Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 207–218.

Inter-procedural static slicing using advanced caching algorithm*

Mihály Biczó, Krisztián Pócza, Zoltán Porkoláb

Dept. of Programming Languages and Compilers
Fac. of Informatics, Eötvös Loránd University
e-mail: kpocza@kpocza.net, mihaly.biczo@t-online.hu, gsd@elte.hu

Abstract

Since the notion of program slicing was introduced in the early 80's, the area has seen a continuous evolution. While the more recent approach of dynamic slicing gained significant interest, static slicing has also been widely studied. Original methods used dataflow analysis techniques, later different approaches using multi-graphs were formed. Although these improved algorithms offer the capability to slice real-world applications, their usability is still heavily confined by their resource intensive characteristics. In this paper an advanced caching algorithm intended to decrease average complexity of SDG-based static slicing is proposed. Caching different formal parameter setups of a procedure might imply significant speed growth since re-computing dependences between formal input and output parameters is avoidable.

Keywords: static slicing, caching, inter-procedural

MSC: 68M15 Reliability, testing and fault tolerance

1. Introduction

The notion of *program slicing* [12] has emerged at the late seventies and early eighties. In [13] Weiser defined the concept of slicing as identifying parts of a program that satisfy specific criteria. The original idea was to map mental abstractions made by programmers to a reduced set of source lines in order to ease the refactoring, debugging and other program maintenance tasks.

Along with the evolution of programming languages and environments, many slightly different slicing definitions and algorithms have been developed. The introduction of procedures and pointers, later object orientation and it is "side-effects" like polymorphism altered existing techniques. On the part of multi-threaded programming, inter-process communication has widely been studied [7].

*Supported by the Hungarian Ministry of Education under Grant FKFP0018/2002.

Wide application possibilities account not only for the variation of definitions and corresponding algorithms, but also for their fine-grained division. Basically, there are two different approaches to program slicing. In the case of static slicing the actual input of the underlying program is completely disregarded and only statically available information is used. While static slicing neglects actual program input, dynamic slicing [1, 2, 10] takes it into consideration.

Static slicing can be further classified based on algorithms and data structures used during program analysis. While intra-procedural static slicing is only capable of analyzing demonstrative programs containing one main method, its inter-procedural counterpart can be used to study real-world applications as well.

Weiser's intra-procedural method is based on calculating consecutive sets of indirectly relevant statements based on control flow and data dependency analysis in a graph whose vertices are position-labeled statements of the program and edges correspond to dependences among them [7, 12, 13]. In the case of inter-procedural slicing, these dependences have to be extended from procedure to procedure. The notion of the program dependence graph or PDG [6] was first introduced in [4] based on the idea that different kind of dependences could be represented in a single graph. The PDG have to be extended to appropriately describe multi-procedure environments. The extension, the so-called system dependence graph is able to model languages that allow procedure calls. The system dependence graph is a graph composed of simple PDGs with additional dependence edges among them.

In spite of the fact that the SDG is a simple, straightforward, almost natural extension of the PDG, the method presented earlier suffers from significant drawbacks from the real-world application point of view. When the SDG is applied to larger programs, it can easily pass the limit where dependences can be tracked and understood. Also, there is a lack of capability to slice recursive programs since recursion may lead to dead loops in the graph and there is no extra information regarding when to stop analysis.

As a consequence, to support larger programs, new methods and approaches have to be established. Xu et al. [14] have developed a new kind of analysis of recursive Java programs. Based on the approach of Horwitz, this method uses unconnected PDGs that interact with each other along parameter dependences. Therefore, inter-procedural dependence analysis can be transformed to simple intra-procedural analysis.

Xu introduced an inter-parameter dependence set that substitutes summary edges in the SDG. Procedures can interact with each other only via parameter dependencies. Parameter dependences – mappings between actual and formal parameters – can be calculated using the algorithm given in [14]. After the subscribed transformations procedure calls are as easy to handle as any other statement in the program.

