Global Dynamic Slicing for the C Language

Árpád Beszédes

University of Szeged, Department of Software Engineering
Árpád tér 2, H-6720 Szeged, Hungary
beszedes@inf.u-szeged.hu

Abstract: In dynamic program slicing, program subsets are computed that represent the set of dependences that occur for specific program executions and can be associated with a program point of interest called the slicing criterion. Traditionally, dynamic dependence graphs are used as a preprocessing step before the actual slices are computed, but this approach is not scalable. We follow the approach of processing the execution trace and, using local definition-use information, follow the dependence chains "on the fly" without actually building the dynamic dependence graph, but we retain specialized data structures. Here, we present in detail the practical modifications of our global dynamic slicing algorithm, which are needed to apply it to programs written in the C language.

Keywords: program slicing; dynamic slicing; program analysis; program dependence; C

1 Introduction

Program slicing [1, 2] is a program analysis technique that is used to help solve various software engineering problems. A slice of a program is the program's subset which consists of only those statements that directly or indirectly affect the value of a variable occurrence (known as the slicing criterion). This form of slicing is referred to as backward slicing. In contrast, forward slicing involves looking for forward dependences; those statements that may be affected by a specific program point. If the dependence set is determined in such a way that it reflects the dependences for all possible executions, we call it a static slice. However, if only a specific program execution is investigated, it is called a dynamic slice. In our study we will focus on backward dynamic slicing.

Dynamic program slicing [3] has a certain advantage in some applications; namely, the dynamic slices are significantly smaller than their static counterparts. For instance, when debugging we seek the possible cause(s) of an error that was observed at a specific program point and for a specific run. The more precisely this set of causes is defined, the more effective the debugging should be.

A common approach to dynamic slicing is based on computing the *dynamic dependences* among the program elements. The method by Agrawal and Horgan [3] uses a graph representation called the Dynamic Dependence Graph (DDG), which includes a distinct vertex for each occurrence of a statement in the execution history (the list of statements executed), and the edges correspond to the dynamically occurring dependences among them. Based on this graph, the computation of a dynamic slice means finding all the reachable vertices starting from the slicing criterion. The DDG-based method can be used to compute dynamic slices in a general way, since it performs a full preprocessing [4] before the actual slicing. When building the graph in advance, the user has the possibility of computing different slices starting from different program points (the criteria) and going in different directions (forwards or backwards). Since the computation and storage for such a graph is expensive more specialized approaches that take into account the desired slicing scenario should be considered.

In previous studies [5, 6, 7], we devised new efficient dynamic slicing methods that were based on dynamic dependences, but did not require full preprocessing and the building of huge representations like the DDG graph. We also process the execution history and some elements of the complexity of our approach are related to the length of the execution as well; but other, more specialized data structures and algorithms are applied in order to improve the overall efficiency. One of the results is a backward slicing algorithm [6, 7] that computes all the possible dynamic slices globally, with only one pass through the execution history. This method significantly differs from the previously published slicing algorithms, and it is believed to be applicable for real-size programs and executions. We presented some details of the algorithm in different contexts; for C [6, 8] and for Java [9] programs, and for different applications [7, 10].

In this paper, we provide details on the implementation of the global algorithm for backward slicing for C programs. This makes its implementation possible in virtually any context and platform for C.

2 Previous Results and Related Work

Program slicing has a large literature and many different approaches have been devised. Surveys can be found at various places, e.g. [2, 11]. While the practical static slicing methods are mostly based on the PDG-based algorithm by Horwitz *et al.* [12], there are several, quite different approaches to dynamic slicing. One usual categorization of the dynamic slicing methods comes from asking whether the program subset produced (the slice) is an executable program or not. Executable slices are needed for certain applications, but they are less accurate.

Actual implementations of dynamic slicing algorithms were mentioned in very few publications, such as in [13, 14]. However, these implementations did not prove to be suitable for real-life applications. In a study, Venkatesh experimented with different algorithms and provided experimental data [14]. In this experiment, four kinds of slicing algorithms were implemented for the C language, including the dependence-based approach by Agrawal and Horgan [3] and the executable slicing method by Korel and Laski [15]. Unfortunately, no details were given on the design and functionality or the special features of the implementation for C.

In the DDG-based method by Agrawal and Horgan [3], the size of the DDGs may be huge. In fact, it is not bounded by the program dimensions, but it correlates with the execution length. In their study, Agrawal and Horgan therefore proposed a reduced DDG method, where the size of the reduced graphs was bounded by the number of different dynamic slices. Alas, even this reduced DDG may be very large for some programs. In [16], Zhang *et al.* elaborate on the problems of existing dynamic slicing algorithms concerning their computation and space complexity. They claim that most accurate (they use the term "precise") algorithms are significantly less efficient than the approximate methods, which in turn produce inaccurate dynamic slices. This inefficiency may be attributed to two factors: either the execution trace is completely processed before the actual slicing algorithm is performed (referred to as full preprocessing) or the slicing algorithm is invoked on demand, processing the trace from the start for each slicing request (referred to as no preprocessing). Our global and demand-driven algorithms [5] correspond to the first and second cases, respectively. In both approaches the authors gave their own implementations based on dependence graphs. To reduce the overheads of each approach they proposed a combined algorithm called limited preprocessing, where the execution trace is augmented with summary information to allow a faster traversal when the slice is computed.

