Interprocedural Static Single Assignment Form in Bauhaus

Stefan Staiger Gunther Vogel Steffen Keul
Eduard Wiebe
Institute of Software Technology
University of Stuttgart
{staiger, vogel, keul, wiebe}@informatik.uni-stuttgart.de

Technical Report TR 2007/03

Abstract

In this paper we describe interprocedural static single assignment form (ISSA) with optimizations as implemented in the Bauhaus project. We explain our framework which uses an abstract program representation enabling us to use different pointer analyses ranging from fast but imprecise to slow but precise ones. Our implementation includes the computation of (may and must) side effects and optimizations like pruning definitions with simple linear-time algorithms. This paper also provides comprehensive test results and statistics for a large test suite.

1. Introduction

Analyzing programs is an important activity in both compiler construction (see [23]) and software reengineering. Such analyses gather information about the program being compiled or reengineered. Many of these analyses need to know the data flow in the program. For example, wherever a variable is used in the program, we want to know the assignments ("definitions" in compiler terminology) that could have set the value used there. A first idea here is to insert pointers from all uses of a variable to all definitions reaching that use. Static single assignment (SSA) form [8] simplifies this situation in that every use of a variable in SSA form refers to exactly one definition. To achieve this, the construction algorithm for SSA form inserts artificial definitions wherever different definitions may reach a certain point.

In this paper we consider an extension to the classical SSA form, namely *interprocedural* SSA form (ISSA). Although much work was published on SSA in the last decade (e.g. [3, 23, 27]), a clear description of what ISSA form looks like and how it can be constructed is still missing. To the best of our knowledge, the only publications mentioning this topic are from Liao [20] and focus on a different application (semi-automatic parallelization).

Constructing ISSA form has to face several challenges. To stay conservative, it is necessary to include side effects of subprogram calls in the construction process. Moreover, since no pointer analysis is superior [13], an implementation should be able to support different pointer analyses. In this paper we describe our ISSA implementation which addresses these problems. We explain our abstract program representation that serves as an interface to pointer analysis and how it is created. Based on that representation, we create the graph of definitions and uses. We furthermore provide a simple and efficient algorithm which prunes ISSA form and improves side-effect information.

ISSA analysis as described in this paper was implemented in the context of the Bauhaus project [25] which offers a comprehensive infrastructure of tools and libraries for program analyses.

The structure of the paper is as follows: Section 2 reviews basics on SSA and discusses related work and applications. Section 3 gives an overview of our ISSA analysis and describes some challenges a real-world implementation has to cope with. In Section 4 we continue with the description of our abstract program representation that serves as an interface to pointer analysis. Section 5 explains how this program representation can be constructed, including the computation of side effects. Then Section 6 clarifies the construction of ISSA form and Section 7 shows useful and fast optimizations which improve precision and reduce complexity. Section 8 provides statistics and results for our test suite. In Section 9 we describe details on our concrete implementation in Bauhaus before we draw our conclusions in Section 10. Compared to the conference paper [29], this technical report contains extended sections, results for a larger test suite, and an additional section on the implementation in Bauhaus.

2. SSA: basics, variants, and related work

Static single assignment form (SSA) became popular mostly through the classical paper by Cytron et al. [8]. The key property of SSA is that every use of a variable refers to exactly one definition or, equivalently, that the definition of a variable *dominates* every use. This means that a variable is a name of a value (in the mathematical sense) and not of a storage place and implies correspondence between SSA and functional programming [2, 16].

To achieve the goal of one definition, it is not enough to rename a variable with every new definition. Furthermore, we must introduce artificial definitions wherever two distinct definitions merge (e.g., at a join point in control flow) or wherever a potential definition might override another definition (e.g., an assignment through a pointer). These artificial definitions are traditionally called *ϕ-nodes* or *ϕ-functions*: they take different names of the same variable (i.e., different reaching definitions) as input and produce a new definition as output.

Nowadays, we know of different versions of SSA. These versions differ in the number of ϕ-nodes they introduce. Obviously, client analyses using SSA form become faster and consume less memory if less ϕ-nodes are produced. However, reducing the number of ϕ-nodes slows down the calculation of SSA form. We thus have to choose the SSA version that best fits our needs in this trade-off between the speed of SSA construction and the speed of client analyses thereafter.

The first SSA version is called *minimal* [8], although it produces the most ϕ-nodes among the versions discussed here. It inserts such nodes at every control-flow join for all variables for which different definitions reach that join. This sometimes inserts nodes that are not live, i.e. the new definition introduced by such nodes is not used anywhere. These superfluous ϕ-nodes are removed in the so-called *pruned* SSA version. However, to achieve that, a solution to the live-variable problem must be computed and thus constructing pruned SSA form takes longer than the construction of minimal SSA version.

A compromise was invented by Briggs et al. [3] with the *semi-pruned* SSA version which removes only those ϕ-nodes that belong to variables which are not live at the beginning of *any* basic block. Determining these variables is much cheaper than solving live-variable analysis, and especially compiler-generated variables are only live within a single basic block and are thus already ruled out by this simple strategy.

These SSA versions were all described as intraprocedural analyses; in contrast, our ISSA analysis is interprocedural, respecting data flow across procedure boundaries.

2.1. Related work

The work that is closest to ours is from Liao [20]. In contrast to his work on ISSA, we include must-def side-effect computation, different pointer analyses, optimizations like pruning ISSA form and comprehensive test results.

Besides SSA form, other data structures were proposed to capture the data flow in a program. For example, Singer [27] discusses static single information form (SSI), which is a symmetric extension of SSA: if there are uses of a variable in different alternative branches of the CFG, then this intermediate representation introduces new definitions for every branch. Muchnick [23] describes Webs, which are maximal unions of def-use chains sharing a common use.

Ottenstein introduced program dependence graphs (PDGs) [24] for the use in software development, debugging and compilers. Those graphs represent statements and predicates of a subprogram as vertices and dependencies as edges. This permits the formulation of advanced program analyses such as slicing [33] as simple graph traversals. Considerable work was done that deals with the construction of those graphs. It is obvious that once the data (and control) dependencies have been established (e.g. in SSA form), the construction of PDGs is simple. Krinke [18] presented a construction method that applies dataflow analyses to establish the def-use relations. Another approach [11, 12, 21, 22] uses syntax directed methods and interval analysis but has limitations with unstructured control flow and can only provide rudimentary support for languages with pointers.

Horwitz et al. [15] developed system dependence graphs (SDGs) as an extension of program dependence graphs which represent a whole program. The general idea is to use PDGs for the representation of each subprogram and link those graphs to model the effects of subprogram calls. For call sites a call edge to the corresponding subprogram is inserted. Parameter handling is made explicit by special vertices for formal and actual parameters which are connected by copy-in and copy-out edges. This modelling is quite common in interprocedural program analyses and is also used in our ISSA analysis. Also, we use the same concepts for the handling of global variables which are treated as parameters of subprogram calls.

Horwitz also describes how calling contexts with different alias configurations can be distinguished. With our framework we are not only able to model contexts for different alias patterns but for arbitrary criteria which might have an effect on the data dependencies (e.g. constraints on values of actual parameters or restrictions on the flow of control in the called subprogram).

Much work was done that shows how to build dependence graphs in the presence of pointers [6, 14, 19] but the presented methods are always specific to one pointer analysis and use restricted languages, e.g. pointers may only refer to heap objects. In contrast, our framework can be used with arbitrary pointer analyses and is independent from a specific source language. We already support full ANSI C and C++. Frontends for Java are currently being integrated.

Hind identifies characteristics of pointer analyses (flow-sensitivity, context-sensitivity, heap modeling, aggregate modeling) in [13]. But until today their impact on scalability as well as precision of client analyses is not fully understood. Ryder et al. compare the results of side-effect analyses based on flow and context sensitive pointer analyses and argue that for some applications sensitivity might be required [26, 31]. Our framework generically supports all characteristics of Hind and does not impose any limitations on the accuracy of the base analyses. This will help us to gain more insight into the relationship between base and client analyses, and it will enable us to optimally select a base analysis for the application requirements.

2.2. Applications of ISSA

We motivate the consideration of ISSA by presenting some of the current and intended applications in software reengineering.