In [5] Gallagher presented a stack based alternative to SDG based inter-procedural static slicing in order to eliminate drawbacks imposed by connected PDGs. The notion is to generate as precise slicing criteria as possible using the approach of Horwitz. The equivalence of the stack based and classical approach is also covered.

Our work was motivated by the above mentioned alternative solutions especially [5]. In Section 2 the basic version of our modified algorithm will be discussed in detail. In Section 3 we fine tune the algorithm in order to access high speeds when slicing real-world applications. We introduce the idea of caching, or in other words, the possibility to establish a mapping between formal output and input parameters. Handling recursion will also be solved.

Section 4 gives some insight into the complexity estimation of traditional approaches and the improved algorithm presented.

2. Basic Algorithm Outline

From the previous sections it is clear what the limitations and difficulties of recent static slicing algorithms are. Based on previous approaches we present an algorithm which holds some new ideas also.

In this section the basic algorithm will be covered. Some optimizations and a method for handling recursion will be presented in the next section.

Gallagher's idea [5] of using a named set of PDGs joined only by the call graph instead of building the SDG and Xu's [14] idea of maintaining in/out parameter dependences across method calls were the main inspirations to think of interprocedural slicing methods again.

In an SDG, call edges and parameter nodes connect the PDGs of all methods. When the SDG is applied to a larger program it might be too complicated to be analyzed and understood. We tried to solve backward static slicing without using the SDG.

2.1. Modified CFG and PDG

The classical CFG and PDG consist of assignment, condition and loop nodes. We complement them with a new type of node called *method invocation* node. In order to distinguish the modified graphs from traditional ones, we use the notation mCFG and mPDG. The additional node stores the signature of the target method, the possible input (ref) and output (def) parameters and also real input and output parameters. There are some parameters which are treated as input and output at the same time (e.g. reference parameters).

When the mCFG is built and a method invocation statement is encountered in the source, the corresponding method invocation node is inserted to the graph. Control dependency and control dependence edges are treated in the usual way.

The method of building data dependence edges in the mPDG is almost the same except when a method invocation node is found.

Definition 2.1. A node m is data dependent on node n, if

1. there is a path p from n to m in the $mCFG$,
2. there is a variable ν, with $\nu \in \big(\mathrm{def}(n) \cup \mathrm{postdef}(n)\big)$ and $\nu \in \big(\mathrm{ref}(m) \cup \mathrm{postref}(m)\big)$,

3. and for all nodes $k \neq n$ of path p, where $\nu \notin \big(\mathrm{def}(k) \cup \mathrm{postdef}(k)\big)$.

It is clear that this definition is an upper estimation of possible data dependence nodes. Nodes can be added to or removed from the mPDG dynamically.

2.2. Algorithm steps

Before a method invocation it cannot be decided whether the caller will be interested in all output – and also input – parameters. This is why they are treated as only possible input and possible output parameters. Unused edges induced by unused input parameters can be safely removed from the mPDG. Fortunately there is no need to add new edges to the mPDG due to unused input parameters because input parameters do not get new values thus would not kill any data dependence edge.

Main steps of the algorithm:

0. Preprocessing: Generate the mCFGs and mPDGs of all methods in the system. Identify the starting method.

1. Preparing for slicing: The preprocessed mPDGs must be unchanged. Clone the mPDGs. Set slicing criterion, select starting nodes for the mPDG and add them to the worklist.

2. Selecting next node: Select the 'best' node from the worklist. We will discuss the selection algorithm later.

3. New method: If the type of the selected node is not method invocation then follow the classical slicing algorithm. If its type is method invocation then identify real actual output parameters, assign formal parameters to each of them and recursively go to step 1. Returning from the recursion identify the corresponding actual input parameters of the used formal input parameters. Remove the data dependence edges from the (cloned) mPDG starting at the method invocation node induced by input parameters which were not used in the called method.