2.1 Our Global Dynamic Slicing Algorithm

In a previous study [5], we investigated practical ways of computing the dynamic slices based on dynamic dependences, but without requiring costly global preprocessing. We proposed alternative methods based on the same dynamic dependences, but instead of DDG graphs specific data structures were used for each algorithm. These structures are different depending on the slicing scenario, and – having specific applications – some of the algorithms are more efficient in terms of storage, while others have improved runtime efficiency. The different slicing scenarios that we investigated are global vs. demand-driven slicing and computing backward vs. forward slices. One favorable property of these algorithms is that they are able to compute the same dynamic slices as the original DDG-based method. It turns out that the slices can be produced by traversing the execution history either in a forward or in a backward way, and that some processing directions fit better in one slicing scenario than in another. This gives

eight possibilities, some of which give useful algorithms, while others prove unfeasible. In our paper, we elaborate the topic by providing details on handling real C language constructs for the global algorithm.

Our approach for computing dynamic slices differs significantly from the previous methods. We designed the slicing algorithms so that they can be effectively implemented and used in practice. Hence, we tried to minimize the amount of information that must be computed and stored during the computations [5, 6, 7]. In our algorithms, we track the data and control dependences among the program instructions that arise dynamically during execution. The algorithm works on the trace of the execution, which is produced using a statically instrumented version of the program. The trace includes all the necessary information about the runtime behaviour of the program.[1] For producing the required slicing results, the algorithm relies on statically computed information from the code as well. The global algorithm (also referred to as the *forward* algorithm) starts at the beginning of the trace with the first executed instruction and propagates the dynamic dependences in parallel with the execution, and eventually provides the required slices for all the possible dynamic criteria (for all the variables). Evidently, this approach has its benefits and drawbacks, but the other algorithms presented in [5] provide feasible alternatives. For details of the conceptual algorithm with examples, please see the articles cited above.

3 Global Dynamic Slicing for C Programs

In other studies [5, 6, 7], we presented conceptual algorithms for dynamic slicing. The concepts were introduced for programs in which only scalar variables were used and without interprocedurality. The application of the conceptual algorithm to C programs gives rise to several problems. In our study, the handling of various language constructs were addressed in the following way:

1) All computations are performed on *memory locations* instead of handling scalar variables, pointers and other more complex objects differently. This approach enables an easy and uniform handling of pointers, pointer dereferences, arrays and structures by transforming them to the actual memory locations. In our approach, the dependences for a pointer dereference will include the dependences of the pointer itself and the dereferenced memory location as well. Also, accesses to union members and C bitfields are treated as dependences for the whole data structure (struct and array members are handled individually). Slicing on memory

[1] Execution tracing is used in other areas of software engineering as well; it has recently been proposed to extend software product quality frameworks [18].

locations is a feasible approach since all the dynamic information on the actual storage of objects is available.

2) Since each C statement (and expression with side-effects) may imply the definition of more than one object, a *definition-use list* is defined for each executable instruction, rather than a single definition-use pair as we have with the conceptual algorithm [5, 6, 7] (a *definition-use pair* or *def-use pair* consists of a variable name that is defined at the instruction and a set of variables that is used in the instruction for computing the value of the defined variable). This list is essentially a sequence of def-use pairs that all occur in an instruction (see below for a complete definition).

3) All slicing criteria and slicing results are given for line numbers in the original source file. However, since the computations are made on memory locations and for (possibly multiple) objects defined for one statement, the necessary mappings must be made.

4) Interprocedural dependences that arise across function calls can be handled relatively easily by adopting the memory slicing approach, since each memory address can be viewed as a "global variable." The execution history will contain each realized function call, and the order of the instructions executed will also be known. We only have to handle the actual arguments as special local variables and the return value as a special variable defined at the call site.

5) Local variables are also handled by using their addresses on the actual call stack frame. We only need to track the block scopes dynamically for lookup purposes. The handling of globals is also simple due to using their addresses for computation (which are fixed for the whole execution of the program).

6) The unstructured control transfers (*goto* and other jump statements) are handled by adding all the possible control dependences to the def-use representation (for a block-based language as in our conceptual description, the control dependences are determined by the syntax). As this way some statements may be dependent on multiple predicates, the handling of predicate variables in the presence of jumps needs to be slightly extended (the details are given below). Currently, C "long jump" constructs are not handled, but they could be treated in the same way.

7) The conceptual algorithm uses the concept of execution history to record the instruction numbers executed. To be able to slice a C program, however, some other information is also needed that is generated upon executing the program, and which is used by the slicing algorithms. This includes the addresses of variables, function calls and block scope information. We will call this extended execution history the *trace*.

8) Declaration lines will be added to the slices whenever the definition of the declared variable is added to the slices. Also, the eventual initializations will be added to the def-use representation.

9) Since programs generally rely on standard library code as well, we must handle interprocedural dependences arising from the parameters, side effects and return values of calls to library code. Since the source code of library functions is often unavailable, we will rely on the semantics of such functions and prepare, in advance, a def-use representation of each standard library function based on the specifications.

10) Real programs usually consist of multiple source files composed of header files (`.h`) and implementation files (`.c`), which produce translation units after preprocessing. Our slicing algorithm works on preprocessed units, which makes it possible to compute slices for the whole program. What is needed to achieve this is a global numbering of statements over all the source files of the program, and solving name identification for definitions coming from common header files and placed into multiple translation units, as we do with a linker.