SSA forms can be utilized to find *anomalies in the usage of variables*, e.g. unreferenced variables, uninitialized variables, and unnecessary assignments where the assigned value is never read. Many compilers compute intraprocedural SSA forms and give warnings when such situations are detected. But the results are necessarily imprecise because only one routine is analyzed and no accurate information about the side effects of calls is available. In this paper we describe ISSA form that helps to precisely detect such anomalies for the whole program.

The data-flow information stored in the ISSA form can be used to *navigate the source code* and other program representations and helps in gaining a better understanding of the source code and its inherent relations. The implementation of *program slicing* in Bauhaus is based on the ISSA form. Backward slicing allows us to identify all statements that affect a given variable at a certain location. Forward slicing yields all statements that are affected by a certain statement. Both algorithms perform interprocedural traversals of the ISSA form and take summary edges for the precise handling of calling contexts into account.

Data-flow information can not only be utilized for debugging and low-level *program understanding*, but also to extract *global design information*. Data dependencies between two components reveal communication and show that the components interact. Moreover, component recovery techniques [17] can benefit from taking data-flow information into account by grouping elements with high cohesion into the same component [5].

Client analyses often suffer from inaccurate results produced by flow-insensitive base analyses. Those results can be improved by additionally taking data-flow edges back to an allocation site into account. For example, the extraction of object trace graphs for protocol recovery requires locating all operations on one specific object [10]. Especially for heap objects, this analysis strongly depends on the quality of the pointer analysis. With the help of ISSA form, we can improve flow-insensitive pointer information, *improving both precision as well as efficiency* of the client analysis. The computation of the ISSA form itself also suffers from imprecise pointer analyses because it uses an overestimation of actual accesses to variables. After a first computation of ISSA form, one can try to gain optimized points-to information from the ISSA form and use this in a recalculation of ISSA form. Before as well as after the recalculation, the ISSA form is consistent and conservative, thus this scheme can be iterated as often as desired, e.g. until the improvement is below a specific value.

Our static analysis of graphical user interfaces (GUI) uses data-flow information to determine which expressions designate the same widgets [28]. This is needed to detect the widget hierarchies of which an application's windows consist. Similarly, data-flow information is used to associate events and attributes with widgets in these hierarchies. The ISSA form described in this paper is used to infer such data-flow connections.

3. Overview of ISSA

Classical SSA form only handles local variables. ISSA is an extension which additionally also handles global variables and subprogram parameters, respecting the data flow across procedure boundaries. ISSA as described in this paper stays close to intraprocedural SSA in that we basically treat global variables and parameters like local variables. To achieve this, we need to know the side effects a subprogram may transitively (that is, including the effects caused by subprograms called within that subprogram and so on) issue on a global variable. Then we can respect the data flow across procedure boundaries with special actions before and after call sites and at the beginning and end of subprograms. Section 6 describes this aspect in more detail.

However, an interprocedural analysis has to cope with additional problems. First of all, since we need calling relationships between subprograms, ISSA construction is based on the callgraph and thus suffers from imprecision related to function pointers and calls to virtual functions. Moreover, a well-known problem for interprocedural analyses are unrealizable paths that might be present in the interprocedural control-flow graph. Other problems related to control-flow analysis include explicit halts and exception handling which introduce further complexity. The ISSA algorithm relies on the results of control-flow analysis and is therefore directly affected from how that pre-analysis copes with the problems mentioned and how the results are represented.

Another problem arises from pointers and aliasing "hiding" the elements affected by a statement. To achieve good precision, it is therefore necessary to cope with pointers and aliases. However, pointer analysis is a complicated topic of its own and until today there is no pointer analysis clearly superior to other pointer analyses. Rather, it seems like different pointer analyses are needed for different programs and analysis goals. Creating ISSA form should thus be possible with many different pointer analyses.

A further improvement in the precision can be achieved if we allow a subprogram to be analyzed in different contexts. That is, the ISSA algorithm should have the option to be context-sensitive. Since the value of such a context-sensitivity depends on the pointer analysis used, this can be seen as a noteworthy part of the requirement that we should be prepared for different pointer analyses.

Finally, another difficulty arises from arrays and structures. These composite data types make it hard to decide which part of the storage is actually modified or used. Again, this is a topic for pointer analysis that must be considered when designing ISSA analysis.

The following sections provide a detailed description of our framework for ISSA analysis which copes with these problems. The steps that must be executed for ISSA generation can in short be summarized as follows:

1. A frontend extracts an annotated abstract syntax graph for the full source program.

2. A local control-flow analysis creates intraprocedural control-flow graphs for all subprograms.

3. A pointer analysis is executed that provides approximations for the effects of pointers and determines the targets of indirect or dispatching calls.

4. A program representation with abstract variables and instances of subprograms distinguishing different contexts is created.

5. May-def and may-use side effects are computed.

6. Data-flow edges for parameters and global variables are inserted at call sites.

7. Intraprocedural SSA form is created for every context.

8. Optionally, ISSA form is pruned and must-def side effects are determined.

4. Abstract program representation for ISSA

ISSA analysis can be combined with different pointer analyses, e.g. those implemented in Bauhaus: Steensgaard [30], Das [9], Andersen [1] and Wilson [34]. The program representation which allows this is conceptually an interprocedural control-flow graph (ICFG) with the special feature that the CFG for a single subprogram can appear multiple times. That is, we consider *instances* of subprograms. These instances are used to distinguish different contexts in which some subprogram appears. For example, Wilson's pointer analysis computes summaries per alias pattern among the parameters of a subprogram. In our model, Wilson's analysis therefore creates a new instance of the subprogram for every such alias pattern. Note that we can also handle recursive subprograms for which pointer analysis uses an approximation to avoid infinitely many replications. We call our variant of the ICFG a *context-sensitive control-flow graph* (CS-CFG) and we call the subprogram instances *contexts*.

As we can see, the CS-CFG used in our model depends on the pointer analysis: pointer analysis determines which contexts are created and how they are connected. Also, different pointer analyses store different results and additional data per context; that is, the internal data structure for contexts is determined by the pointer analysis, too. More details on this are given in Section 9.

Using instances of subprograms instead of the original subprogram is one part of our abstraction. The other part concerns the variables in the program: we replace original variables with abstract variables called *locators*. ISSA analysis considers locators to be atomic; that means they do not have an internal structure and do not consist of parts. The key property of these locators is that they do not overlap, at least locally: the memory locations designated by a locator are only accessed via this locator; other locators are no aliases, not even for parts of the memory regions. This is enforced locally, i.e. for all locators in the same context. We do not enforce this globally since we want to treat global variables later on like local ones, using different locators for one global variable in different contexts. The purpose of the restriction of (local) alias-freeness is that definitions of a locator will have no effect on other locators. Other restrictions are not imposed by our ISSA framework.

Creating locators is based on the pointer analysis but is a separate phase that can be reused for similar pointer analyses. We can freely choose to collapse several original variables into one locator or to split structures into several locators, so long as the key property of local alias-freeness is guaranteed. A locator can be used to represent any kind of memory elements, like variables and heap objects. The next section gives more details about the generation of locators and the CS-CFG in general.

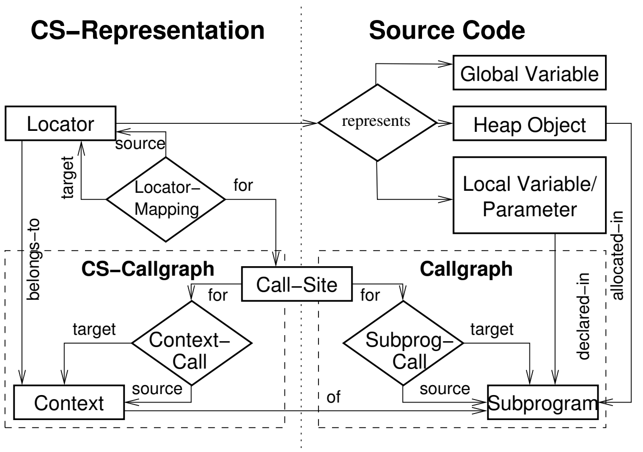


Figure 1. Abstract program representation

ISSA construction also requires that the abstract program representation distinguishes uses and definitions of locators. It is, however, enough to provide may-use and may-def accesses, stronger "must" information is not needed (and can be inferred later, after ISSA construction; see Section 7). Providing these access kinds for directly visible effects is straight-forward. However, for function calls we can not easily decide which locators are affected and in which way. It is therefore also necessary to determine may-use and may-def side effects to come up with a valid program representation for ISSA construction.

