Fast and Precise Slicing of Low-Level Code

Björn Lisper and Husni Khanfar
School of Innovation, Design, and Engineering
Mälardalen University
Västerås, Sweden
bjorn.lisper@mdh.se, husni.khanfar@mdh.se

December 11, 2013

Abstract

Program slicing is a technique to identify the parts of a program that may affect certain properties of the program, such as the outcomes of conditions that decide the program flow, or certain outputs. It is useful for many purposes: in particular, it has uses in timing analysis of code for real-time systems where it can aid the identification of program flow constraints that are subsequently used when bounding the execution time of the program. As code-level timing analysis often is performed on low-level code, slicing of low-level code becomes interesting. The classical methods for slicing concern high-level languages. In this report we discuss two ways to improve these methods. First, we show how to avoid the construction of some intermediate data structures by combining data-flow analysis with slicing on the fly: the purpose of this is to speed up the analysis. Second, we show how to perform the data dependence analysis for a programming model with a low-level view of memory, where memory operations target numerical, possibly dynamically computed addresses rather than distinct program variables.

1 Introduction

Program slicing refers to a collection of techniques to identify which parts in a program may affect a so-called "slicing criterion" that expresses certain properties of a program. The slicing criterion may be, for instance, the possible values of some program variables in some program points. The result of the slicing is usually a new program, formed by extracting certain statements from the original code. Slicing comes in many forms: *static* slicing computes a safe overapproximation of the code that might affect, or be affected by the slicing criterion, whereas *dynamic* slicing considers the statements that will affect (or be affected by) the criterion in different runs.

In this paper we consider *static backwards slicing*, which was first considered by Weiser [11]. This slicing extracts a safe subset of the statements in the program that can possibly affect the slicing criterion. It is best explained by example. Consider the following piece of code:

```
i = 1;
s = 0;
while (i < 100) {
  s = s + a[i];
  i = i + 1;}
```

Now assume that we want to extract the part of the program that can affect the outcome of the while-loop condition. A safe approximation of this part can be computed by static backwards slicing using the variable occurrences in the condition as slicing criterion. The resulting slice is shown below in red boldface:

```
i = 1;
s = 0;
```

```
while (i < 100) {
  s = s + a[i];
  i = i + 1;}
```

Here, it is easy to see that the assignments to `s` never will be able to affect the value of `i` in the loop condition. On the other hand, both assignments to `i` might do so: thus, they are included in the slice.

The example also demonstrates an application of slicing that motivates our interest. We are experts on timing analysis on code level, in particular so-called *Worst-Case Execution Time (WCET) analysis* [12]. An important part of WCET analysis is to constrain the possible program flows, like the possible number of iterations of a loop. In order to detect such constraints, it is only necessary to analyze the part of he program that can possibly affect the outcomes of the conditions in the program. Thus, backwards slicing with respect to some or all of the conditions can be done in order to reduce the program that has to be analyzed. This can be used to reduce the analysis time [9], or to find better upper limits on the number of program states which in turn can be used to bound program flows [2, 7].

WCET analysis is often performed in the final verification phase of safety-critical, hard real-time systems. It is then important that the analysis is *safe*, i.e., that is does not underestimate the longest possible execution time. Therefore it is often the final, linked binary that is analyzed since this is the code that is actually run in the final system. This implies that also the program flow constraints have to be computed on this level. Thus, there is also a need to perform slicing on machine code level. The standard methods for slicing (see Section 3) are formulated on source level. A major difference between source- and binary level is the memory model: on source level there are distinct program variables, and aliasing is typically limited. On the binary level, on the other hand, memory is accessed through numerical addresses that may be calculated dynamically. This affects slicing, and in particular the detection of data dependences becomes less straigthforward.

Our contributions are twofold: first, we demonstrate a way to speed up the standard method for static backwards slicing by eliminating the need to build the data dependence graph. Second, we propose a way to deal with the numerical addressing in low-level code by performing a value analysis prior to the slicing. We plan to implement these improvements in our WCET analysis tool SWEET [10].

The rest of this paper is organized as follows. In Section 3 we describe the standard method for static backwards slicing. Section 4.1 explains how the speed and memory consumption of this method can be improved, by using an alternative data flow analysis for the part of the analysis that detects data dependences. In Section 5 we show how to constrain addresses by a value analysis prior to the data dependence analysis. Section 6 describes a forthcoming implementation. In Section 7, finally, we wrap up and give directions for future research.

