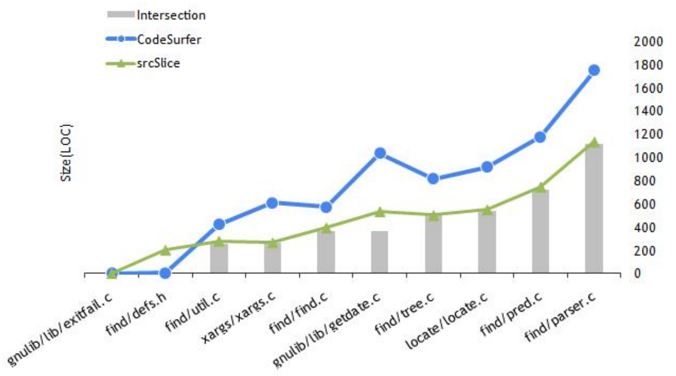

Figure 4. Comparison of CodeSurfer, srcSlice, and slice intersection over 10 files from the program findutils-4.4.2 ordered by intersection size. Except for a single file, srcSlice was much closer to the slice intersection.

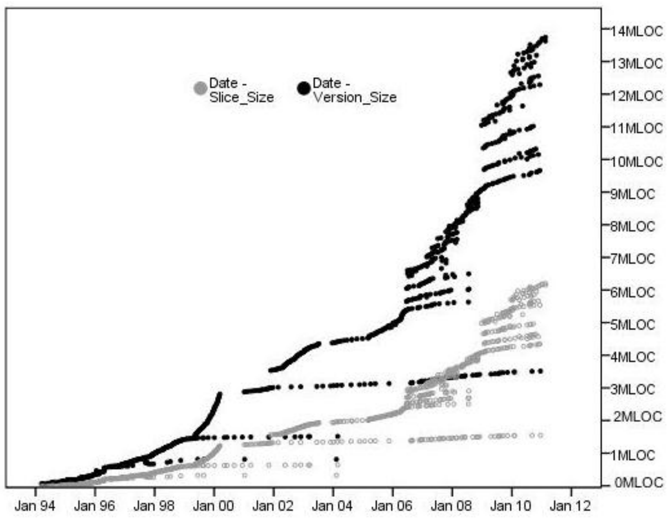

Figure 5. A comparison of the size of the srcSlice's slices to the size of the Linux versions measured in MLOC, the x-axis is the version date.

VI. RELATED WORK

Here we focus on slicing approaches directly related to our approach. We refer readers interested in PDG-based slicing approaches to Tip's and Xu's surveys [2] and [3]. Gallagher et al [1] proposed the definition of decomposition slicing as a maintenance aid in order to capture all the computation related to a given slicing variable. His objective was to define and isolate the parts of the code that are affected by the proposed change or modification of the slicing variable so he could eliminate the need for regression testing. The decomposition-slicing definition is used by Tonella [29] to construct a concept lattice of decomposition slices. The idea was to combine the decomposition slice graph produced by Gallagher and the lattice program representation model. The concept lattice of decomposition slices is used to support software maintenance, by providing relevant information about the computations performed by a program and the related dependences.

A lightweight-slicing approach for object-oriented programs using dynamic and static analysis, called dependence-cache slicing, is proposed in [30]. This approach is based on dynamic data dependence analysis and static control dependence analysis. The slice construction process starts by locating the defined variables, and then extracts the data and control dependencies between statements that include those variables. These dependencies are used to construct the PDG that is traversed backwards for a user given slicing variable. 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 since 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.