Figure 1 shows a summary of our representation. On the right hand side, it shows the relevant entities of the source code, like subprograms and variables. On the left we can see the corresponding entities in our context-sensitive representation. Here, locators are used as an abstraction for all sorts of variables, and contexts are used instead of subprograms. The figure also contains the locator mapping needed in general to map locators between callers and callees.

5. Creating the program representation

This section describes how we generate the program representation for ISSA construction sketched in the previous section. The description given here is based on context- and field-insensitive pointer analyses. Remarks for other pointer analyses can be found in Section 5.3.

5.1. Locators for local accesses

Context-insensitive pointer analyses do not distinguish different contexts for subprograms and thus create a CS-CFG containing exactly one instance of every subprogram. In this case, the CS-CFG is identical to the standard ICFG. We can therefore concentrate on the description of how locators are generated for pointer analyses like the one by Steensgaard. In fact, our locator generator is not part of the pointer analysis and is thus applicable to other, similar analyses, namely those of Das and Andersen.

The easiest way to create locators would be to create one locator for every equivalence class of variables determined by Steensgaard's analysis. This strategy easily meets our requirement that no two locators in the same context are aliases. Another advantage is that all accesses in the program then refer to exactly one locator. However, this generation of locators for each equivalence class is quite imprecise. The variables can not be further distinguished, e.g. when one variable is directly accessed.

We call this strategy *equivalence-class sensitive*. Our abstract program representation allows other strategies to represent variables (or parts) by locators, listed here in increasing order of precision:

1. equivalence-class sensitive – one locator for multiple variables
2. variable sensitive – one locator for each variable
3. field sensitive – one locator for each non-aliased part of a variable

Note that the term *equivalence-class sensitive* is used to capture all possible strategies of using one locator for multiple variables. It therefore also captures the method of only assigning locators to local aliases: locators are like local variables of contexts and the requirement of being no alias to another locator is only needed within a context.

We can achieve better results by generating one locator for each variable if we make the assumption that no variables overlap in memory. Now each statement refers to those locators which might get accessed, in general more than one locator. We used this strategy for the generation of our test results (Section 8) as it can easily be adjusted to all of our context-insensitive pointer analyses.

Field-sensitive locators finally allow an even better precision by generating one locator for each non-aliased part of a variable.

5.2. Side effects

A prerequisite for the ISSA algorithm is the knowledge about side effects on global variables for every subprogram. For each subprogram the set of all global variables that it may define or may use, including transitive side effects occurring inside a call issued in the subprogram's body is calculated. Some pointer analyses like the one by Wilson compute these side effects on their own, but we need a separate analysis for other analyses like the one by Steensgaard. This section describes our algorithm for the latter case.

```
procedure Visit (V : Context) is
begin
   -- local (non-transitive) side effects
   Compute_Local_Side_Effects(V);

   Push(V);
   DFS[V] := Head[V] := Current_DFS;
   Current_DFS := Current_DFS + 1;
   Visited[V] := True;

   forall S ∈ Succ(V) loop
      if not Visited[S] then
         Visit(S);
         Head[V] := Min(Head[V], Head[S]);
         -- propagate side effects of S to V
         Update_Side_Effects(V, S);
      elsif Is_On_Stack(S) then
         Head[V] := Min(Head[V], DFS[S]);
      end if;
   end loop;

   if Head[V] = DFS[V] then
      -- V is the head of a SCC
      loop
        W := Pop();
         exit when W = V;
        -- propagate side effects of V to W
        Update_Side_Effects (W, V);
      end loop;
   end if;
end Visit;
```

Figure 2. Computing side effects

Collecting the side effects issued locally in a subprogram (i.e., without those effects inherited from calls) is a straightforward process: a simple intraprocedural analysis walks over all statements in the body and inspects them for such effects.

Additionally to those local side effects we propagate effects from callees to callers. This proposes a postorder traversal of the callgraph; however, we have to cope with cycles in that graph in general: a simple postorder traversal would potentially miss some side effects and therefore would require iterations to achieve correctness. But we note that a cycle is in fact a simple case: all members of a cycle transitively issue the same side effects, namely the union of all local side effects of all subprograms in the cycle, and those effects inherited from callees that can be reached from anywhere in the cycle. Therefore, Tarjan's algorithm [32] to detect cycles is our basis for side-effect computation without iterations. Whenever it detects a cycle, we ensure that all members have the same set of side effects (which is known at that time as the side effects of the cycle's root determined by the algorithm).

Figure 2 presents our algorithm. It shows the core routine of the recursive depth-first search on the callgraph corresponding to the CS-CFG. The only modifications to Tarjan's algorithm are the additional invocations of `Compute_Local_Side_Effects` and `Update_Side_Effects`. The first of these determines local side effects, and the second adds side effects propagated from the context given as second argument to those of the first argument. We call these propagated locators *nonlocals*. Nonlocals include (for C-style programs) locators for global variables and for local variables of other subprograms passed by reference (these are detected by pointer analysis and thus occur in points-to sets).

A call `Update_Side_Effects (V,S)` will also generate new locators for nonlocals which are now visible in context `V` after the side effects of `S` have been added. Note that, to be conservative, whenever a context has a may-def side effect on a variable, it also has a may-use side effect on it. This is needed because later the may-def will result in a ϕ-node which potentially uses some definition from outside the subprogram.

The algorithm visits all edges in the callgraph once and all nodes at most twice. Computing local side effects can be distributed over the frontend (for direct accesses to globals) and pointer analysis and therefore adds no costs here. Updating side effects due to propagation requires at worst G insert operations per update where G is the number of nonlocals. Inserting entries into a set can be done in constant time for example with the data structure proposed by Briggs and Torczon [4]. If we have F contexts and E call edges in the CS-CFG's callgraph we therefore have total costs of $\mathcal{O}(G*(F+E))$. Compared to the classical side-effect algorithm by Cooper and Kennedy [7], we have a simpler algorithm because we computed points-to sets interprocedurally in advance, respecting reference parameters there. As our measurements in Section 8 indicate, our algorithm is fast in practice even for large applications. Computing side effects has thus total costs linear in the size of the CS-CFG, which in our example equals the size of the ICFG. The algorithm is thus linear in the program size.

The propagation is simple and fast since we are only interested in "may" side effects which are never killed in a caller and thus finally propagate to the callgraph's root. This first estimation of side effects is later improved in two ways (cf. Section 7): first, stronger "must" side effect information can be inferred in some cases and second, we can then also delete ("prune") some propagated side effects.

5.3. Other pointer analyses

The above description captured pointer analyses with a rather low precision. Our framework, however, is far more general and can deal with many other pointer analyses. Going to field-sensitive analyses, for example, is simply a matter of locators, and going to context-sensitive analyses can be done by creating several instances of a subprogram in the CS-CFG. In Bauhaus we are working on an implementation of Wilson's very precise analysis which is field- and context-sensitive and thus exploits the power of our program representation.

It is beyond the scope of this paper to also describe the creation of the program representation for other pointer analyses. Therefore, let us simply note that this creation process can of course reuse some parts of the steps described above for context-insensitive analyses. It is also possible that some steps are not needed: Wilson's analysis for example also determines side effects and thus does not need an additional side-effect analysis.

6. Interprocedural SSA form

We have now computed a program representation consisting of our CS-CFG and locators, respecting transitive side effects. The next steps in the analysis chain outlined in Section 3 compute ISSA form and are described here.

6.1. Nonlocal locators and parameters

We ultimately want to treat nonlocal locators and subprogram parameters like locals. For this to work, we have to separately take care of the data flow across procedure boundaries caused by these entities. *Nonlocal* locators (for C-style programs) include global locators and locators for local variables of other subprograms passed by reference.

The first step converts nonlocals to parameters. Since we know about transitive side effects of all subprograms, this is a straight-forward process: if subprogram `f` may define a nonlocal `g`, `g` becomes an out-parameter of `f`; and if `g` is potentially used, `g` becomes an in-parameter.