2 Preliminaries

We now introduce some concepts and notation. Most is standard, but nevertheless we introduce it for completeness and clarity.

We define a *control-flow graph* (CFG) as a directed graph (N, Flow) where the nodes in N are labeled either with conditions, assignments, a special label *start*, or ditto *stop*, and $\mathit{Flow} \subseteq N \times N$ is a relation describing the possible flows of execution in the graph. Each CFG contains a unique start node and a unique stop node. The start node has no predecessors, and the stop node no successors, An assignment node has exactly one successor, and a condition node has exactly two successors (labeled *true* and *false*, respectively).

We will sometimes write $[c]^n$ for a node n labelled with the condition c, and $[x := a]^n$ for a node n labelled with the assignment $x := a$. This notation makes it easier to define the data flow equations in Sections 2.1 and 4.1.

Conditions, and right-hand sides in assignments, are expressions. We assume that expressions have no side-effects, and that they are simple expressions built from program variables, constants, operators and primitive (not user-defined) functions. For simplicity we do not allow pointers and operations on such: all the analyses presented here can however be extended to deal with them. We assume that program variables are unaliased, i.e., an assignment to the program variable x can not affect the value of another program variable $y \neq x$. For an expression e, $FV(e)$ denotes the set of program variables that appear in e.

Note that we label the nodes by single statements, whereas in compiler literature the nodes often are considered to represent basic blocks. This is for two reasons: first, we want to perform the slicing on statements rather than basic blocks, and similarly it is easier to define data flow analyses and other static program analyses by equations over statements.

Postdominators play an important role in slicing. A node n in a CFG is said to postdominate a node n' if and only if every path from n' to the stop node goes through n. There are many algorithms to compute postdominators, as well as *postdominator trees* which can be used to efficiently represent sets of postdominators for different nodes.

2.1 Data Flow Analysis

An important part of slicing is to compute data dependencies. A *data flow analysis* [8] is often used for this purpose. Data flow analyses are a class of static program analyses that traditionally are used in optimizing compilers. They are usually defined over CFG's in the following way. For each node n in the CFG, the analyses compute sets $S_{entry}(n)$ and $S_{exit}(n)$ which are present before and after the node, respectively. The sets represent some kind of data flow information. Depending on the nature of the data flow computed, an analysis is a *forward* or *backwards* analysis, respectively. The sets are related through equations. For a forward analysis, they have the following form:

$$\begin{array}{rcl} S_{entry}(start) & = & S_{init} \\ S_{exit}(start) & = & S_{entry}(start) \\ S_{exit}(stop) & = & S_{entry}(stop) \\ S_{exit}(n) & = & (S_{entry}(n) \setminus kill(n)) \cup gen(n), \quad n \neq start, n \neq stop \\ S_{entry}(n) & = & \bigcup_{n' \in pred(n)} S_{exit}(n'), \quad n \neq start, n \neq stop \end{array} \tag{1}$$

Here, $pred(n)$ is the set of immediate predecessors to n in the CFG, and S_{init} is some set describing the data flow information that is present at the entry of the program. For a backwards analysis, the directions of the equations are reversed:

$$\begin{array}{rcl} S_{exit}(stop) & = & S_{init} \\ S_{entry}(stop) & = & S_{exit}(stop) \\ S_{entry}(start) & = & S_{exit}(start) \\ S_{entry}(n) & = & (S_{exit}(n) \setminus kill(n)) \cup gen(n), \quad n \neq start, n \neq stop \\ S_{exit}(n) & = & \bigcup_{n' \in succ(n)} S_{entry}(n'), \quad n \neq start, n \neq stop \end{array} \tag{2}$$

Here, $succ(n)$ is the set of immediate sucessors to n in the CFG, and S_{init} is some set describing the data flow information that is present at the exit of the program. Data flow analyses are further divided into *may* and *must* analyses. In this paper we will only deal with may analyses: (1) and (2) are valid for these.