Since our approach is scalable in term of time and program size as shown in Section V, the need was to evaluate the correctness of our results. Our evaluation criteria was based on the study proposed by Hoffner [19] in which he discussed several possible aspects to evaluate the performance of proposed slicing approaches. These aspects are the slice size compared to the original size, and the time and space complexities. The author compares a set of dynamic slicing tools, i.e., *Kamkar's* and *Spyder*, with other static tools including the *WPIS*, *FOCUS* and *Schatz's* tool. Only three of the tools support inter-procedural slicing, which are *Kamkar's*, *Spyder* and *WPIS*. In addition, evaluation criterions were established for comparing these different tools. In his evaluation, the 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 approach is distinguished from this related work in multiple ways. The method used is not PDG based. There is no graph to traverse or data-flow equations to be solved. Only on-the-fly information is retrieved 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 locals and globals. As new variables are encountered they are added to the slice profile. Our approach supports both system evaluation and comprehension by allowing the user to investigate the program by using the slices at different levels of granularities (e.g., variable, function, file, and system).

VII. CONCLUSIONS AND FUTURE WORK

A method for efficient and scalable slicing was presented and compared to an existing tool. The results demonstrated that the approach produces fairly accurate slices as compared to an existing tool and is highly scalable. The limitations of the approach are related to deep aliasing and certain array indexing. The *srcSlice* tool was shown to work on a variety of C programs and language features. The approach uses the srcML format and toolkit. 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. Additionally, this feature is very useful for adaptive maintenance tasks involving new API/libraries.

In practice we see the usefulness of this and similar lightweight approaches being as a quick-check mechanism rather than a replacement for more heavyweight (and hopefully more accurate) slicing tools. That is, developers can use this approach to judge if it is prudent to expend the time and money to run a more rigorous analysis on a large software system. Moreover, the ability to slice multiple versions of large programs in a very short amount of time also opens up new avenues of research. We can investigate how system slices change over the entire history of a large software system, and how slices reflect different types of changes occurring in a system possibly identifying refactoring changes [23, 32]. With current tools this is impractical. In the future, we plan to also investigate metrics based on program slicing in the context of coupling and cohesion. We also plan to make the *srcSlice* tool part of our srcML toolkit. Currently, *srcSlice* also works for C++ programs but more evaluation is needed.

VIII. REFERENCES

[1] Gallagher, K.B. and J.R. Lyle, *Using Program Slicing in Software Maintenance.* IEEE TSE, 1991. **17**(8): pp. 751-761.

[2] Tip, F., *A Survey of Program Slicing Techniques.* Journal of Programming Language, 1995. **3**: pp. 121-189.

[3] Xu, B., J. Qian, X. Zhang, Z. Wu, and L. Chen, *A Brief Survey of Program Slicing.* ACM Software Engineering Notes, 2005. **30**(2): pp. 1-36.

[4] Weiser, M., *Program Slicing*, in *Proceedings of the International Conference on Software Engineering (ICSE'81)*. 1981: pp. 439-449.

[5] Weiser, M., *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD Thesis. 1979, University of Michigan: Ann Arbor, MI, USA.

[6] Gallagher, K. and D.W. Binkley, *Program Slicing*, in *Frontiers of Software Maintenance (FoSM'08)*. 2008: Beijing, China. pp. 58-67.

[7] Horwitz, S., T. Reps, and D. Binkley, *Interprocedural Slicing using Dependence Graphs.* ACM SIGPLAN Notices, 1988. **23**(7): pp. 35-46.

[8] Kumar, S. and S. Horwitz, *Better Slicing of Programs with Jumps and Switches*, in *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE'02)*. 2002. pp. 96-112.

[9] Ottenstein, K.J. and L.M. Ottenstein, *The Program Dependence Graph in a Software Development Environment.* ACM SIGSOFT Software Engineering Notes, 1984. **9**(3): pp. 177-184.

[10] Liang, D. and M.J. Harrold, *Slicing Objects Using System Dependence Graphs*, in *Proceedings of the International Conference on Software Maintenance (ICSM'98)*. 1998: Bethesda, Maryland, USA. pp. 358-367.

[11] Collard, M.L., J.I. Maletic, and B.P. Robinson, *A Lightweight Transformational Approach to Support Large Scale Adaptive Changes*, in *Proceedings of the International Conference on Software Maintenance (ICSM'10)*. 2010: Timisoara, Romania. pp. 1-10.

[12] Collard, M.L., M. Decker, and J.I. Maletic. *Lightweight Transformation and Fact Extraction with the srcML Toolkit*. in *Proceedings of International Source Code Analysis and Manipulation (SCAM'11)*. 2011: pp. 173-184.