As shown in Figure 3, those artificial parameters are represented by pairs of Pre_Call- and Post_Call-nodes at the call site and Link_In- and Link_Out-nodes for in- and out-parameters of a context. Link_In-nodes represent a set of ϕ-nodes summarizing all in-parameters of all calls to the same context, while Link_Out-nodes act as sets of ϕ-nodes summarizing the out-parameters of a call. Inside of a context, Link_In-nodes are treated as artificial definitions and Link_Out-nodes as artificial uses.

Note that in general a $m : n$ mapping between locators of the caller and the callee is necessary. Thus the actual implementation of that mapping is not always as obvious as shown in the example.

6.2. Creating SSA form

After preprocessing the program in this way, we can construct SSA form for every subprogram (in the CS-CFG, that is, in all contexts) in the standard way [8]. Our implementation for intraprocedural SSA follows the classical approach and consists of two phases: determine places for ϕ-nodes using iterated dominance frontiers and rename variables. Note that may-def accesses on locators will result in ϕ-nodes combining the definition with the previous one.

Paths to a may-use which bypass all definitions (producing an uninitialized variable) receive an artificial must-def initialization in the Link_In-node as usual in SSA forms. For uninitialized nonlocals this artificial initialization is automatically inserted in the callgraph's root because the may-use side effect was propagated. Later, pruning is able to remove those artificial initialization for nonlocals that are unnecessary (see Section 7).

6.3. Complexity

Handling a parameter takes constant time for the artificial new assignments and for every call site relevant to the parameter. Assuming that the number of parameters per subprogram and the number of locators per variable are bound by a (typically small) constant, we can bound the total costs for parameters by $\mathcal{O}(F + E)$ where F is the number of contexts in the CS-CFG and E is the number of call edges therein. That is, parameter handling is linear in the size of the CS-CFG's callgraph. Note that the number of contexts in a CS-CFG depends on the pointer analysis. For context-insensitive analyses there is exactly one context per subprogram, making parameter handling linear in the size of the program's callgraph. However, more precise analyses might need a number of contexts exponential in the program size.

For a nonlocal we need constant time for each subprogram, which has a transitive side effect on it. Additionally, we have to add constant time per call site for such a subprogram. The costs for nonlocals can therefore be bound by $\mathcal{O}(G * (F + E))$ where G is the number of nonlocals. Finally, we run the classical SSA algorithm for all contexts in the CS-CFG. If we denote the costs for classical SSA with S, this adds $\mathcal{O}(F * S)$. Therefore, ISSA construction has total costs of $\mathcal{O}(F + E + G * (F + E) + F * S) = \mathcal{O}(F * (G + S) + G * E)$. Note that S, the costs for intraprocedural SSA analysis, depend on the number of local variables, i.e. the number of locators used in the subprogram instances.

7. Optimizations

In this section we consider some optimizations for side-effect information and the ISSA form. Our algorithm converts may-def accesses into must-defs wherever possible and adjusts side effects accordingly. Additionally, it removes definitions that are not needed because their result is never used. This optimization for classical intraprocedural SSA is known as *pruning*.

For example, in Figure 3(a), line 9, the assignment to the variable `g` is a definitive must-def. In the caller the values represented by locators `main.g` and `main.e` are not used, therefore the assignment to `g` is marked to be pruned.

```
1  #include<stdlib.h>
2
3  int g;
4
5  void f(int *x, int *y)
6  {
7      *x = *y;
8      if (...) *y = 1;
9      g = *x + *y;
10 }
11
12
13 int main(int argc, char *argv[])
14 {
15     int *p;
16     int a = 1;
17     int b = 0;
18
19     f(&a, &b);
20
21     if (...) p = &a;
22     else     p = malloc(sizeof(int));
23
24     f(p, &a);
25
26     return a - b;
27 }
```

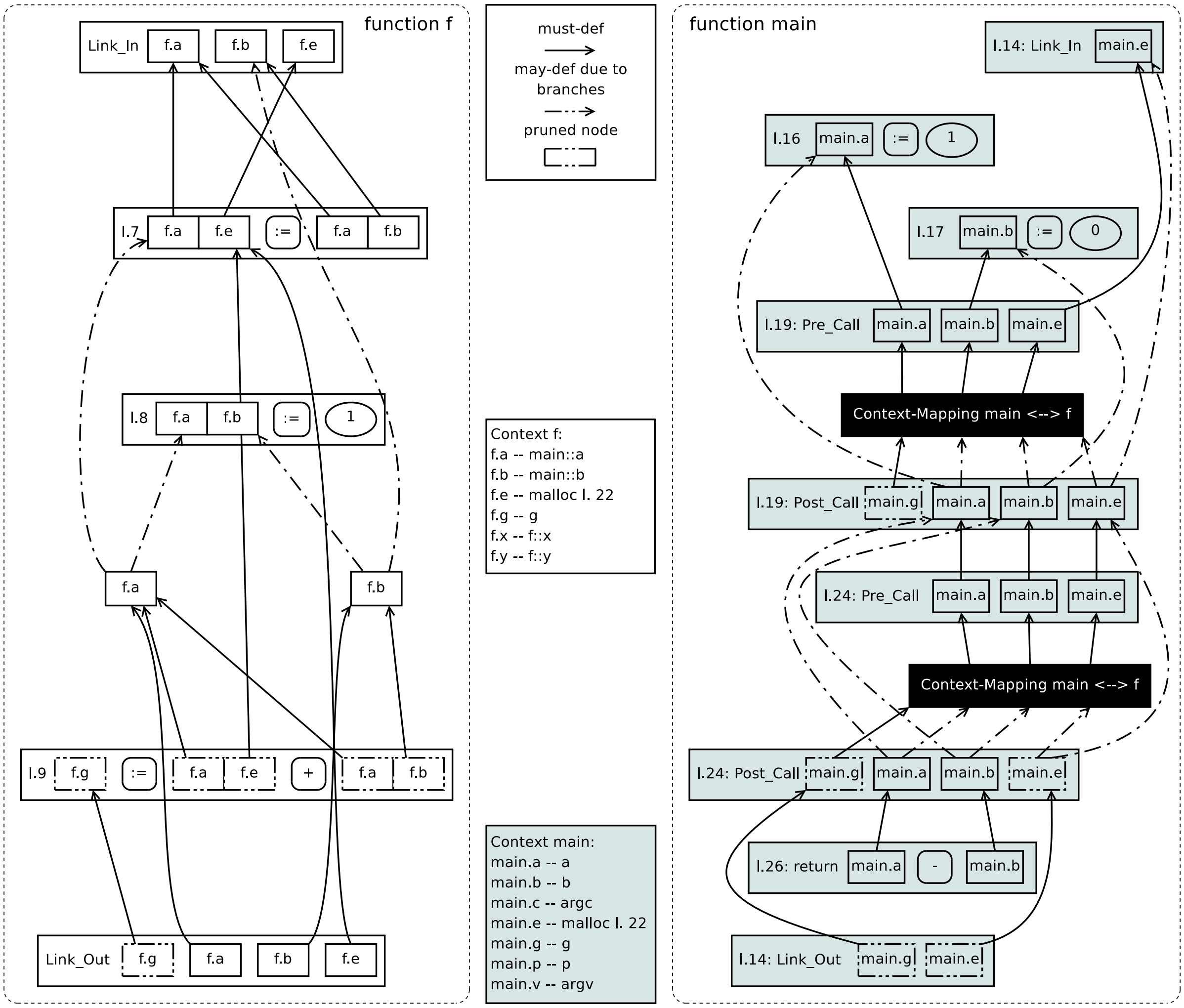
(a) Example program

Context and flow-insensitive Points-To analysis

Points-to sets	
pointer	targets
main::p	main::a, malloc@l.22
f::x	main::a, malloc@l.22
f::y	main::a, main::b

Locators and Context-Mapping for both calls	
context main	context f
main.a	f.a
main.b	f.b
main.c	
main.e	f.e
main.f	
main.g	f.g
main.p	
	f.x
	f.y