A classical data flow analysis is *Reaching Definitions* (RD). It is a forward may analysis: thus, its equations are formed according to (1). RD computes sets of pairs (x, n), where x is a program variable and n is a node in the CFG. If (x, n) belongs to the set associated with program point p, then the value of x that was assigned at n may still be stored in x when at p: thus, there is then a possible data flow from n to p. For assignments $x := a$, and conditions c, the *kill* and *gen* sets are defined as follows (as well as the set S_{init}):

$$\begin{array}{rcl} S_{init} & = & \{ (x, ?) \mid x \text{ is a variable in the program} \} \\ kill([x := a]^n) & = & \{ (x, n') \mid n' \in N \} \cup \{(x, ?)\} \\ gen([x := a]^n) & = & \{(x, n)\} \\ kill([c]^n) & = & \emptyset \\ gen([c]^n) & = & \emptyset \end{array}$$

where N is the set of nodes in the CFG. Here, $(x, ?)$ represents an initial value that x has at the beginning of the program execution. These definitions assume that conditions have no side-effects, but can easily be modified to take such into account. The equations can also be extended to handle features like pointers, arrays, and function calls.

```
$Analyze(N, F, e, S_{init}, \langle f_n \mid n \in N \rangle) =$
  /* Initialization */
  $W := \emptyset;$
  $for\ all\ n\ where\ (e,n) \in F\ do\ W := W \cup \{(e,n)\};$
  $S[e] = S_{init};$
  $for\ all\ n \in N \setminus \{e\}\ do\ S[e] = \emptyset;$
  /* Iteration */
  $while\ W \neq \emptyset\ do$
    $(n,n') := select(W);$ /* Pick new element from $W$ */
    $W := W \setminus \{(n,n')\};$
    $if\ f_n(S[n]) \not\subseteq S[n']\ then$ /* $S[n']$ has changed: add new work item to $W$ */
      $S[n'] := S[n'] \cup f_n(S[n]);$
      $for\ all\ n''\ where\ (n',n'') \in F\ do\ W := W \cup (n',n'');$
  /* Finalization */
  $return\ S;$
```

Figure 1: Generic worklist algorithm.

(1) (or (2)) yields a system of $2 \cdot |N|$ equations, where the unknowns are the sets $S_{entry}(n)$ and $S_{exit}(n)$ for the different nodes n in the CFG. The classical way to solve the system is by *fixed-point iteration*: all unknown sets are initialized to $\emptyset$, and then the system of equations is iterated until no more sets change. The underlying theory of complete lattices [8] ensures that this procedure converges, and always yields the *least* (most precise) solution.

There is a generic worklist (or work set) algorithm to perform fixed-point iteration [8]: it handles both forward and backwards analyses. It computes an array S of sets, indexed by the nodes in the CFG (N, Flow), where each element $S[n]$ equals $S_{entry}(n)$ for forward analyses, and $S_{exit}(n)$ for backwards analyses. For each node n, the *transfer function* f_n defines how to compute $S_{exit}(n)$ from $S_{entry}(n)$ according to (1), for forward analyses, or $S_{entry}(n)$ from $S_{exit}(n)$ according to (2), for backwards analyses. F is either the set of edges Flow in the CFG (forward analyses), or the set of reversed edges Flow^{-1} (backwards analyses). e is either the *start* node (forward analyses), or the *stop* node (backwards analyses). W, the *work set*, holds a subset of edges from F: if $(n, n') \in W$ then this indicates that $S_{entry}(n')$ ($S_{exit}(n')$) has changed, and thus $S_{exit}(n')$ ($S_{entry}(n')$) may have to be recomputed. When W is empty the algorithm terminates, and the analysis result is present in the array S. In addition to the usual set operators, $select(W)$ picks non-deterministically some element from W. Fig. 1 gives pseudocode for the worklist algorithm.

The number of fixed-point iterations can be at most $h \cdot |N|$, where h is the height of the lattice iterated over and $|N|$ is the number of nodes in the CFG. For dataflow analysis, h equals the size of the largest possible set of data flow information. Thus the execution time is $O(t \cdot h \cdot |N|)$, where t is an upper bound for the time needed to perform one fixed-point iteration.

3 Program Slicing using the Program Dependence Graph

The standard method for static backwards program slicing is based on the so-called *Program Dependence Graph* (PDG) [3]. This graph is derived from the control flow graph (CFG) of the program by extracting data dependencies into a *Data Dependence Graph* DDG, and so-called control dependencies into a *Control Dependence Graph*. All these graphs have the same set of nodes as the CFG. We here assume that the slicing criterion is a set of nodes C in the CFG, or, to be more exact: for each node $n \in C$ the possible values at n of the program variables that can affect the statement labelling n. Somewhat informally, the method then works as follows:

1. Perform RD, computing a set of reaching definitions $S_{entry}(n)$ for each node n in the CFG.

```
void main() {
0.  int a[9],x,y,i;
1.  x=42;
2.  y=9;
3.  if (x==42) {
4.    y-;
    }
5.  i=0;
    do {
6.    a[i] = i * 4;
7.    i++;
8.  } while (i<y);
9.  return; }
```