4. Refresh worklist: Select all control dependence and data dependence edges ending with the current node. Add their starting node to the worklist and to the slice. For every data dependence edges with a starting node of type method invocation add the variable which induced the edge to the real actual output parameter list of the called method (at starting node). Add all statements to the worklist from which a control dependence edge points to the investigated method invocation node.

5. While not end: Go to step 2 until the worklist is empty.

2.3. Preprocessing and Preparation

Control dependence and data dependence are static properties so mCFGs and mPDGs of all methods can be generated first. A new node for every output parameter should be inserted to both mCFG and mPDG. In a later step, nodes that are directly or indirectly control dependent on any node must be identified, so while generating a mCFG store this dependence information also. These graphs will serve as master instances of mPDGs to be generated later. At this point the starting method is selected.

2.4. Selecting Next Node From Worklist

Control dependence and data dependence are static properties so mCFGs and mPDGs of all methods can be generated first. A new node for every output parameter should be inserted to both mCFG and mPDG. In a later step, nodes that are directly or indirectly control dependent on any node must be identified, so while generating a mCFG store this dependence information also. These graphs will serve as master instances of mPDGs to be generated later. At this point the starting method is selected.

Listing 1. Source code fragment (C# dialect).

```
1: void Main()
2: {
       ...
3:     A(out x, out y);
4:     B(y, out z, out a);
5:     int b = a;
6:     int res = z + x + b;
       ...
7: }
```

Data dependences of Listing 1 can be seen on Figure 1. Nevertheless, the assignment at line 6 is directly control dependent on assignment at line 5 and also control dependent on method invocations at line 3 and 4. Assignment at line 5 is directly control dependent on method invocation at line 4. Method invocation of B at line 4 is directly control dependent on statement at line 3. Let us assume that we are at assignment at line 6. In the next step nodes corresponding to statements 3, 4 and 5 are inserted to the worklist. In the next step assignment 5 has to be chosen, because if we choose 4 then we would miss one output (possible def) parameter, if we choose 3 we would miss other output (possible def) parameter.

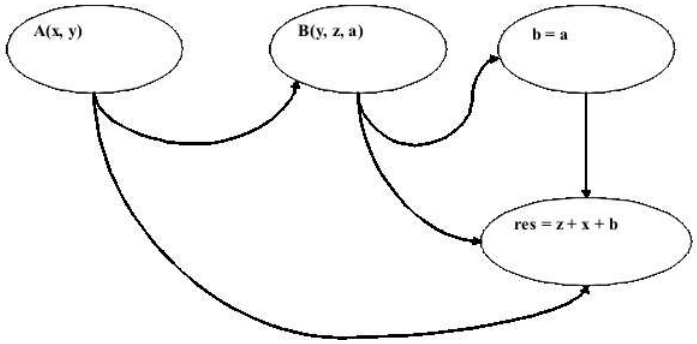

Figure 1: mPDG fragment

The algorithm selecting the next "best" node is simple: If there is any node whose type is not method invocation then select that node otherwise select the next method invocation node from which there is no control dependence edge to any node in the worklist. Note that when there are two method invocations in the body of a loop and they use one or more output parameters of each other, some further problems arise. In this case they are control dependent on each other therefore the algorithm cannot choose any of them. We can overcome this problem by introducing some restrictions to our algorithm or allow a looser slice in that situation. The restriction would be simply to exclude this kind of code structure knowing it is very uncommon in real world applications. On the other hand if this construction is allowed we can still overestimate the slice by assuming that one of the methods has real outgoing data dependence edges from all of its output parameters.

2.5. When a Method Invocation Node is Selected

If the type of the selected node is not method invocation then follow the classical slicing algorithm. If its type is method invocation then identify real actual output parameters, assign formal parameters to each of them and recursively restart the same slicing algorithm using new slicing criterion (containing real formal output parameters) and mPDG of the invoked method. As can be seen this algorithm does not handle recursion. The method we used for handling recursion [14] will be presented in the next section and the pseudo code of the full algorithm will be presented also there.