(b) Points-To results



(c) ISSA graph

Figure 3. Example ISSA graph

Pruning for intraprocedural SSA requires to solve the standard problem of live variables. In contrast, our implementation builds on ISSA results and is a simple and fast marking algorithm. The complete optimization algorithm is shown in Figure 4. It operates on the graph of definitions and uses produced by ISSA analysis and has linear time complexity in the size of this graph. The following sections explain it. Notice that steps 1 and 2 can be executed independently, but step 2 will benefit from step 1.

```
procedure Optimize is
begin
    -- step 1a: convert may-def to must-def
    forall $m \in$ May_Defs loop
        Convert_To_Must_Def_If_Possible(m);
    end loop;

    -- step 1b: propagate must-defs
    Contract_Cycles;
    forall $m \in$ Must_Defs loop
        Mark_Must_Def(m);
    end loop;
    Expand_Cycles;

    -- step 1c: convert may-def side effects to must-def
    forall $\ell \in$ Link_Outs loop
        if Marked($\ell$) then
            Context($\ell$) has a must-def side effect
        end if;
    end loop;

    -- step 2: prune unnecessary nodes
    forall $m \in$ non-artificial uses loop
        Mark_Nodes_In_Use(m);
    end loop;
    forall $v \in$ Nodes loop
        if not In_Use(v) then
            Remove v;
            if $v \in$ Link_Ins then
                Remove may-use side effect from Context(v)
            end if;
        end if;
    end loop;
end Optimize;

-- mark nodes only reachable via must-defs
procedure Mark_Must_Def(m) is
begin
    return when $\exists\, p \in$ pred(m): not Marked(p); -- in O(1)
    Marked(m) := True;
    for all $s \in$ succ(m) loop
        Mark_Must_Def(s);
    end loop;
end Mark_Must_Def;
```

Figure 4. Optimization algorithm

7.1. Detecting must-def accesses

ISSA construction as described in Section 6 uses only may-def accesses on locators. Step 1a of the algorithm in Figure 4 now first improves the results in that we identify assignments which were classified as may-defs but are in fact must-defs. Whenever we find such a case, we can replace the ϕ-node created for the original may-def with a direct definition, removing an outgoing edge from the previous definition.

Determining must-def accesses in general has to be conservative. A must-def should only be recognized if all previous definitions are overwritten. Therefore, we restrict ourselves here to the description of two cases in which a must-def can be safely identified. These cases are handled by the function `Convert_To_Must_Def_If_Possible` which is not shown in the figure.

The first case occurs when we find a direct assignment to some variable. If the assignment is a full update of the variable (it might be not for structures), we can conclude that it is a must-def access. This case produces most of the must-defs.

The second case occurs when we find an assignment to some pointer target where points-to analysis determined that there is exactly one target. But since points-to information is actually only may-point-to information, we have to check additional requirements here to be sure that it is indeed a must-def.

The first of these additional requirements is that the target is not an abstraction for some allocation site, since otherwise it could have been used to summarize several heap objects. The second additional requirement is that the target should not be a local variable of some function within a cycle in the callgraph since otherwise many instances of this variable might exist at the same time while the assignment only touches one of them. And finally the assignment again has to be a full update. Only if all requirements are fulfilled we can conclude that the assignment is a must-def.

We assume that some previously executed analysis has marked those subprograms that are part of a cycle. Then every call to `Convert_To_Must_Def_If_Possible` takes only constant time.

7.2. Propagating must-def side effects

After the detection of must-defs, we check whether some subprogram now has a must-def side effect. This is true for all subprograms for which a may-def side effect was estimated and which have no control-flow path lacking a must-def. This is equivalent to the condition that all paths in ISSA form starting at a Link_Out node (see Section 6.1) contain a must-def before reaching the Link_In-node.

Step 1b of the algorithm therefore now marks those nodes that can only be reached on paths containing a must-def (shown in `Mark_Must_Def` in Figure 4) and step 1c changes a subprogram's may-def side effect into a must-def if its Link_Out node is marked. Since ISSA form is interprocedural, this also captures the propagation of such side effects.

Cycles in the ISSA form are a small problem in that they would always stop the marking, but they can be contracted because they behave like a single ϕ-node: all members of the cycle have to be marked if and only if all predecessors of the contracted cycle are marked. When expanding the cycles again, all members receive the marking state of the contracted node.

7.3. Pruning ISSA form

The final step then is to prune superfluous nodes, including ϕ-nodes. For this, we simply mark nodes that are used somewhere and remove those that are left unmarked.

Our approach is simpler than standard interprocedural live-variable analysis because we make use of ISSA results. We traverse all uses of a variable, except arguments of ϕ-nodes and the artificial uses introduced in Post_Call-nodes (see Section 6.1) at call sites. Optionally, Post_Calls in the callgraph's root can be treated as uses so that may-def and must-def side-effect information is not deleted. For every use, we mark the corresponding definition as live; if that definition is a ϕ-node or an artificial node for data flow across subprogram boundaries, we also mark all predecessors (definitions) for the arguments as live, and so on. That is, the function `Mark_Nodes_In_Use` (not shown in Figure 4) is simply a depth-first search from nodes to predecessors, setting `In_Use` for every yet unseen node it visits.

At the end, we can remove those definitions and artificial nodes that are left unmarked. If we remove a Link_In node in this pass, this means that the subprogram has no longer a may-use side effect. Since the conversion of a may-def into a must-def disconnects the node from its predecessor, this correctly prunes artificial nodes and may-use side effects and also handles the propagation of these side effects.

8. Statistics and test reports

We implemented and tested ISSA analysis in our Bauhaus project. Table 1 shows the applications which we used as test suite. For all applications we give the source lines of code (column *sloc*) as measured with the SLOC-Count utility[1]. Moreover, the number of variables (including parameters) is shown in column *vars* and the number of global variables among them is presented in column *globals*. The remaining columns list the characteristics of the callgraph.

We ran our implementation against the test suite on our Linux machine (4 Intel Xeon processors with 3 GHz and 16 GB RAM) under normal system load. The results are presented in Tables 2 and 3. Both tables have three rows per application of the test suite: for using Steensgaard's pointer analysis (row *ecr*), the one by Das (row *das*) and the one by Andersen (row *and*).

Table 2 on page 14 lists measurements on the intermediate representation. The first column shows the number of locators. Then follow three columns on definitions: the total number of definitions in the program, the number of must-definitions, and the number of definitions that could be pruned. Then follows one column showing the number of all uses of a locator in the program. Last, there are two columns on ϕ-nodes, the total number of ϕ-nodes, and then the number of ϕ-nodes that could be pruned. The numbers of pruned definitions and pruned ϕ-nodes also include the percentages relative to the total number of definitions or ϕ-nodes, respectively.

In Table 3 on page 16, runtime and memory consumption are listed for side-effect computation including locator generation, ISSA analysis, and the optimization phase. As runtimes we measured the user time. Currently, in our implementation the must-def computation is included in ISSA analysis, and therefore the time needed for computing must-defs and must-def side effects is included in the runtime of ISSA construction.

If we inspect the runtimes we can observe that generating locators along with may-def and may-use side effects is faster than ISSA construction and also very fast in absolute numbers. Both ISSA construction and optimization are also fast with pruning being rarely slower than one minute. As could be expected, the number of locators directly influences all runtimes. Often, but not always, Steensgaard's analysis creates more locators than the other pointer analyses. This can be explained with larger points-to sets for may-use and may-def accesses and more call edges for indirect calls (leading to more propagated side effects) produced with this analysis. The number of ϕ-nodes increases with the number of definitions which itself increases with the number of locators. This means that an imprecise pointer analysis creates more locators, more definitions, and more ϕ-nodes: the size of ISSA form increases inversely with the precision of pointer analysis. Often, the relative percentage of pruned nodes is highest for Andersen's analysis, but there are some exceptions.

As we can see, our optimizations are worth the effort. Pruning is able to remove substantial amounts of nodes in many cases. This reduces memory consumption and runtime for client analyses. Moreover, we were able to detect many must-defs which improves the precision of ISSA form and side effects and also helps to prune even more nodes.

9. Implementation details

We implemented the ideas described in the previous sections in our Bauhaus tool suite. Whereas the previous sections gave a conceptual description of our modelling and algorithms, we now report on some implementation details that are important to realize the concepts.