(a) Example code

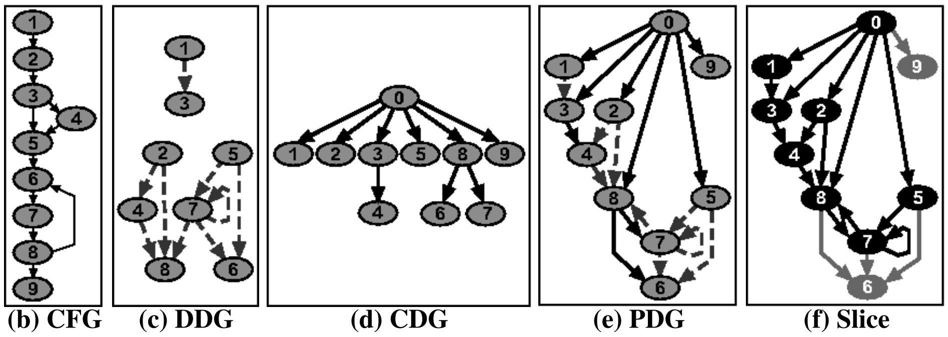

Figure 2: PDG slicing illustration

2. Compute the DDG as follows: for each node n in the CFG, if x is used by n and if $(x, n') \in S_{entry}(n)$ then (n', n) is an edge in the DDG.

3. The CDG is defined as follows: (n, n') is an edge in the CDG iff n is a condition, there is a path from n to n' in the CFG, and n' is not a postdominator to n. It is computed using an algorithm for computing postdominator trees.

4. Compute the PDG as the union of the DDG and the CDG.

5. The static backwards slice with respect to C is now computed as all nodes in the PDG that are reachable by a backwards search from any node in C.

An example is shown in Fig. 2. As can be seen, the computation of the backwards slice involves the formation of some graphs. As shown in Section 4, this formation may in the worst case take time quadratic in $|N|$. The final backwards search can be done in time $O(|N| + |Flow|)$, which also is quadratic in $|N|$ in the worst case but often tends to be smaller for CFG's, where the typical restrictions on the number of successors to nodes will curb $|Flow|$. Thus, if the graph formation can be eliminated or reduced, chances are that the total execution time for the slicing can be significantly reduced.

4 Slicing-on-the-Fly using Strongly Live Variables

As mentioned in Section 3, the RD dataflow analysis is used as the first step when building the DDG. It computes one RD set for each node in the CFG, and it is typically implemented by a worklist algorithm for fixed-point iteration according to Fig. 1. Each RD set must contain at least v reaching definitions, where v is the number of variables in the program, as each variable value must have at least one origin (possibly the start of the program, if the variable is not initialized). It can contain at most $|N| \cdot v$ reaching definitions, where $|N|$ is the number of nodes in the CFG, as any variable value can have at most $|N|$ origins. Thus, the total size of all the RD sets is between $|N| \cdot v$ and $|N|^2 \cdot v$.

The time to do fixed-point iteration for RD is $O(t \cdot |N| \cdot v \cdot |N|) = O(t \cdot |N|^2 \cdot v)$, where t is an upper bound to the time required to perform one fixed-point iteration. t includes the time to compute a transfer function in the analysis, to take the union of two sets of RD's, and to check whether such a set is included in another one or not. These times depend on the chosen set representation, but can be expected to be at least logarithmic in the set sizes.

The size of the DDG is $O(|N|^2)$ in the worst case. Again the time to compute and store each edge depends on the chosen data representations, but should for each node in N not need to be more than proportional to the size of its RD set. This yields a worst-case time complexity for building the DDG of $O(|N|^2)$ as well.

In total we obtain the space complexity $O(|N|^2 \cdot v)$, and time complexity $O(t \cdot |N|^2 \cdot v)$, for identifying the data dependencies and building the DDG. The potentially quadratic costs in the number of nodes in the CFG indicates that this phase of the slicing can be costly. It is thus motivated to look for alternative analyses that can help detecting the data dependencies and slice according to them at a lower cost. We now describe such an approach. It has the following advantages:

- it eliminates the need to build the DDG, and
- the data flow analysis used, the *Strongly Live Variables* analysis, has a lower worst-case complexity than RD.