Based on these considerations, the implementation of our dynamic slicing approach consists of four phases. During a static analysis, the def-use representation of the program is produced and stored on the disk, and the source code is instrumented.[2] Next, the instrumented code is built to produce an executable program, which is executed in the next phase. During this operation a trace of the program is produced with the help of the instrumentation code. Lastly, the slicing algorithm is executed, where the trace is used to drive the propagation of the dependences, in the global algorithm starting from the beginning of the trace. The slicing algorithm relies on the def-use representation produced in the first phase. Below, we will describe these phases and specific features of the implementation for C.

3.1 Static Analysis

Static analysis has two goals: to produce the def-use representation (Section 3.2) and to instrument the code (Section 3.3). Another task here is to create a mapping between the physical source code lines and the internal identifiers given to program elements by the analyzer. Our static analysis front end works on the preprocessed code, and it performs lexical and syntactic analysis, producing an annotated Abstract Syntax Tree as the result for each unit. The AST contains

[2] In this study, we used source code-level instrumentation, but other ways exist as well such as binary-level and virtual machine-level solutions. It should be added that source code-level instrumentation has the highest risk of changing program behavior, but when experimenting with our prototype we did not encounter any such problems.

sufficient information to compute the def-use representation and perform code instrumentation.

3.2 Def-Use Representation for C

In our implementation for the C language, an extended def-use representation is created and stored in a file, which will be used later by the algorithms. In the conceptual algorithm [5, 6, 7], the def-use representation was defined as *i. d: U* for each program instruction number *i*. For real C programs, this representation (also called the *D/U representation* below) will be extended so that it contains a sequence of *d: U* items for each instruction *i* in the program: $i.\ \langle (d_1 :\ U_1), (d_2 :\ U_2), \ldots, (d_{m_i} :\ U_{m_i}) \rangle$. We will use the notation $DU^C(i)$ for the D/U sequence of the *i*-th instruction.

This extension is needed because in a C instruction (i.e. an executable expression with side-effects), several l-values may be assigned new values. Note that the sequence order is important, since the *d* values of a previous D/U item can be used by the subsequent *U* sets. This sequence order is determined by the "execution-order" (evaluation) of the corresponding subexpressions. The order of the evaluation of subexpressions in C is not always defined by the language, hence there might be complications arising from the use of different compilers and compilation options. In our current implementation, we will rely on the parsing sequence determined by the context-free grammar of C, which proved to be sufficient in our prototype. In a production tool, however, care should be taken to handle the various possibilities.

The other modification needed for the D/U representation for C is that the variables (including artificially created ones) in it are not only simple scalar or predicate variables, but they can also take several different meanings as follows:

1) *Scalar variables.* These are the "regular" global or local variables (with static storage, they have a constant address for the actual call stack frame). The formal parameters of functions are also represented as if they were local variables in the function's scope. Note that dynamic variables used with dynamically allocated memory on the heap do not need special treatment as they will be treated as pointers and the corresponding allocator functions as library code (see above).

2) *Predicate variables.* Denoted by p_n, where *n* is the serial number of the predicate instruction, the predicate variables are artificial variables with the same semantics as those described in the conceptual algorithm. In the case of the C language, all iteration and selection statements will induce predicate variables. An additional, special form of predicate variables will be introduced, one for each function and will be denoted by *entry(f)*, to generalize the representation of control dependences. Such an "entry-

predicate" is defined upon entering the function *f* and is used by all statements outside any other predicates in the function.

3) *Output variables*. Denoted by o_n, the output variables are artificial variables that are generated at the places where a set *U* is used, but no other variable takes any value from *U*. These include function calls with their return values ignored, single expression-statements with no side-effects, jump statements, and some output statements in C such as `printf`.

4) *Dereference variables*. The notion of the (artificial) dereference variables is employed where a memory address is used in any possible way or where it gets a value through a pointer (or an array or structure member). They are denoted by d_n, where *n* is a global counter for each dereference occurrence. Dereference variables will be created for the following code constructs: `*expr`, `object.member`, `ptr->member` and `array[index]`. Note that in an implementation, some of these could be handled uniformly as a base pointer+offset, but source code instrumentation requires a different treatment. Dereference variables will be used in such a way that their dependences will be noted in the D/U representation only symbolically, while the actual dependences will be computed for the associated addresses written to the trace. Note that the order in which the dereference variables are stored in the use sets must be the same as the order in which they will be evaluated.

5) *Function call argument variables*. These are artificial variables denoted by *arg(f,n)*, where *f* is a function name and *n* is the function argument (parameter) number. An argument variable is defined at the function call site and used at the entry point of the function (by defining the formal parameter).

6) *Function call return variables*. Denoted by *ret(f)*, where *f* is a function name, the artificial return variables are defined at the exit point of the function and used at the function caller after returning.

In the extended D/U representation, regardless of the type of variable, all dependences are treated equally. For instance, a pointer dereference may be dependent on a predicate variable if the dereference subexpression is control-dependent on a predicate. This uniform handling allows a very concise capturing of the interdependences of the program, and a straightforward implementation of the algorithms. In the following, we will describe how the dependences in the D/U representation are built up and relate to special features of the C language.