Returning from the recursion identify the actual input parameters of the used formal input parameters. Remove the data dependence edges from the mPDG starting at the method invocation node induced by input parameters which were unused in the called method. If these edges would not be removed then the algorithm could follow them and add unused statements to the slice as Weiser's original method did.

Lemma 2.2. *Removing these edges is safe.*

Proof. It is obvious that these edges can be removed because the variables which induced them are not used in the called method; the statements which gave them value are not in the slice. In parallel of the edge deletion no new edges should be inserted because the unused input parameters are only referenced variables, consequently no data dependences are killed by them. □

2.6. Refreshing Worklist

Adding new nodes to the worklist is done almost in the legacy way however there are some minor differences. When a method invocation node is reached along a data dependence edge, then the variable which induced this edge is added to the real output parameters of the called method.

Look at the following pseudo code where W is the working set, S is the slice and w means the actual node:

Listing 2. Worklist refreshing.

```
Function: RefreshWorkList(Node w)
foreach edge  {pdg.Edges ending with w} loop
    contDepOnNodeSet.Empty
if edge is DDE && edge.BeginNode is MethodInvoke then
        MethodInvoke mi = edge.BeginNode
contDepOnNodeSet = pdg.ContDepOnLoops(mi)
        mi.Defs.Insert(edge.AffectVar)
    end if

    if not S.Contains(edge.BeginNode) then
        W.Insert(ContDepOnLoops)
        W.Insert(edge.BeginNode)
        S.Insert(edge.BeginNode)
    end if
end foreach
```

3. Optimization, Handling Recursion and OOP

In the previous section a basic algorithm was presented without optimizations, support for recursive calls and object oriented constructions. In this section these extensions will be presented.

3.1. Optimization – Caching

Caching different formal parameter set-ups of a procedure might imply significant speed growth since re-computing dependences between formal input and output parameters is avoidable.

Consider the following code fragment:

Listing 3. Example (C# dialect).

```
void Main()
{
int a = 0;
int b = 1;
int x = 1, y=2;
A(a, x, out c);
A(b, y, out d);
int res = c+d;
}

void A(int x, int y, out int z)
{
z = DoCalculation(x);
Console.WriteLine(y);
}
```

Let the slicing criterion be (<End of Main>, {res}). As it can be seen method A is used twice and its output parameter is used after both calls. The question is whether it is possible to store formal input parameters given a set of formal output parameters that affect the value of variables in the slicing criterion of the caller.

It turns out that the answer is yes since static analysis is independent of the current value of parameters. In Listing 6 actual output parameters `c` and `d` are also used in method `Main` so the information that the usage of formal output parameter `z` involves the usage of formal input parameter `x` but not `y` can be cached.

When a method is reached do the following:

1. If this method using the same formal output parameter setup cannot be found in the cache, then slice this method, identify used formal input parameters and put it in the cache.

2. If it can be found simply abandon the slicing step and retrieve the used formal input parameters from the cache.

The algorithm continues in the same way as presented in the previous section.

This mechanism might be very complicated to implement in the SDG based static slicing algorithm.

The code fragment in Listing 4 shows the full algorithm with caching and recursion.