4.1 The Strongly Live Variables Analysis

The Strongly Live Variables (SLV) analysis (Excercise 2.4 in [8]) is an alternative data flow analysis for computing data dependencies. Given some sets of variables in some different program points, representing *uses* of these variables, it computes for each program point a set of "strongly live variables" whose values in that program point might affect some use of some variable. It is a backwards may analysis, with kill, gen and S_{init} sets defined as follows:

$$\begin{array}{rcl} S_{init} & = & \emptyset \\ \mathit{kill}([x := a]^n) & = & \{x\} \\ \mathit{gen}([x := a]^n) & = & FV(a) \\ \mathit{kill}([c]^n) & = & \emptyset \\ \mathit{gen}([c]^n) & = & FV(c) \end{array}$$

The transfer functions for SLV analysis are set up according to (2), however with the following modification (for n being an assignment or a condition):

$$\begin{array}{rcl} S_{entry}(n) & = & (S_{exit}(n) \setminus \mathit{kill}(n)) \cup \mathit{gen}(n), \quad \mathit{kill}(n) \subseteq S_{exit}(n), \\ S_{entry}(n) & = & S_{exit}(n), \quad \text{otherwise} \end{array}$$

Each SLV set can contain at most v elements, where v is the number of variables in the program. Thus, the total size of the SLV sets is $O(|N| \cdot v)$, where $|N|$ is the number of nodes in the CFG. The height of the lattice is also v and thus the time to do fixed-point iteration for SLV is $O(t \cdot |N| \cdot v)$ where t is an upper bound to the time required to perform one fixed-point iteration. As for RD, t depends on the time required to perform set operations.

The above defines the "standard" SLV analysis, as known from the literature. It is based on the following assumptions on variable uses:

- variables are not used at the end of a program,
- variables in conditions are always used, and
- variables are not used in any other program points.

For slicing, a slicing criterion corresponds to a set of variable uses. To allow for a more flexible specification of slicing criteria, not necessarily adhering to the assumptions on variable uses above, we modify the definition of SLV as follows:

$$\begin{array}{rcl} S_{init} & = & S' \\ \mathit{gen}(c) & = & \emptyset \end{array}$$

Here, S' is a set of variables providing a slicing criterion for the end of the program. Slicing criteria for other statements in the program are provided for in the worklist algorithm for fixed-point iteration, through initializing array elements $S[n]$ to sets of variables, constituting slicing criteria for certain nodes n in the CFG, rather than initializing $S[n] = \emptyset$. See further Section 4.2 below.

	dataflow set sizes	**time for fixed-point iteration**
PDG	$O(\|N\|^2 \cdot v)$	$O(t \cdot \|N\|^2 \cdot v)$
SLV	$O(\|N\| \cdot v)$	$O(t \cdot \|N\| \cdot v)$

Table 1: Table with worst-case complexities for standard PDG-based slicing, and SLV slicing-on-the-fly.

4.2 A Worklist Algorithm for Slicing on the Fly

We now describe an algorithm to perform static backwards slicing, where the slicing is done concurrently with the SLV analysis rather than using a data flow analysis to build the DDG prior to the actual slicing. It is an adaptation of the generic worklist algorithm in Fig. 1, modified in the following way:

- A set N_{slice} of sliced CFG nodes is maintained by the algorithm.
- A node $[x := a]^n$ is included in the slice whenever $x \in S[n]$. This takes care of slicing due to data dependencies.
- The algorithm uses the explicit CDG C to perform slicing due to control dependencies. If n is included in the slice, and if $(n', n) \in C$ (where n' is a conditional node with condition c), then n' is included in the slice as well, $FV(c)$ is added to $S[n']$, for each edge $(n'', n') \in C$, (n', n'') is added to the worklist W, and if $S[n']$ changes then all edges $(n', n'') \in F$ are added to the worklist as well.
- To support the above the worklist W is modified to hold two types of edges, which are tagged to keep them apart: (n, n', CFG) for edges in the CFG, and (n, n', CDG) for edges in the CDG.
- Rather than initializing $S[e] = S_{init}$ and $S[n] = \emptyset$ for all other CFG nodes n, the array S is initialized to an array S_{crit} containing the slicing criteria (sets of variables in certain program points). Furthermore, N_{slice} is initialized to the set of nodes n where $S_{crit}[n] \neq \emptyset$.

We assume that if a slicing criterion $S_{crit}[n]$ is nonempty, then n is always to be included in the slice. This is true, e.g., when slicing with respect to the conditions in a program in order to perform control flow analysis.