Computation of the data dependences. Generally speaking, the structure of the D/U representation is such that it captures the definition-use relationships *locally* for each statement. This means that we do not need to deal with the classical problems of computing data dependences in the static case, as is required with the dependence graphs [12]; in our case only the names of the dependent variables

(and not the corresponding definition) need be stored. Thus, our representation for C can be constructed in a simple syntax-directed manner following the semantics of each C expression construct.

Function calls. Function calls and parameter passing are handled in the D/U representation using the artificial variables *arg* and *ret* (see above). Whenever a function call expression is found in a C instruction, a corresponding D/U item is created with the *arg* variable as the defined one and the appropriate use set. Next, in each function a D/U item is constructed for all its formal parameters in which the parameter is the defined variable (the parameter is later treated as a local variable) and the corresponding *arg* variable constitutes the use set. Furthermore, for each return statement in the functions a D/U item is created with the *ret* variable defined and the corresponding use sets. Lastly, at the call site these *ret* variables are used in the corresponding use sets for the expressions containing the function call. The order of elements in the D/U lists is important as this is required for the synchronization with the trace.

Structured control dependences. The predicate variables will be used in the D/U representation to capture the control dependences among the program instructions. In the case of structured control transfers (the *if* selection and the three types of C loops), for each predicate corresponding to the respective decision statement a predicate variable will be created and the dependences will be based on the nesting structure of the program; the directly nested statements of *if* branches or a loop will be dependent on the corresponding container predicate. To make the algorithm more general, for each function an additional predicate called the *entry-predicate* will be defined as well. The instructions that are not nested within another predicate statement will be dependent on the entry-predicate. (The entry-predicates are implicitly defined at the function beginning and their use sets are empty.) Note that shortcut logical expressions do not influence this operation.

Handling of goto and other unstructured jumps. While the direct control dependences can be readily determined for structured programs, *goto*-s and other arbitrary control transfers (*switch*, *continue* and *break*) must be handled in a more elaborate way. We will compute control dependences in the static analysis phase based on the traditional approach using postdominance relations [17], and then build the extended D/U representation based on this information. Namely, if an instruction *i* is found to be control dependent on some other instruction (which is then a predicate), we extend the use set of *i* with the corresponding predicate variable. Since in a program with arbitrary control flow an instruction may be control dependent on more than one instruction, our use sets may also contain several predicate variables. In one specific execution only one of them will be responsible for the actually realized control dependence, which we will call the *active predicate*. When propagating the dependences through the current instruction's use set, we must select just one predicate variable to continue with. If there are more predicate variables in the use set, our approach is to choose the one that has been defined most recently. In other words, for i^j. *d: U*, we will choose

predicate *p* for which *LD(p)* = max{*LD(r)* | *r* ∈ *U* and *r* is a predicate variable}, where *LD(v)* is the last definition of variable *v*, i.e. the execution step at which *v* was defined just before the *j*-th step where *i* was executed. (In the following, we will refer to execution history elements as *actions* with the notation i^j, where *i* is the serial number of the instruction executed at the *j*-th step or position.)

Complex l-values. A side effect of certain C expressions is that the sub-expression on the left hand side of the operator takes the value of the right hand side (this includes the assignment operators as the most common ones). These operators require that their left hand side be an l-value (meaning that it is modifiable). Quite frequently, the l-value is a simple variable occurrence, but these sub-expressions can be arbitrarily complex. In such cases, the D/U representation needs to be constructed carefully to include all the defined and used variables appropriately. One important issue is the handling of pointer dereference expressions of the form `*p`. Strictly speaking, the data pointed to by a pointer is not dependent on the address itself. However, we will apply a conservative approach and include such pointers as well (this approach is also used by some other algorithms). In Figure 1, we list some other cases and the way we treat them in our representation (following the principles for dereference variables introduced above).

```
a[i]       = r; //        d1:{r,i,a}
*(p+x)     = r; //        d2:{r,p,x}
m.a        = r; //        d3:{r}
p->a       = r; //        d4:{r,p}
```

Figure 1
Handling of field accesses

Clearly, this is a conservative approach as, for example, the array name `a` and the index variable `i` both appear in the use-set of the first statement; however from a computational point of view only the data at the address pointed to depends on `r`. Although debatable, here we shall choose this approach to be able to compute a conservative-type of dependence which can be used, for instance, to assist impact analysis.

Pointers, pointer dereferences, address-of and arrays. Our algorithm computes the dependences on memory locations, which makes the handling of pointers and related structures straightforward, but there are several special features worth mentioning. As we said previously, in the case of pointer dereferences both the pointer and the dereferenced object will be included. Using the address-of operator does not induce a new dependence because the address itself can be viewed as a constant value. All the other operations with pointers are treated in the same way as in the case of regular variables. Arrays can be handled in a similar way as pointers since they can be interpreted as pointers with appropriate offsets corresponding to the index. The only extension is that the variable(s) used in the index operators are also treated as used variables. Multiple pointers and indirections can be handled in the same way as well.

The handling of function pointers does not require major modifications to the presented algorithms, but we omitted these details from the formal algorithm (in the next section) to aid readability, and we describe them more fully here. Statically, we cannot determine the called function, so in the D/U representation we cannot use *arg(f,n)* and *ret(f)* variables either. Instead, we use their special form in which the actual names are not given, just some symbolic names of the form *arg(?,n)* and *ret(?)*. These temporary variables will be resolved upon the execution of the slicing algorithms as soon as the called functions become known.