---

[1] generated using David A. Wheeler's SLOCCount

name	sloc	vars	globals	routines	direct calls	indirect calls
Astro/Astro	5393	534	358	310	429	0
SNNSv4.2/snns2c	83027	152	11	64	509	0
bash-3.1/bash	88655	3256	567	1433	7022	37
bc-1.06/bc	8526	442	94	177	1038	21
bison-2.3/bison	23412	2601	274	1164	3326	209
bluefish-1.0.5/bluefish	40765	3066	210	1590	15919	18
codebreaker-1.2.1/codebreaker	1220	106	15	62	609	0
concepts-0.3f/concepts	3948	388	53	216	651	1
cook-2.26/cook	63099	2053	344	1239	5087	37
dia-0.95.0/dia	123035	7005	667	5566	17602	596
euler-1.60.6fix/euler	24056	3643	412	1029	6988	76
gnuplot-4.0.0/gnuplot	68611	5005	1293	2145	12509	539
gqview-2.0.1/gqview	52998	5557	297	3266	18073	48
grep-2.5.1a/grep	20846	628	132	226	1078	11
gzip-1.3.9/gzip	8606	462	152	243	706	4
make-3.75/make	17424	959	134	410	2146	1
nano-1.2.3/nano	9903	627	82	283	2309	2
screen-4.0.2/screen	37618	1987	250	775	4660	51
sed-4.1/sed	21739	991	72	299	1796	1
soundtracker-0.6.7/soundtracker	33049	2022	565	1139	8620	106
tar-1.16/tar	47249	3162	861	1525	5243	50
tcc-0.9.23/tcc	42390	1156	91	341	2517	3
tcsh-6.15.00/tcsh	50506	2763	677	1421	7657	27
time-1.7/time	1395	57	22	35	141	2
trueprint-5.3/trueprint	8313	459	259	260	1246	25
units-1.86/units	2837	250	50	89	630	3
unzip-5.52/unzip	52701	918	394	272	1240	317
uucp-1.07/uucico	53730	2074	349	1488	4568	78
wget-1.10.1/wget	25925	1724	182	752	3994	23

Table 1. Our test suite

Conceptually, we have (potentially) different contexts for every subprogram in our CS-CFG. However, the implementation does not duplicate the abstract representation for subprograms. Instead, we use a sparse model and annotate all uses and definitions of variables in the subprogram's representation with an array of `Def_Tables`. In this array, there is one `Def_Table` for each context in which the definition or use may be executed.

Each `Def_Table` provides a mapping from a locator to the set of possible previous definitions that might reach the definition or use. If a definition is a strong (must-) update then the `Def_Table` is empty. In contrast, for a weak (may-) update, the `Def_Table` links to the previous definitions that may or may not still be live after the definition. The `Def_Table` also holds a flag `Live` for each locator; it is set if the reaching definition is needed and thus cannot be pruned.

9.1. ISSA data structures

Our representation uses a number of different classes of nodes to represent definitions and uses:

Begin_Of_Lifetime A node is inserted at the point of the CFG where a local variable's lifetime begins. This node acts as an artificial definition. Its purpose is to provide a definition for otherwise uninitialized locators. Whenever a previous definition in some `Def_Table` references a Begin_Of_Lifetime-node then that variable might be uninitialized.

Assignment Nodes of the class Assignment represent all explicit and implicit assignments to variables including initializations. An Assignment-node acts only as a definition.

Join_Phi A Join_Phi-node is inserted at control flow join points where different definitions of at least one locator meet. A Join_Phi-node provides the possible previous definitions for all such locators within a single `Def_Table`. It also provides information to determine in which predecessor basic block (eg. true/false-predecessor after if-statements) each definition occured.

Read A Read-node represents the use of the value of an object. In its `Def_Table` several locators might be present (because a variable might consist of several locators, or because points-to analysis cannot determine which object is actually used), but each of those locators has exactly one reaching definition.

Pre_Call A Pre_Call-node is inserted directly before a call to a subprogram. It summarizes the reads of all actual parameters in one node.

Link_In A Link_In-node represents an artificial definition of a locator that represents a nonlocal. This is the first node in every subprogram's CFG, and in its `Def_Table` there are never any previous definitions annotated. During pruning however, the Link_In acts as a use to the actual parameters. So if all locators from the callee's context that map to one specific actual parameter are pruned, then that actual parameter can be pruned in its Pre_Call-node.

Link_Out The Link_Out-node is inserted as the last node in the CFG of a subprogram. It is an artificial read of all locators that may be side-effects for the subprogram.

Post_Call The Post_Call-node is inserted directly after a call. It represents the artificial definition of all locators in the context of the caller which the called subprogram might have modified. For definitive side-effects, the `Def_Table`'s reaching definition-entries reference no definition, for possible side-effects they reference the live definition in the local CFG, analogously to Assignment-nodes. During pruning, a Post_Call-node acts as a read to the Link_Out-node of the called context to propagate liveness into the called context.

9.2. The context-implementation

In our implementation, locators are modelled as integers that are unique within one context. In order to support different points-to analyses, we model contexts as a class hierarchy. Each points-to analysis must provide an implementation of the abstract class `Context` and redefine methods that provide the mapping from objects in the source code to locators and the mapping between locators in different contexts.

Note that the unique numbers for locators can be used as indices into arrays to provide efficient access into the data structures. In fact the `Def_Table`s mentioned earlier are implemented as arrays indexed by the number of a locator.

10. Conclusions

In this paper we presented interprocedural SSA form as an extension to classical SSA. We described our framework which can be used together with many different pointer analyses thanks to an abstract program representation consisting of the CS-CFG and locators. Our ISSA analysis includes the computation of side effects. We explained a simple and fast algorithm for optimizations; it computes must-def side effects and creates a pruned ISSA form. Finally, we reported results for a large test suite.

References

[1] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[2] A. W. Appel. SSA is Functional Programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[3] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software Practice and Experience 28(8)*, pages 859–881, July 1998.

[4] P. Briggs and L. Torczon. An Efficient Representation for Sparse Sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, 1993.

[5] D. L. Carver and R. Valasareddi. Object localization in procedural programs: a graph-based approach. *Journal of Software Maintenance*, 12(5):305–323, 2000.

[6] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310. ACM Press, 1990.

[7] K. D. Cooper and K. Kennedy. Interprocedural Side-Effect Analysis in Linear Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–66. ACM Press, 1988.

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems 13*, pages 451–490, 1991.

[9] M. Das. Unification-based Pointer Analysis with Directional Assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[10] T. Eisenbarth, R. Koschke, and G. Vogel. Static Object Trace Extraction for Programs with Pointers. *Journal of Systems and Software 77 (3)*, pages 263–284, 2005.

[11] M. J. Harrold, B. A. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. In *International Symposium on Software Testing and Analysis*, pages 160–170, 1993.

[12] M. J. Harrold and G. Rothermel. Syntax-Directed Construction of Program Dependence Graphs. Technical Report OSU-CISRC-5/96-TR32, Ohio State University, 1996.

[13] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM Press, 2001.

[14] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 28–40. ACM Press, 1989.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[16] R. A. Kelsey. A Correspondence between Continuation Passing Style and Static Single Assignment Form. *SIGPLAN Notices*, 30(3):13–22, 1993.

[17] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute for Computer Science, University of Stuttgart, 2000.

[18] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, 2003.

[19] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31. ACM Press, 1988.

[20] S.-W. Liao. *SUIF Explorer: An Interactive and Interprocedural Parallelizer*. PhD thesis, Stanford University, 2000.

[21] P. E. Livadas and S. Croll. System Dependence Graph Construction for Recursive Programs. *Computer Software and Applications Conference*, pages 414–420, 1993.

[22] P. E. Livadas and S. Croll. System Dependence Graphs based on Parse Trees and their use in Software Maintenance. *Inf. Sci. Intell. Syst.*, 76(3-4):197–232, 1994.

[23] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[24] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *ACM SIGSOFT/SIGPLAN*, pages 177–184. ACM Press, 1984.

[25] A. Raza, G. Vogel, and E. Ploedereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In *Reliable Software Technologies, Ada Europe 2006 (LNCS 4006)*, pages 71–82, 2006.

[26] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, 2001.

[27] J. Singer. *Static Program Analysis based on Virtual Register Renaming*. PhD thesis, University of Cambridge, Mar. 2005.