Pseudocode for the resulting worklist algorithm is given in Fig. 3. The transfer functions f_n are the ones for our modified SLV analysis, as described in Section 4.1. For conditional nodes n in the CFG, $c(n)$ denotes the condition of n. The complexity is essentially the same as for running the generic worklist algorithm for SLV, $O(t \cdot |N| \cdot v)$, since the termination criterion for the worklist iteration is the same as well as the lattice of SLV sets. The factor t will of course change, but its order should be the same as for the original SLV analysis since the same set operations are used. The total size of the SLV sets is still $O(|N| \cdot v)$, as these sets are the same. Since the generation of the DDG is eliminated, this is the total complexity for the slicing algorithm. Overall, the worst-case complexities are reduced by a factor $|N|$. See Table 1.

4.3 Slicing on the Fly With Respect to all Conditions

As mentioned in Section 1, in the flow analysis phase of WCET analysis it is often interesting to slice with respect to all the conditions in the program. In the SLV slicing algorithm, the slicing criterion will thus consist of all conditions in the program plus their sets of variables. *As the conditions thus are contained in the slice from the beginning, the control dependencies can be disregarded.* This is since the sole function of the control dependencies is to determine which conditions to include in the slice: thus, if all conditions are already included the control dependencies will have no use.

The SLV slicing worklist algorithm in Fig. 3 can be simplified accordingly, see Fig. 4. This is close to the original SLV algorithm, with the modifications mentioned in Section 4.1. The basic complexities will be the same as for the general SLV slicing algorithm, but the real execution time can be expected to be lower since the handling of the CDG is eliminated. Furthermore the CDG does not even have to be generated, which eliminates another phase in the slicing. This results in an algorithm that works directly on the CFG and does not require any other additional data structures than the worklist and the SLV sets.

```
$SLV\_slice(N, F, C, S_{crit}, \langle f_n \mid n \in N \rangle) =$
  /* Initialization */
  $W := \emptyset;$
  $N_{slice} := \emptyset;$
  $for\ all\ n \in N\ do$
    $S[n] := S_{crit}[n];$
    $if\ \ S_{crit}[n] \neq \emptyset\ then$
      $N_{slice} := N_{slice} \cup \{n\};$
      $for\ all\ n'\ where\ (n, n') \in F\ do\ W := W \cup \{(n, n', CFG)\};$
      $for\ all\ n'\ where\ (n', n) \in C\ do\ W := W \cup \{(n, n', CDG)\};$
  /* Iteration */
  $while\ W \neq \emptyset\ do$
    $(n, n', type) := select(W);$ /* Pick new element from W */
    $W := W \setminus \{(n, n', type)\};$
    $if\ n'$ is an assignment $x := a \wedge x \in S[n'] \cup f_n(S[n]) \wedge n' \notin N_{slice}\ then$
      $N_{slice} := N_{slice} \cup \{n'\};$
      $for\ all\ n''\ where\ (n'', n') \in C\ do\ W := W \cup (n', n'', CDG);$
    $if\ type = CFG \wedge f_n(S[n]) \not\subseteq S[n']\ then$ /* $S[n']$ has changed: add new work item to W */
      $S[n'] := S[n'] \cup f_n(S[n]);$
      $for\ all\ n''\ where\ (n', n'') \in F\ do\ W := W \cup (n', n'', CFG);$
    $if\ type = CDG\ then$ /* The edge is a control dependency */
      $S[n'] := S[n'] \cup FV(c(n'));$
      $for\ all\ n''\ where\ (n', n'') \in F\ do\ W := W \cup (n', n'', CFG);$
      $if\ n' \notin N_{slice}\ then$
        $N_{slice} := N_{slice} \cup \{n'\};$
        $for\ all\ n''\ where\ (n'', n') \in C\ do\ W := W \cup (n', n'', CDG);$
  /* Finalization */
  $return\ N_{slice};$
```

Figure 3: Worklist algorithm for slicing-on-the-fly with SLV analysis.

5 Data Dependence Analysis for Slicing of Low-Level Code

All the slicing algorithms presented so far rely on a high-level model of memory. The data dependence analyses, whether separate or integrated with the slicing, are based on program variables that are distinct and non-overlapping: an assignment to a variable x can thus not affect the value of any other variable $y \neq x$. Even if pointers to program variables are introduced, as long as this assumption holds it is well-known how to modify the data dependence analyses to cope with the possible aliasing that ensues.