Structs and unions. C language *unions* and *bitfields* can be handled in a conservative way. Namely, we will treat unions and bitfields as scalar variables because when we define a field we virtually define all the others as well. Bitfields can introduce multiple dependences due to overlaps in memory regions, which will be handled in a similar way to that with type-casts, described later on. In the case of *structs*, however, we want to preserve the individual tracking of the dependences of the fields as in the case with arrays. For this, we follow a similar approach to the handling of arrays because the structs can also be interpreted as memory regions with a fixed base address and offsets corresponding to the fields. That is, for each field access we create a distinct dereference variable, which we can use separately in the D/U sets.

The handling of the individual struct variables as parts of expressions is more complicated because the expression operations in this case will correspond to all the fields together (struct copying). In this case, we model the dependences for each field access combination (which may be recursive). The struct variable itself will not be part of the D/U sets, but all the references to it will be transformed to the actual field accesses for all the fields. Figure 2 provides examples for handling structs, members and dereferences (in the commented lines below line 2, we can see how the fields are modelled).

```
   struct S s,t,*p,*q;
   t.a = ...
   t.b = ...
1. q      = &t;   //     q : {}
2. s      = *q;   //     : {}
// s.a    = q->a; // d2:{q,d1}
// s.b    = q->b; // d4:{q,d3}
// ...
3. x      = s.a;  //     x : {d5}
4. p      = &s;   //     p : {}
5. y      = p->b; //     y : {p,d6}
```

Figure 2
Handling of struct variables

For instance, during a slice computation, the runtime addresses of `d5` and `d2` will be the same, which will result in correct dependences between `s.a` and `t.a`.

Type casts. Type casts can cause a problem for our slicing algorithms in cases where the sizes of the original type and the new type are different. The basic methods discussed above will lose any dependences among overlapping memory regions of the objects (e.g. structure members). The problems related to type casts can be handled only by maintaining the length of the referenced memory addresses as well as their starting address. We did not include this in the formal description of the algorithms for the sake of clarity, but we will overview the basic method here. The D/U representation does not include any specific extensions, but in the execution trace we will output the dereference addresses and the sizes of the variables in question, which will form regions instead of single addresses (`sizeof` can be used in the instrumented code for this purpose). The slicing algorithm will then take into account each byte of the referenced memory region, which will result in not losing any dependences; and this will be suitable for all kinds of type casts, including casts between scalars and pointers.

3.3 Instrumentation and the Trace File

The purpose of code instrumentation is to produce a semantically equivalent code that, upon execution, produces a *trace* of the execution. The trace records the executed i^j actions and other information required by the slicing algorithm. It is a linear sequence of elements with various meanings, which is, upon execution, stored in a file for later processing. The sequence can be described with a context free grammar shown in Figure 3.

$$
\begin{array}{rcl}
\langle trace\text{-}file \rangle & ::= & \{ \langle global\text{-}var \rangle \}\ \langle main\text{-}function \rangle \\
\langle global\text{-}var \rangle & ::= & G\ (\ \mathbf{id}\ ,\ \mathbf{addr}\ ) \\
\langle local\text{-}var \rangle & ::= & D\ (\ \mathbf{id}\ ,\ \mathbf{addr}\ ) \\
\langle function \rangle & ::= & FB\ (\ \mathbf{id}\ )\ \{ \langle function\text{-}body \rangle \}\ FE \\
\langle main\text{-}function \rangle & ::= & FB\ (\ \mathtt{main}\ )\ \{ \langle function\text{-}body \rangle \}\ FE \\
\langle function\text{-}body \rangle & ::= & \langle local\text{-}var \rangle \\
 & | & \langle action \rangle \\
 & | & \langle block\text{-}scope \rangle \\
\langle block\text{-}scope \rangle & ::= & BB\ (\ \mathbf{bnum}\ )\ \{ \langle function\text{-}body \rangle \}\ BE\ (\ \mathbf{onum}\ ) \\
\langle action \rangle & ::= & E\ (\ \mathbf{inum}\ ,\ \mathbf{jnum}\ )\ \{ \langle action\text{-}suffix \rangle \} \\
\langle action\text{-}suffix \rangle & ::= & P\ (\ \mathbf{addr}\ ) \\
 & | & \langle function \rangle
\end{array}
$$

Figure 3
Formal description of the trace

The order of elements in the trace is determined by the execution of the instrumented program. First the data for all of the global variables are dumped (mark *G* with the variable name and its actual address). Then the execution is traced starting with the `main` function. On entering a function, a function-begin

mark with the function name (*FB*) is generated, and on exiting it a function-end mark (*FE*) is generated. During the execution of a function body, three kinds of events can occur: the data for a local variable (*D*) is generated in a similar way to that for the globals, or a nested block (corresponding to a syntactic block in a C program) is generated with the delimiting marks (*BB* with a unique block serial number and *BE* with the identifier of an outer block), or an executable instruction (action) is traced. The delimiting marks are not generated only for the blocks according to the syntax with { and }, but for each jump instruction into or out of some blocks and single statement sub-instructions as well. The block identifier that comes with *BE* is the number of the block in which the next executable instruction is located. Usually, it is the block containing the current one, but in the case of unstructured jumps it may be any block in the current function.