Listing 4. Pseudo code of main algorithm.

```
Function: DoSlice(MethodInfo curMethod, mPDG pdg,
mCFG cfg, SlicingCriterion crit)
W.Insert(crit.Node)
S.Insert(crit.Node)
while W not empty loop
```

```
    w = W.RemoveBest()
    if w is MethodInvoke then
        MethodInvoke mi = w
        MethodInfo targMethod = mi.targetMethod;
        mPDG targetmPDG = targMethod.mPDG.Clone
        outFormParams = targMethod.Actual2Formal(mi.Defs)
        vis = callChain.Isvisited(curMethod, targMethod)
        if not vis then
            inputFormalParams =
targMethod.SliceCache.AffectedIns(outFormParams)
callChain.SetVisited(curMethod, targMethod)
if inputFormalParams is null then
                targetCrit.ResetSlicingCrit(outFormParams)
                targetmCFG = targMethod.mCFG
                slice=DoSlice(targMethod, targetmPDG,
                    targetmCFG, targetCrit)
                inputFormalParams=slice.QueryReferencesInputs
                targMethod.SliceCache.Add(outFormParams,
                    inputFormalParams, slice)
end if
inputActParams=targMethod.Formal2Actual(
inputFormParams)
            mi.Refs=inputActParams
            pdg.RemoveNotUsedDataDepEdges();
        else
            targetmCFG = targMethod.mCFG.Clone
            targetCrit.ResetSlicingCrit(outFormParams)
            contDepOn = pdg.ContDepOn(mi)
            targetmPDG.RemoveChildren(contDepOn)
            targetmCFG.RemoveAndReconnect(contDepOn)
            CalcDataDepEdges(targetmPDG, targetmCFG)
slice=DoSlice(targMethod,
targetmPDG,targetmCFG, targetCrit)
            inputFormalParams=slice.QueryReferencesInputs
            targMethod.SliceCache.Add(slice)
inputActParams=targMethod.Formal2Actual(
inputFormParams)
            mi.Refs=inputActParams
            pdg.RemoveNotUsedDataDepEdges();
        end if
    end if
    RefreshWorkList(w)
end loop
```

3.2. Handling Recursion

Sometimes it is very complex to handle recursive method calls nonetheless there are a lot of methods implemented to solve this problem.

When we implemented our algorithm we have chosen Xu's method because it also uses separate mPDGs for all methods.

There are two types of recursion: direct and indirect. In direct recursion a method calls itself back, in indirect recursion there is a longer path.

We will present only direct recursion in a very sort form. Both types of recursion are covered in [14]. Xu's idea is simple and efficient: Use a call graph, detect when the method calls itself and simulate the last step of the recursion by not allowing the condition or loop to run to the recursive call. So remove all the statements from the mPDG and the mCFG which are inside the loop or branch of the condition where the recursive call is and reconnect mCFG. Of course we have to clone the mCFG and the mPDG also and no cache management has to be used.

3.3. Object Oriented Languages

When the aim is to work with object oriented languages like C++, C# and Java we have to apply some extensions to handle class variables and polymorphism.

Class variables, static and global variables (if the language allows them) are treated as method parameters. Objects whose type might change during execution are called polymorphic objects and the methods of the base class can be overridden in derived ones. Fortunately modern object oriented languages are mainly statically typed thus the possible types of an object can be determined statically. When a not polymorphic method is found no further analysis is required. For polymorphic methods all corresponding methods have to be analyzed.

There are new language features in C# like delegates which are type-safe function pointers. Delegates are registered for an event and are activated when the event occurs. Fortunately it is possible to determine the potential delegates which might execute based on their type. Other new features are properties which originate from Delphi and can be simply treated as get/set methods.

4. Summary

The cost of original Horwitz, Reps and Binkley algorithm [6] is dominated by the cost of computing summary edges according to [11]:

$$O((TotalSites * E_{PDG} * Params) + (TotalSites * Sites^2 * Params^4)),$$

where $TotalSites$ is the total number of call sites in the program, E_{PDG} is the maximum number of edges in any PDG, $Params$ is the maximum number function parameters and $Sites$ is the maximum number of call sites in any procedure.

In [11] a new algorithm was presented by Reps et. al. for computing summary edges in SDG whose cost is bounded by:

$$O((P * E_{PDG} * Params) + (TotalSites * Params^3)),$$

where, in addition, P stands for the total number of procedures in the program.

In both cases the complexity depends on two factors. In the first case (original HRB algorithm) the first term is the cost of SDG compression and the second term is the cost of computing summary edges. In the second case there is no need to compress the SDG. The cost of both algorithms is dominated by the cost of finding summary edges. The second algorithm is asymptotically faster than the first one. Recursive functions are not mentioned.