However, for low-level code the memory model is different. Here a memory access is typically done to a numerical address, accessing a certain number of bits. Both the addresses and the sizes of accesses may vary dynamically (an example of the latter is block transfer instructions, where whole memory blocks are copied). Thus, it is not entirely straightforward to decide whether two memory accesses may overlap or not. Here we will sketch a way to analyze memory references to decide this. As the data flow analyses under consideration are may analyses, we are interested in approaches that can safely decide when accesses can *not* overlap: if they surely don't, then they cannot carry a data dependence. On the other hand, if there is a possible overlap then we will assume that there is a possible data dependence: this will yield a safe analysis where all data dependences surely are included in the result.

We will assume the memory model of the language ALF [5, 6], which is the language analyzed by the WCET analysis tool SWEET [10]. The idea is to translate different formats into ALF for subsequent analysis. ALF is thus designed to be able to faithfully represent both high- and low-level code, and its memory model is chosen according to this.

Memory in ALF is organized into *frames*. Each frame is a separate memory area. An ALF address consists of a so-called "frameref", which can be seen as a symbolic base pointer to the frame, and a

```
$SLV\_slice\_allconditions(N, F, \langle f_n \mid n \in N \rangle) =$
  /* Initialization */
  $W := \emptyset;$
  $N_{slice} := \emptyset;$
  for all $n \in N$ do
    if $n$ is a condition $c$ then
      $S[n] := FV(c);$
      $N_{slice} := N_{slice} \cup \{n\};$
      for all $n'$ where $(n, n') \in F$ do $W := W \cup \{(n, n')\};$
    else
      $S[n] := \emptyset;$
  /* Iteration */
  while $W \neq \emptyset$ do
    $(n, n') := select(W);$ /* Pick new element from $W$ */
    $W := W \setminus \{(n, n')\};$
    if $n'$ is an assignment $x := a \wedge x \in S[n'] \cup f_n(S[n])$ then $N_{slice} := N_{slice} \cup \{n'\};$
    if $f_n(S[n]) \not\subseteq S[n']$ then /* $S[n']$ has changed: add new work item to $W$ */
      $S[n'] := S[n'] \cup f_n(S[n]);$
      for all $n''$ where $(n', n'') \in F$ do $W := W \cup (n', n'');$
  /* Finalization */
  return $N_{slice};$
```

Figure 4: Worklist algorithm for slicing-on-the-fly with respect to all conditions.

numerical offset. This memory model is close to the one for unlinked code, where the final linking and memory allocation has not yet been made and base addresses are still not resolved. Frames are assumed to be non-overlapping, such that memory accesses to different frames can never overlap. Accesses to the same frame can, however, do so.

This memory model supports both the high- and low-level view. If a high-level language is translated to ALF, then distinct variables are preferrably mapped to single, distinct frames with the same number of bits as the variables they represent. For low-level code, on the other hand, larger memory areas might be mapped to frames: e.g., for executable binaries, the data memory may be modeled by a single frame.

Memory accesses in ALF uses an address as above, and a specified size. Thus an access can be represented by a triple (f, o, s), where f is a symbolic frameref, o is an offset (non-negative, in bits) and s is a size (ditto). The semantics is that the bits o to $o + s - 1$ are accessed in frame f. Two memory accesses (f, o, s) and (f', o', s') will thus overlap iff $f = f'$, and $[o, o + s - 1] \cap [o', o' + s' - 1] \neq \emptyset$.

It should now be clear how to analyze potential data dependencies. What is needed is a value analysis that yields safe overapproximations to the memory access triples (f, o, s). Such analyses are standard, and can be developed within the framework of abstract interpretation [1, 8]. The set of possible triples T for a memory access in the code will then be approximated by an "abstract triple" $T^{\#}$ in an abstract domain, where $T^{\#}$ represents a set of triples that surely contains T.

The scenario described above is somewhat simplified. Since not only addresses depend on values, but also values may depend on addresses, it is in general not possible to first perform a value analysis and then an address analysis. Thus, the address- and value analyses have to be combined into a joint analysis where approximated addresses and values are computed concurrently. What is needed is a combined abstract domain for these: then, the analysis can be carried out by a fixed-point iteration using standard methods.