An action is generated for each C instruction (expression) and it consists of two parts. The main part (*E*) designates the executed instruction number *i* and the execution step *j*. In addition, an optional list of information (the action suffix) related to the current instruction may be generated. Here, there are two types of action suffixes. If a function call is a part of the expression of the current action, the trace for the whole function will be dumped as a suffix for the current action. This can clearly result in a large amount of recursive data structures being generated, which may be similar at different instances if the invocation is similar. This could be optimized in an implementation by applying some kind of a compression; however, here we do not implement such a feature. The other kind of action suffixes will be generated whenever a pointer dereference is encountered in the expression of the current action. The accessed memory address is dumped into the trace using *P*. For each action, the additional dereferences will correspond to the relevant dereference variables in the D/U. Note that the order in which the *P* marks will be generated is the same as the way they are executed, and this order must also be the same as the corresponding dereference variables are listed in the D/U representation. This property will be exploited by the slicing algorithm.

To get the required contents of the trace file, the source code needs to be *instrumented* at several locations. At each relevant point, a call to an instrumenting function is generated, which will place the necessary marks into the trace file. We chose the instrumented code to be C++ rather than C for practical reasons.[3] For example, some instrumentor functions are easier to implement as template functions, and we can also put the calls to the instrumentor functions before the variable declarations. The instrumentor functions are provided in additional source and header files, which need to be included in the linking phase when the program is built. To implement the instrumentation for each trace element, several practical solutions had to be elaborated, for instance: block and function delimiting marks had to be placed at various places due to possible

[3] Note, that this solution might be problematic when certain language features are used in the original C program that are incompatible with the selected C++ compiler.

jumps; local and global variables are dumped using the address-of operator; action marks are generated for each expression using the comma-operator; dereference marks are generated by a C++ template function that returns the pointer to a type passed in the template parameter, etc. Figure 4 shows an excerpt from the C program bzip, its instrumented version and a part from the generated trace file.

Since the instruction numbers are generated incrementally, we need to maintain a data structure to map the instruction numbers to the physical file line numbers (line numbers will be essential in presenting the actual results of slicing). The method of mapping line numbers to instruction numbers depends on the actual implementation of the static phase. In our toolset, we used the information taken from our static analyzer for this purpose, which takes into account both the fully qualified file names and the absolute line numbers. Here, we can use line information got from both the preprocessed file and the original file locations.

```
Int32 nb, na, mid;
nb = 0;
na = 256;
do {
   mid = (nb + na) >> 1;
   if (indx >= cftab[mid]) nb = mid; else na = mid;
}
```

```
D_VA(&nb,"nb");
D_VA(&na,"na");
D_VA(&mid,"mid");
D_EH( 1259,D__ec++); D_EB() ,(nb = 0);
D_EH( 1260,D__ec++); D_EB() ,(na = 256);
do
{ /*BlockGuard*/
{
D_SB(243);
D_EH( 1262,D__ec++); D_EB() ,(mid = (nb+na)>>1);
D_EH( 1263,D__ec++); if ( D_EB() ,(indx>=(*D_P(&cftab[(mid)]))) )
{ /*BlockGuard*/
D_EH( 1264,D__ec++); D_EB() ,(nb = mid)
;} /*BlockGuard*/
else
{ /*BlockGuard*/
D_EH( 1265,D__ec++); D_EB() ,(na = mid)
;} /*BlockGuard*/
;
D_SC(242);
}
;} /*BlockGuard*/
```

```
D nb 0x0012F018
D na 0x0012F010
D mid 0x0012F014
E 1259 9578
EB
E 1260 9579
EB
SB 243
E 1262 9580
EB
E 1263 9581
EB
P 0x0012F3C0
E 1265 9582
EB
SC 242
```

Figure 4

Instrumentation and trace file example

3.4 Global Algorithm for C

The extended global algorithm for slicing C programs with the solutions to the problems elaborated on earlier can be seen in Figures 5 and 6. Here, the notation *TR >> tr* is used to denote the reading of the next trace element *tr* from the trace *TR*, which is viewed as a stream of elements, as described in Section 3.3. Other formalisms are self-explanatory. Note that for the sake of clarity we omitted such supporting activities as error handling and synchronization support between the trace and the algorithm.

The algorithm begins with the program GlobalAlgorithmForC, which has two input parameters; namely, program *P* that is to be sliced and input `x` for which the dynamic slices will be computed globally. First the trace is produced (in a file), which is read sequentially (lines 1-2). The algorithm is driven by the elements found in the trace, but its structure must be in sync with the static D/U representation (see, for example, function calls and dereference marks). The function calls are captured in the trace recursively, so they are also handled by the algorithm by recursively calling the function ProcessFunction when such a call is found. The main program of the algorithm (after storing the addresses of global variables on the scope stack written in lines 3-6) starts by processing the `main` function in line 7.

During processing, a helping structure is maintained for the local and global scalar variables. This structure (*sc*) is a stack of scopes that are entered dynamically upon execution. The scope stack is maintained in the function ProcessFunction, as dictated by the trace. Namely, a new function scope is created on the top when entering a function (*FB*) in lines 12-13. As we saw earlier, the block beginning (*BB*) and ending marks (*BE*) are found in the trace in the case of structured control flow and for unstructured jumps as well (lines 14-17). Therefore, a new scope for a block is created only if it has not already been created for the current function. Otherwise, the current scope pointer is simply set to this block. Since jumps into blocks are possible, they cannot be deleted upon exiting (only the current scope pointer is set), but the whole function scope is deleted when exiting (*FE*).