[28] S. Staiger. Reverse Engineering Graphical User Interfaces using Static Analyses. In *Proceedings of the 14th Working Conference on Reverse Engineering*. IEEE Computer Society, 2007.

[29] S. Staiger, G. Vogel, S. Keul, and E. Wiebe. Interprocedural Static Single Assignment Form. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE)*, 2007.

[30] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[31] P. A. Stocks, B. G. Ryder, W. A. Landi, and S. Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 21–31. ACM Press, 1998.

[32] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing 1 (2)*, pages 146–160, 1972.

[33] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[34] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 1995.

Table 2. Numbers of locators, definitions and uses

name	*pta*	*locators*	*defs*	*must defs*	*pruned defs*	*uses*	*ϕ-nodes*	*pruned ϕ-nodes*
Astro	and	2255	1131	968	632 (55.88%)	2232	439	155 (35.31%)
Astro	das	1793	1061	968	637 (60.04%)	1551	362	149 (41.16%)
Astro	ecr	2326	1341	968	615 (45.86%)	2484	469	139 (29.64%)
bash	and	100208	141320	7725	31182 (22.06%)	245075	110741	27407 (24.75%)
bash	das	293845	854442	7558	140038 (16.39%)	1325306	522392	92480 (17.70%)
bash	ecr	204572	508563	7795	58743 (11.55%)	911013	318323	46172 (14.50%)
bc	and	2957	11585	1303	881 (7.60%)	21079	6827	1108 (16.23%)
bc	das	2255	8410	1294	331 (3.94%)	18792	4525	692 (15.29%)
bc	ecr	6384	34709	1438	412 (1.19%)	78642	15010	811 (5.40%)
bison	and	27033	65180	4494	5307 (8.14%)	184878	36902	5197 (14.08%)
bison	das	30268	105797	4494	4921 (4.65%)	291504	54319	4737 (8.72%)
bison	ecr	34273	125984	4491	4221 (3.35%)	374281	60553	4300 (7.10%)
bluefish	and	33310	10124	4784	4753 (46.95%)	62676	5333	3729 (69.92%)
bluefish	das	25831	5627	4781	765 (13.60%)	51564	4243	2955 (69.64%)
bluefish	ecr	27634	31450	4796	8805 (28.00%)	90793	12053	5641 (46.80%)
codebreaker	and	255	174	166	11 (6.32%)	720	75	38 (50.67%)
codebreaker	das	246	174	166	13 (7.47%)	699	75	38 (50.67%)
codebreaker	ecr	274	174	166	11 (6.32%)	793	75	38 (50.67%)
concepts	and	3464	6179	712	681 (11.02%)	18028	3026	808 (26.70%)
concepts	das	3311	6948	708	368 (5.30%)	20237	3179	576 (18.12%)
concepts	ecr	5876	11212	708	667 (5.95%)	30491	4883	896 (18.35%)
cook	and	10945	11456	3645	1921 (16.77%)	22667	14712	4359 (29.63%)
cook	das	107245	312498	3616	42652 (13.65%)	727283	121782	21349 (17.53%)
cook	ecr	102249	256046	3687	21329 (8.33%)	552770	104884	17951 (17.12%)
dia	and	32107	18426	7261	7977 (43.29%)	48937	9972	5929 (59.46%)
dia	das	279671	655413	7241	120740 (18.42%)	1730164	192228	53466 (27.81%)
dia	ecr	367744	662485	7314	148139 (22.36%)	1713405	260564	99513 (38.19%)
euler	and	185994	392318	9528	166333 (42.40%)	525786	196513	95544 (48.62%)
euler	das	55714	123047	8887	43932 (35.70%)	232300	57758	24804 (42.94%)
euler	ecr	273469	713147	9845	183457 (25.72%)	1292219	379223	128111 (33.78%)
gnuplot	and	447451	817379	16330	360602 (44.12%)	1451834	772299	376791 (48.79%)
gnuplot	das	293075	663190	15986	287109 (43.29%)	1431677	529083	291206 (55.04%)
gnuplot	ecr	778697	2804689	16229	759962 (27.10%)	8463732	1945700	624687 (32.11%)
gqview	and	72562	73712	7894	32074 (43.51%)	125180	50453	25503 (50.55%)
gqview	das	91364	54889	7815	29996 (54.65%)	151379	38025	22131 (58.20%)
gqview	ecr	214935	433843	7888	92314 (21.28%)	1005400	199437	60313 (30.24%)
grep	and	3760	8891	1597	659 (7.41%)	23647	7475	1372 (18.35%)
grep	das	3911	11440	1590	676 (5.91%)	29486	9223	1324 (14.36%)
grep	ecr	5618	17449	1597	991 (5.68%)	48005	13897	1564 (11.25%)
gzip	and	3109	6272	1789	801 (12.77%)	11112	6196	1127 (18.19%)
gzip	das	2912	6695	1784	704 (10.52%)	12121	6352	996 (15.68%)
gzip	ecr	4089	9045	1790	1148 (12.69%)	18154	8368	1294 (15.46%)

continued on next page

name	*pta*	*locators*	*defs*	*must defs*	*pruned defs*	*uses*	*ϕ-nodes*	*pruned ϕ-nodes*
make	and	13993	62415	2437	1518 (2.43%)	135694	46390	2491 (5.37%)
make	das	11334	59296	2405	1073 (1.81%)	144097	40049	2283 (5.70%)
make	ecr	14114	72196	2434	1775 (2.46%)	165124	49518	2618 (5.29%)
nano	and	8063	26198	2057	2518 (9.61%)	45160	14298	2058 (14.39%)
nano	das	7840	24380	2031	2475 (10.15%)	44654	13704	2046 (14.93%)
nano	ecr	9037	31833	2085	3475 (10.92%)	55392	17021	2625 (15.42%)
screen	and	163888	971600	6219	89535 (9.22%)	1778390	585444	63925 (10.92%)
screen	das	123867	796814	5416	42239 (5.30%)	1662773	447096	30738 (6.88%)
screen	ecr	203329	1449372	6349	64264 (4.43%)	3190516	803970	47004 (5.85%)
sed	and	13652	52312	1853	357 (0.68%)	149950	37884	2044 (5.40%)
sed	das	13742	63596	1840	323 (0.51%)	189158	42407	2017 (4.76%)
sed	ecr	23367	80426	1856	402 (0.50%)	250361	58676	2000 (3.41%)
snns2c	and	748	1965	302	43 (2.19%)	7378	1381	206 (14.92%)
snns2c	das	902	2310	302	43 (1.86%)	11422	1426	206 (14.45%)
snns2c	ecr	1015	1711	302	60 (3.51%)	9310	1065	232 (21.78%)
soundtracker	and	14519	19256	3901	4858 (25.23%)	40382	10408	4301 (41.32%)
soundtracker	das	26800	58870	3892	9300 (15.80%)	120347	28194	7501 (26.60%)
soundtracker	ecr	40851	66071	3914	13148 (19.90%)	148042	31941	11150 (34.91%)
tar	and	113458	250313	4505	54848 (21.91%)	374475	181508	39043 (21.51%)
tar	das	77570	185185	4343	29239 (15.79%)	328482	125402	21716 (17.32%)
tar	ecr	200842	567705	4693	71912 (12.67%)	998019	359879	45883 (12.75%)
tcc	and	23278	116522	4696	17973 (15.42%)	174005	65184	11745 (18.02%)
tcc	das	11012	56486	3205	593 (1.05%)	96507	30766	2344 (7.62%)
tcc	ecr	45861	313876	5014	10108 (3.22%)	502298	149952	7098 (4.73%)
tcsh	and	426974	1554096	10348	239162 (15.39%)	2556297	986451	154584 (15.67%)
tcsh	das	299593	1653364	9867	293577 (17.76%)	2426474	908746	118260 (13.01%)
tcsh	ecr	555754	3209897	10453	482712 (15.04%)	4859916	1694488	182247 (10.76%)
time	and	179	124	90	44 (35.48%)	393	84	42 (50.00%)
time	das	170	124	90	46 (37.10%)	325	84	42 (50.00%)
time	ecr	181	124	90	44 (35.48%)	399	84	42 (50.00%)
trueprint	and	6961	23801	1585	1684 (7.08%)	31356	13296	1894 (14.24%)
trueprint	das	6828	25261	1612	1650 (6.53%)	48354	13395	1686 (12.59%)
trueprint	ecr	7489	10084	1582	1522 (15.09%)	30107	9268	1925 (20.77%)
units	and	1468	3281	641	185 (5.64%)	6302	2548	435 (17.07%)
units	das	916	1527	602	83 (5.44%)	4091	1264	340 (26.90%)
units	ecr	2409	6880	650	155 (2.25%)	16303	4347	420 (9.66%)
unzip	and	5452	12621	2410	1011 (8.01%)	37462	14737	1400 (9.50%)
unzip	das	4651	19308	2389	868 (4.50%)	38837	18266	1101 (6.03%)
unzip	ecr	8303	38940	2419	1227 (3.15%)	69674	37978	1558 (4.10%)
uucico	and	168488	901366	4656	121307 (13.46%)	1296770	552131	76402 (13.84%)
uucico	das	133611	834915	4376	72314 (8.66%)	1252870	458668	43442 (9.47%)
uucico	ecr	194026	1110954	4745	134367 (12.09%)	1525546	650133	80729 (12.42%)
wget	and	17054	37641	3626	1422 (3.78%)	83985	32097	3246 (10.11%)
wget	das	15724	61854	3580	1900 (3.07%)	107014	41779	3148 (7.53%)
wget	ecr	35269	132533	3623	3935 (2.97%)	222954	92212	4772 (5.18%)