[13] *CodeSurfer Grammatech Inc*, www.grammatech.com/products/.

[14] Bergeretti, J.-F. and B.A. Carre', *Information-Flow and Data-Flow Analysis of While-Programs.* ACM TOPLAS, 1985. **7**(1): pp. 37-61.

[15] Dragan, N., M.L. Collard, and J.I. Maletic. *Reverse Engineering Method Stereotypes*. in *Proceedings of the International Conference on Software Maintenance (ICSM'06)*. 2006: Philadelphia, Pennsylvania, USA.

[16] Gallagher, K.B., *Some Notes on Interprocedural Program Slicing*, in *Proceedings of International Source Code Analysis and Manipulation (SCAM '04)*. 2004: Chicago, IL, USA. pp. 36-42.

[17] Weiser, M., *Program Slicing.* IEEE TSE, 1984. **10**(4): pp. 352-357.

[18] Murphy, G.C. and D. Notkin, *An Empirical Study of Static Call Graph Extractors.* ACM TOSEM, 1998. **7**(2): pp. 158-191.

[19] Hoffner, T., *Evaluation and Comparison of Program Slicing Tools.* T.R. LiTH-IDA-R-95-01, 1995, Department of Computer and Information Science, Linkping University: Sweden.

[20] Horwitz, S. and T. Reps, *The Use of Program Dependence Graphs in Software Engineering*, in *Proceedings of International Conference on Software Engineering (ICSE'92)*. 1992: Melbourne, Australia. pp. 392-411.

[21] Bent, L., D.C. Atkinson, and W.G. Griswold, *A Qualitative Study of Two Whole-Program Slicers for C.* T.R. CS20000643, 2000, University of California at San Diego. A preliminary version appeared at FSE '00.

[22] *Wisconsin Program-Slicing Project*, www.cs.wisc.edu/wpis/html/#codesurfer.

[23] Pan, K., S. Kim, and J. Whitehead, E., James, *Bug Classification Using Program Slicing Metrics*, in *Proceedings of International Source Code Analysis and Manipulation (SCAM'06)*. 2006: PA, USA. pp. 31-42.

[24] Danicic, S., C. Fox, M. Harman, R. Hierons, J. Howroyd, and M.R. Laurence, *Static Program Slicing Algorithms are Minimal for Free Liberal Program Schemas.* Compututer Journal, 2005. **48**(6): pp. 737-748.

[25] Binkley, D., N. Gold, and M. Harman, *An Empirical Study of Static Program Slice Size.* ACM Transactions on Software Engineering and Methodology (TOSEM'07), 2007. **16**(2).

[26] Binkley, D., N. Gold, M. Harman, Z. Li, and K. Mahdavi, *An Empirical Study of Executable Concept Slice Size*, in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*. 2006: Italy. pp. 103-114.

[27] Binkley, D. and M. Harman, *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'03)*. 2003: Amsterdam, The Netherlands. pp. 44-53.

[28] Binkley, D., M. Harman, Y. Hassoun, S. Islam, and Z. Li, *Assessing the Impact of Global Variables on Program Dependence and Dependence Clusters.* Journal of Software Systems, 2010. **83**(1): pp. 96-107.

[29] Tonella, P., *Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis.* IEEE Transactions on Software Engineering (TSE'03), 2003. **29**(6): pp. 495-509.

[30] Ohata, F., K. Hirose, M. Fujii, and K. Inoue, *A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information*, in *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference (APSEC'01)*. 2001: Macau, China. pp. 273-283.

[31] Cordy, J.R., N. Eliot, and M.G. Robertson, *TuringTool: A User Interface to Aid in the Software Maintenance Task.* IEEE TSE, 1990. **16**(3): pp. 294-301.

[32] Zhang, X., N. Gupta, and R. Gupta, *A Study of Effectiveness of Dynamic Slicing in Locating Real Faults.* ESE, 2007. **12**(2): pp. 143-160.