The other two activities performed in ProcessFunction are the storing of the addresses of local variables in the stack (*D*, lines 10-11) and the processing of the execution actions (*E*, lines 18-19) by the function ProcessAction. ProcessAction takes an action i^j, computes the corresponding dynamic dependence sets of the defined memory addresses and variables and outputs the corresponding slices. The DU^C items are processed for the statement *i* starting with the first one and the so-called *dynamic D/U item i. d'$_k$: U'$_k$* is computed for each step (*for* loop in lines 24-39). Then, the usual operations for computing the dynamic dependence sets are performed [5, 6, 7] (here, *DynDep* stores the actual dependences, while *LS* and *LD* denote the last defining statement number and execution step, respectively). First the used variables are processed (lines 26-33), then the dependence set for the defined variable is computed and output in lines 34-39.

```
program GlobalAlgorithmForC(P, x)
inputs:   P = program
          x = input of P for which dynamic slices should be computed
outputs: dynamic slices for all (x, i^j, V_i) criteria
          (j = 1 ... J, V_i = union of DU^C(i) sets)
globals:  tr = trace element
          sc = scope stack
begin
 1 Store trace TR
 2 TR >> tr
 3 Create global scope on the top of sc
 4 while tr = G(id, addr)
 5   Store id with addr on the top of sc
 6   TR >> tr
   endwhile
 7 ProcessFunction(main)
end
procedure  ProcessFunction(f)
begin
 8 while tr ≠ FE
 9   case tr of
10     D(id, addr) :
11       Store id with addr on the top of sc
12     FB(f) :
13       New function scope on sc
14     BB(block-num) :
15       If ∄ block scope with block-num for actual function on sc then create it
16     BE(outer-num) :
17       Set current scope pointer in sc to outer-num
18     E(i, j) :
19       ProcessAction(i, j)
     endcase
20   TR >> tr
   endwhile
21 Delete all block scopes and the function scope on the top of sc
22 TR >> tr
end
```

Figure 5
Global algorithm for C

Each static D/U variable is resolved with the help of the Resolve function (lines 30, 34). Resolving means finding the memory addresses which the scalar and dereference variables point to at the *j*-th step. Addresses of scalars are looked up in the scope stack by using the usual lookup rules for the function at the top of the stack (lines 41-42 of the Resolve function). The actual addresses of memory dereferences are taken from the trace (*P*), taking into account the fact that the order in which the addresses are dumped into the trace must be the same as the order the static D/U lists the dereference artificial variables (lines 43-45). All other variables (e.g. predicates) will be the same after resolution (lines 46-47).

```
procedure ProcessAction(i, j)
local:    PR = set of predicate variables
begin
23 S := ∅
24 for all (d_k, U_k) ∈ DU^C(i) items
25   PR = ∅
26   for all u_kl ∈ U_k
27     if u_kl = ret(g) for some function g then
28       TR ≫ tr
29       ProcessFunction(g)
       endif
30     u'_kl = Resolve(u_kl)
31     if u'_kl is a predicate variable then PR = PR ∪ {u'_kl}
32     else U'_k = U'_k ∪ {u'_kl}
     endfor
33   U'_k = U'_k ∪ {p}, for which LD(p) = max{LD(r)|r ∈ PR}
34   d'_k = Resolve(d_k)
35   DynDep(d'_k) = ⋃_{u'∈U'_k} (DynDep(u') ∪ {LS(u')})
36   LS(d'_k) = i
37   if d'_k is a predicate variable then LD(d'_k) = j
38   S = S ∪ DynDep(d'_k)
39   Output S as the dynamic slice for (x, i^j, V_i)
  endfor
end
procedure Resolve(x)
begin
40 case x's type of
41   scalar:
42     return x' = lookup variable x in sc and get its address
43   dereference:
44     TR ≫ tr
45     return x' = addr from tr in the form of P(addr)
46   all other artificial:
47     return x' = x
  endcase
end
```

Figure 6
Global algorithm for C (continued)

The other modification for processing one action is that the control dependences are handled in the way described in Section 3.2. Namely, we determine the active predicate by choosing the one from the set U_k that was the most recently defined in line 33 (*PR* contains all static predicate dependences, from which the one with maximal *LD* is taken. ProcessAction also implements the handling of function calls by invoking ProcessFunction recursively, if a function call return variable (*ret*) is found in the D/U. In this case, the trace is processed until the function returns (lines 27-29).

3.5 Implementation and Measurements

We implemented the presented algorithm in a prototype tool and performed experiments about the feasibility of the approach on real-world programs. We used five small to medium size C programs from the open source domain, whose main parameters can be observed in Table 1.

Program	Lines of Code	Statements	Static variables	Scalar variables	Predicate variables	Dereference variables
bcdd	442	78	179	31%	24%	2%
unzoo	2,900	932	1,896	26%	34%	5%
bzip	4,495	2,270	4,184	25%	30%	5%
bc	11,554	3,441	6,898	19%	34%	6%
less	21,488	5,373	10,605	18%	41%	4%

Table 1
Basic program properties

The number of variables found in the program and their types are relevant to the performance of the algorithm. The last four columns of the table overview the total number of static variables in the programs and how their types are distributed. However, the actual computation complexity of the slicing algorithm is mostly determined by the dynamic properties of program elements, which we present in Table 2. The first two columns show the number of test cases we used in our experiments and the average length of the corresponding execution histories, respectively. The next two columns show the average number of dynamic variables (such as memory locations used) and the sizes of the use sets occuring in each step during execution. These two properties are primarily responsible for the actual dependence set sizes and ultimately the space and time costs of the algorithm. The resulting dependence set sizes (the dynamic slices) are shown in the last column in percentage relative to the program size.