Table 3. Resource consumption for ISSA creation

		locator generation		ISSA construction		ISSA pruning	
name	*pta*	*mem (MB)*	*time*	*mem (MB)*	*time*	*mem (MB)*	*time*
Astro	and	30.10	130ms	31.20	1s	31.33	90ms
Astro	das	30.61	140ms	31.80	1s	31.80	50ms
Astro	ecr	30.71	450ms	31.52	1s	31.65	80ms
bash	and	77.54	3s	111.55	1m35s	136.93	12s
bash	das	98.67	11s	213.95	7m6s	313.79	56s
bash	ecr	87.70	20s	160.48	4m11s	226.82	33s
bc	and	31.27	300ms	34.35	5s	35.91	710ms
bc	das	31.41	220ms	34.32	4s	35.50	470ms
bc	ecr	31.42	1s	36.68	11s	40.82	1s
bison	and	117.49	13s	136.97	1m57s	143.70	11s
bison	das	69.79	8s	95.20	1m20s	104.62	6s
bison	ecr	60.59	10s	85.29	1m23s	96.84	7s
bluefish	and	90.57	1s	105.16	19s	111.37	1s
bluefish	das	94.02	1s	111.59	19s	116.50	1s
bluefish	ecr	93.41	4s	109.15	35s	114.63	2s
codebreaker	and	26.74	30ms	27.25	610ms	27.25	20ms
codebreaker	das	26.75	40ms	27.62	640ms	27.62	10ms
codebreaker	ecr	26.87	120ms	27.40	640ms	27.40	30ms
concepts	and	28.94	210ms	31.11	2s	32.01	390ms
concepts	das	29.32	210ms	31.57	2s	32.49	370ms
concepts	ecr	28.97	580ms	31.61	4s	33.43	680ms
cook	and	55.32	720ms	62.88	11s	65.46	970ms
cook	das	73.62	12s	118.22	1m42s	155.45	19s
cook	ecr	64.48	10s	102.52	1m55s	139.32	16s
dia	and	162.88	3s	182.25	26s	186.43	1s
dia	das	209.80	1m15s	310.30	3m25s	381.32	35s
dia	ecr	198.29	1m19s	303.03	5m39s	382.25	40s
euler	and	91.41	4s	161.64	4m52s	208.50	25s
euler	das	82.39	2s	111.95	50s	127.00	6s
euler	ecr	93.88	21s	188.20	4m22s	285.12	59s
gnuplot	and	668.23	2m12s	815.27	25m13s	903.69	3m35s
gnuplot	das	194.64	54s	332.46	11m12s	392.75	1m5s
gnuplot	ecr	166.57	1m51s	619.11	37m15s	963.07	6m9s
gqview	and	126.79	3s	160.86	1m10s	175.46	5s
gqview	das	139.13	5s	169.67	53s	178.56	3s
gqview	ecr	144.28	24s	221.08	3m56s	281.16	27s
grep	and	33.51	300ms	37.25	6s	38.29	620ms
grep	das	34.16	340ms	38.64	6s	39.93	560ms
grep	ecr	33.76	1s	38.31	7s	40.11	830ms
gzip	and	29.99	170ms	32.32	3s	33.22	380ms
gzip	das	31.02	230ms	34.90	3s	35.83	340ms
gzip	ecr	30.50	550ms	33.13	4s	34.42	470ms

continued on next page

		locator generation		ISSA construction		ISSA pruning	
name	*pta*	*mem (MB)*	*time*	*mem (MB)*	*time*	*mem (MB)*	*time*
make	and	53.05	2s	65.66	47s	74.64	5s
make	das	40.81	960ms	53.94	32s	61.20	3s
make	ecr	40.32	1s	53.60	52s	62.64	4s
nano	and	36.06	540ms	42.36	16s	46.96	1s
nano	das	35.16	410ms	41.61	14s	45.98	1s
nano	ecr	35.06	1s	41.15	17s	46.29	2s
screen	and	174.32	28s	324.25	28m17s	424.14	1m32s
screen	das	86.28	10s	198.74	7m43s	277.45	50s
screen	ecr	80.58	26s	270.08	17m39s	407.40	1m47s
sed	and	43.03	1s	55.42	30s	61.07	2s
sed	das	41.48	1s	55.17	21s	61.75	3s
sed	ecr	40.44	3s	55.31	31s	65.58	5s
snns2c	and	26.10	90ms	27.39	1s	27.39	160ms
snns2c	das	26.23	100ms	27.63	1s	27.76	150ms
snns2c	ecr	26.13	190ms	27.21	1s	27.47	120ms
soundtracker	and	73.98	1s	85.67	16s	88.28	1s
soundtracker	das	78.50	1s	96.91	27s	103.64	2s
soundtracker	ecr	78.55	3s	93.38	28s	102.69	3s
tar	and	119.41	10s	164.05	3m53s	195.77	20s
tar	das	74.03	4s	108.20	1m38s	130.93	12s
tar	ecr	78.09	16s	153.54	4m44s	222.19	39s
tcc	and	42.00	780ms	61.33	1m14s	77.07	6s
tcc	das	42.52	650ms	54.13	19s	62.38	3s
tcc	ecr	43.80	5s	79.31	2m15s	117.11	20s
tcsh	and	140.49	14s	352.68	20m19s	577.54	2m40s
tcsh	das	100.39	10s	303.48	14m51s	488.27	1m20s
tcsh	ecr	103.34	35s	457.26	36m57s	823.18	5m46s
time	and	24.19	20ms	24.53	260ms	24.53	20ms
time	das	24.19	20ms	24.53	220ms	24.53	10ms
time	ecr	24.29	50ms	24.50	210ms	24.50	10ms
trueprint	and	37.19	830ms	41.77	11s	44.34	1s
trueprint	das	33.84	520ms	40.28	9s	42.79	1s
trueprint	ecr	33.32	670ms	37.34	5s	39.45	790ms
units	and	27.54	130ms	29.68	4s	30.20	260ms
units	das	27.66	100ms	29.67	2s	29.93	140ms
units	ecr	27.55	340ms	29.65	4s	30.55	420ms
unzip	and	39.41	490ms	44.86	11s	46.72	860ms
unzip	das	39.41	500ms	46.11	8s	48.12	850ms
unzip	ecr	38.75	1s	45.39	15s	49.72	1s
uucico	and	177.21	20s	292.31	15m0s	387.68	1m9s
uucico	das	86.60	11s	184.59	3m57s	267.86	34s
uucico	ecr	75.84	14s	201.43	12m46s	316.36	1m18s
wget	and	63.70	1s	76.83	36s	83.90	3s
wget	das	57.77	1s	71.47	28s	79.45	3s
wget	ecr	56.88	6s	77.20	1m13s	93.71	7s