If $\mathbf{F}$ is the set of framerefs in the program under analysis, and $\mathbf{Int}$ is the domain of integer intervals, then two possible abstract domains for memory access triples are $\mathcal{P}(\mathbf{F}) \times \mathbf{Int} \times \mathbf{Int}$ and $\mathcal{P}(\mathbf{F} \times \mathbf{Int} \times \mathbf{Int})$. The first domain represents sets of triples by "abstract triples" (F, I_o, I_s) where F is a set of framerefs, and I_o and I_s are intervals surely containing the possible offsets and sizes, respectively. The second domain represents sets of triples by finite sets of abstract triples $\{(f_1, I_{o1}, I_{s1}), \dots, (f_n, I_{on}, I_{sn})\}$ where $f_1, \dots, f_n$ are framerefs and I_{ok}, I_{sk} are intervals surely containing the possible offsets and sizes for the accesses to frame f_k. The second domain allows for more precise representations than the first, but is also potentially more costly. Both abstract domains are standard within abstract interpretation, and it is well-known how to implement value analyses that compute abstract values in these domains for different memory accesses in a program. Other abstract domains can also be used, but the two domains above seem natural to use and are obvious candidates for implementation.

If abstract values are computed for memory accesses by a value analysis, safe tests for overlaps will look as follows. With the domain $\mathcal{P}(\mathbf{F}) \times \mathbf{Int} \times \mathbf{Int}$, $(F, [l_o, u_o], [l_s, u_s])$ and $(F', [l'_o, u'_o], [l'_s, u'_s])$ represent possibly overlapping accesses if $F \cap F' \neq \emptyset$, and $[l_o, u_o + u_s - 1] \cap [l'_o, u'_o + u'_s - 1] \neq \emptyset$. For $\mathcal{P}(\mathbf{F} \times \mathbf{Int} \times \mathbf{Int})$, $\{(f_1, [l_{o1}, u_{o1}], [l_{s1}, u_{s1}]), \dots, (f_n, [l_{om}, u_{om}], [l_{sm}, u_{sm}])\}$ and $\{(f'_1, [l'_{o1}, u'_{o1}], [l'_{s1}, u'_{s1}]), \dots, (f'_n, [l'_{on}, u'_{on}], ['l_{sn}, u'_{sn}])\}$ represent possibly overlapping accesses if there exists i, j such that $f_i = f'_j$, and $[l_{oi}, u_{oi} + u_{si} - 1] \cap [l'_{oj}, u'_{oj} + u'_{sj} - 1] \neq \emptyset$. These overlap tests can be used when checking for possible data dependencies in the slicing.

6 Implementation

The new slicing algorithms described in this report have not been implemented yet, but implementations are underway. We plan to implement them in our tool SWEET, where the standard, PDG-based slicing algorithm already is implemented. First implementations are expected Q1 2014. Once the implementations are done, we will carry out experiments comparing the performance and precision of the different algorithms on a number of benchmark codes.

7 Conclusions and Further Research

Program slicing is a technique to extract the program parts that can possibly affect certain slicing criteria. One use of slicing is in WCET analysis, where methods for program flow analysis use slicing to speed up or improve the precision of the analysis. As WCET analysis is often performed on low-level code, slicing methods that can handle such code are needed. We describe two ways to improve existing slicing techniques: the first algorithm gives a more efficient data dependence analysis, and the second extends the data dependence analysis to deal with low-level memory models. Implementations are underway, and once the algorithms are implemented their performance and precision will be experimentally evaluated.

Future work includes additional improvements. For instance, the performance of the SLV-based slicing can most likely be improved by a judicious choice of iteration order [8]. It is also possible that more elaborate abstract domains are needed to obtain sufficient precision for the low-level data dependence analysis: the experiments will tell. An interesting extension is to use the product domain of intervals and congruences [4], rather than just intervals, for the offsets in memory accesses: this would allow for a more precise data dependence analysis of array accesses with different strides.

References

[1] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[2] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007)*, Pisa, Italy, July 2007.

[3] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[4] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, pages 165–199, 1989.

[5] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. ALF (ARTIST2 Language for Flow Analysis) specification. Technical report, Mälardalen University, Västerås, Sweden, January 2009.

[6] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF – a language for WCET flow analysis. In Niklas Holsti, editor, *Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009)*, pages 1–11, Dublin, Ireland, June 2009. OCG.

[7] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In Jan Gustafsson, editor, *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, pages 77–80, Porto, July 2003.

[8] Flemming Nielson, Hanne Ries Nielson, and Chris Hankin. *Principles of Program Analysis, 2nd edition*. Springer, 2005. ISBN 3-540-65410-0.

[9] Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Faster WCET flow analysis by program slicing. In Mary Jane Irwin and Koen De Bosschere, editors, *Proc. ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'06)*, pages 103–112, Ottawa, Ontario, Canada, June 2006.

[10] SWEET home page, 2011. `www.mrtc.mdh.se/projects/wcet/sweet/`.

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

[12] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.