Program	Test cases	Avg. actions	Avg. dynamic variables	Avg. use set size	Avg. dependence set size (wrt. program size)
bcdd	5	623.4	34	5.4	18.27%
unzoo	13	169,557.3	1,173	8.9	5.17%
bzip	18	14,245.7	985	8.1	4.35%
bc	49	5,807.3	634	12.6	3.37%
less	14	101,178.5	2,117	6.9	4.80%

Table 2
Dynamic properties of the programs and the slicing algorithm

The length of the execution naturally influences the expected number of dynamic variables. However, the use set sizes and dependence set sizes typically do not depend on this property, but on the logical structure of the program and its computations. Hence, we may conclude that the performance of the algorithm in each step will not be dependent on the length of the execution, which is one of the primary benefits of the method compared to previous approaches.

Conclusions

The dynamic slicing approach presented above does not require a complete dependence graph to be built as a preprocessing step, but instead our algorithm makes use of customized data structures. This has obvious advantages in practical situations and will presumably make the approach scalable and feasible as well. However, other technical issues remain to be solved (for instance, handling the C "long jump" construct) and an optimized version should be developed before making the algorithm available as a prototype tool to other researchers.

Other possible ways of improving the basic algorithms include the idea of trace block summaries [16]. This could be exploited in the implementation for debugging applications; and this is what we plan to investigate in the near future.

References

[1] Weiser, Mark. Program Slicing. IEEE Transactions on Software Engineering, SE-10(4):352–357, 1984

[2] Xu, Baowen, Qian, Ju, Zhang, Xiaofang, Wu, Zhongqiang, and Chen, Lin. A Brief Survey of Program Slicing. ACM SIGSOFT Softw. Eng. Notes, 30(2):1-36, 2005

[3] Agrawal, Hiralal and Horgan, Joseph R. Dynamic Program Slicing. In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, number 6 in SIGPLAN Notices, pp. 246-256, White Plains, New York, June 1990

[4] Zhang, Xiangyu, Gupta, Rajiv, and Zhang, Youtao. Cost and Precision Trade-offs of Dynamic Data Slicing Algorithms. ACM Transactions on Programming Languages and Systems, 27(4):631-661, July 2005

[5] Beszédes, Árpád, Gergely, Tamás, and Gyimóthy, Tibor. Graph-less Dynamic Dependence-based Dynamic Slicing Algorithms. In Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06), pp. 21-30, September 2006

[6] Beszédes, Árpád, Gergely, Tamás, Szabó, Zsolt Mihály, Csirik, János, and Gyimóthy, Tibor. Dynamic Slicing Method for Maintenance of Large C Programs. In Proceedings of the 5th IEEE European Conference on Software Maintenance and Reengineering, pp. 105-113, March 2001

[7] Gyimóthy, Tibor, Beszédes, Árpád, and Forgács, István. An Efficient Relevant Slicing Method for Debugging. In Proceedings of ESEC/FSE'99, number 1687 in Lecture Notes in Computer Science, pp. 303-321, Springer-Verlag, September 1999

[8] Beszédes, Árpád, Gyimóthy, Tibor, Lóki, Gábor, Diós, Gergely, and Kovács, Ferenc. Using Backward Dynamic Program Slicing to Isolate Influencing Statements in GDB. In Proceedings of the 2007 GCC Developers' Summit, pp. 21-30, July 2007

[9] Szegedi, Attila and Gyimóthy, Tibor. Dynamic Slicing of Java Bytecode Programs. In Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05), pp. 35-44, IEEE Computer Society, September 2005

[10] Beszédes, Árpád, Faragó, Csaba, Szabó, Zsolt Mihály, Csirik, János, and Gyimóthy, Tibor. Union Slices for Program Maintenance. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002), pp. 12-21, IEEE Computer Society, October 2002

[11] Tip, Frank. A Survey of Program Slicing Techniques. Journal of Programming Languages, 3(3):121-189, September 1995

[12] Horwitz, Susan, Reps, Thomas, and Binkley, David. Interprocedural Slicing using Dependence Graphs. ACM Transactions on Programming Languages and Systems, 12(1):26-61, 1990

[13] Agrawal, Hiralal. Towards Automatic Debugging of Computer Programs. PhD thesis, Purdue University, 1992

[14] Venkatesh, G. A. Experimental results from dynamic slicing of C programs. ACM Transactions on Programming Languages and Systems, 17(2):197-216, March 1995

[15] Korel, Bogdan and Laski, Janusz W. Dynamic Program Slicing. Information Processing Letters, 29(3):155-163, October 1988

[16] Zhang, Xiangyu, Gupta, Rajiv, and Zhang, Youtao. Precise Dynamic Slicing Algorithms. In Proceedings of the 25th International Conference on Software Engineering, pp. 319-329, May 2003

[17] Muchnick, Steven S. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997

[18] Galli, Tamás, Chiclana, Francisco, Carter, Jenny, and Janicke, Helge. Towards Introducing Execution Tracing to Software Product Quality Frameworks. Acta Polytechnica Hungarica, 11(3):5-24, 2014