While analyzing our algorithm the following can be established:

$$DiffSites \leqslant TotalSites$$
$$DDE = V_{PDG} * E_{CFG} * Params$$
$$O((DiffSites * E_{PDG} * M^2) + (RecCalls * DDE) + (RecCalls * E_{PDG} * M^2))$$

In practice $DiffSites \ll TotalSites$.

DDE denotes the complexity of data dependency calculation in our implementation. Above we presented the full formula of our algorithm with recursion. The first term stands for non-recursive methods, the second and third for recursive methods.

The running time of non-recursive part of our algorithm depends on the number of different call sites ($DiffSites$) because of caching. This number is bounded by the total number of call sites but hopefully industry applications have much less different call sites than the total number of call sites. M stands for the maximum number of method invocations in any method; V_{PDG} is an upper boundary of vertices in the mPDG of any method.

We have to modify data dependence edges every time the last step of recursive calls ($RecCalls$) is simulated as the second term shows. The third term is the same as first but for recursive methods.

We do not use summary edges thus we cannot compare our algorithm to the mentioned previous approaches directly.

In order to test the algorithm proposed earlier, we have implemented a pilot application that is capable of slicing C# programs satisfying certain restrictions. The source code might contain static functions with arbitrary program constructions (assignment, condition, loop, method invocation) in a single class using local value type variables.

We used an earlier version of Marcel Debreuil's C# source code parser library which employs the ANTLR parser generator. Compilation was performed using Microsoft Visual Studio 2005 beta codenamed Whidbey. We tested both non-recursive and recursive programs. Every time we tested the resulting slice was the same as for the HRB algorithm [6].

References

[1] Agrawal, H., Horgan, J. R., Dynamic program slicing, In *SIGPLAN Notices*, No. 6, (1990), 246–256.

[2] Beszédes, Á., Gergely, T., Szabó, Zs. M., Csirik, J., Gyimóthy, T., Dynamic slicing method for maintenance of large C programs, CSMR (2001), 105–113.

[3] Binkley, D. W., Gallagher, K. B., Program Slicing, Advances in Computers, Vol. 43, (1996).

[4] Ferrante, J., Ottenstein, K., Warren, J., The program dependence graph and its use in optimization, *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[5] Gallagher, K. B., Some Notes on Interprocedural Program Slicing, *IEEE International Workshop on Source Code Analysis and Manipulation*, 2004.

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

[7] Krinke, J., Advanced Slicing of Sequential and Concurrent Programs, PhD Thesis, Universität Passau, (April 2003).

[8] Lakhotia, A., Improved interprocedural slicing algorithm, *Technical Report CACS TR-92-5-8*, University of Southwestern Louisiana, (1992).

[9] Ottenstein, K. J., Ottenstein, L. M., The program dependence graph in software development environment, *ACM SIGPLAN Notices* Vol. 19(5), (1984), 177–184.

[10] Pócza, K., Biczó, M., Porkoláb, Z., Cross-language Program Slicing in the .NET Framework, .NET Technologies 2005, Pilsen, Czech Republic (accepted).

[11] Reps, T., Horwiz, S., Sagiv, M., Rosay, G., Speeding up Slicing, *ACM SIGSOFT Software Engineering Notices* 19, 11–20.

[12] Tip, F., A survey of program slicing techniques, *Journal of Programming Languages*, 3(3) (Sept. 1995), 121–189.

[13] Weiser, M., Program Slicing, *IEEE Transactions on Software Engineering*, SE-10(4) (1984), 352–357.

[14] Xu, B., Chen, Z., Dependence Analysis for Recursive Java Programs, In *SIGPLAN Notices* No. 12, 70–76.

Mihály Biczó, Krisztián Pócza, Zoltán Porkoláb
Dept. of Programming Languages and Compilers
Fac. of Informatics, Eötvös Loránd University
Pázmány Péter sétány 1/c.
H-1117 Budapest
Hungary