Prototype implementation of methods for precise slicing

Abu Naser Masud

December 1, 2014

1 Introduction and Motivation

Program slicing is a technique that identifies the program parts affecting (or be affected by) a given slicing criterion. Slicing criterion represents the use of some program variables at any given program location. There are many applications of program slicing including debugging, program parallelization, software maintenance, integration, testing etc. In the APARTS project, we are interested in improving the performance of the Worst Case Execution Time (WCET) analysis of safety-critical real-time embedded software. Program slicing can benefit the WCET analysis greatly by reducing the program codes that do not affect the WCET program flow. If the given program is sliced with respect to all the conditions of the program, then the subsequent WCET analysis on the sliced code should usually be faster as the analysis is performed only on the codes that surely affect the WCET.

We have developed a light-weight, interprocedural and precise program slicing technique that performs faster than PDG-based slicing when few slices of the same code are computed. WCET analysis is often performed on the low-level code and thus slicing the low level code can reduce the code size and improve the subsequent WCET analysis performance. Hence, we have extended our interprocedural slicing to slice low-level code where the memory model is similar to the monolithic memory model. In this memory model, 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. Our slicing technique is based on an abstract interpretation-based value analysis that is used to determine addresses for memory accesses in the dataflow analysis.

We have implemented our slicing technique in the WCET prototype tool called SWEET [4, 1] and compared our slicing technique with the PDG-based slicing in order to compare the performances of the slicing techniques.

1.1 Implementation Features

In SWEET, there has already been an implementation of slicing based on building the program dependence graph (PDG). The PDG consists of the nodes of

the control flow graph, and an edge between two nodes in PDG represents some data or control dependency between the nodes. This PDG-based slicing did not handle the low-level view of memory and hence it is sometimes less precise. We have implemented two things: (1) our new slicing method that is based on the Strongly Live Variables (SLV) data-flow analysis, and (2) the previous PDG-based slicing is extended so that it handles low-level view of memory. The input to both the slicing programs are ALF programs [2]. ALF is designed to be able to faithfully represent both high- and low-level code, and its memory model. 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 numerical offset. It is assumed that memory accesses to different frames never overlap, whereas accesses to the same frame may do.

1.2 Implementation Details

In order to implement our SLV-based slicing method, we need to implement the followings:

1. the Extended Control Flow Graph (ECFG) and the Control Dependency Graph (CDG)
2. the *abstract domain* for memory accesses based on the *interval abstract domain*, and
3. the *gen* and *kill* functions for each statement type,
4. the slicing algorithm.

The ECFG and the CDG were already built into the SWEET tool. The ECFG is essentially a control flow graph (CFG) where some additional nodes are included in order to represent the control flow between different procedures more explicitly. The CDG contains the same nodes as ECFG but the edges in CDG represent some control dependency among ECFG nodes.

The interval abstract domain was already implemented in SWEET. This domain works on the control flow graph (CFG) of the input ALF program. On the other hand, our slicing method works on the extended CFG of the input ALF program. ECFG is an instantiation of different data types than CFG. We have lifted the interval abstract domain for the CFG to the ECFG by creating mapping between the CFG nodes and the corresponding ECFG nodes. The mapping considers a node in the ECFG and find a corresponding node in the CFG that represent the same ALF statement and the abstract values (i.e. interval values) for the CFG node are lifted into the corresponding ECFG node. However, there exists some ECFG nodes for which no CFG node exists. The abstract values for those nodes are computed from the other adjacent ECFG nodes.

Memory in ALF is organized into frames. Each frame is a separate memory area. We have created a datatype called *FrameAccess* to store abstract values

representing memory segments that are being read or write. Abstract values of type *FrameAccess* stores three kinds of information: (1) the frame identifier, (2) the abstract value represented as an interval of the base pointer of the specified frame, and (3) the abstract value represented as an interval of the offsets into the specified frame, that is being read or write. For each ECFG node, we have created two sets of abstract values of type *FrameAccess*. These sets of abstract values represent which memory segments are *defined* (i.e. *kill* function) or *used* (i.e. *gen* function) by the ECFG node.

We have implemented our slicing algorithm according to Algorithm 1 in [3]. The slicing program traverses the ECFG graph in a backward direction and generates the strongly live variables (SLV) for the ECFG nodes during its traversal. The slicing program also traverses the CDG in order to include some nodes due to control dependency. Initially, the user-specified slicing criteria are converted to build the initial set of SLVs for some ECFG nodes. The SLV set for some other ECFG nodes are empty. Each element in the SLV set is of type *FrameAccess*. Once the initial SLV sets are built, a worklist is generated that contains a list of ECFG edges $A \rightarrow B$ such that the SLV set in B is not empty. The slicing program then iteratively selects an element from the worklist, apply the *transfer function* implemented according to equations (4) and (5) in [3] on an ECFG node corresponding to the selected element, and adds new ECFG edges $C \rightarrow A$ into the worklist if the SLV set of ECFG node A is changed. If ECFG node A defines a memory segment that is present into the SLV set of B, then node A is inserted into the list of sliced ECFG nodes. If node A is sliced, then any control dependency edge $D \rightarrow A$ from the CDG is put into the worklist and node D is sliced later on. The program is terminated when the worklist is empty. The *transfer functions*, and other set operations onto the SLV sets are implemented so that it operates into the abstract data set. That means abstract operations are performed into the data of type *FrameAccess*.

We also have extended the implementation of PDG-based slicing so that the extended slicing handles the ALF memory model. Extending the PDG-based slicing requires building the PDG precisely so that an edge in the PDG precisely represents the data or control dependency. We have extended the construction of PDG into the following ways:

1. First, we consider a vertex of the PDG, and generate all the memory segments that are being *used* by the ALF statement represented by that vertex. We have used our implementation of the *gen* function that generates all such memory segments. It operates into the abstract data of type *FrameAccess* and generates memory segments which are also of type *FrameAccess*.

2. Then, for each memory segment *used* in the previous step, we generate all the vertices of the PDG that may define the previously used memory segments completely or partially into the same frame. We have obtained this information from the *reaching definition* analysis. Since this operation is an overapproximation, in this stage, we may obtain some memory segments that are being defined but not used in the previous step.

3. Finally, we consider the two memory segments corresponding to some uses and definitions of memory segments obtained in step 2 and check weather they overlap or not. We add an edge, from the vertex that defines the memory segment to the vertex that uses some memory segment and overlaps, into the PDG.

The PDG constructed by the above approaches includes all the data and control dependency which handles the low-level memory model precisely. Now the PDG-based slicing program applied to this newly constructed PDG provides a precise slicing.

1.3 Experimental Evaluation

Now, SWEET have three implementations of slicing (i.e. slicers): (1) SLV-based slicing, (2) PDG-based slicing that handles precisely the low-level memory, and (3) PDG-based slicing that does not handle low-level memory. In order to check the relative performances, we have made an experiment by running all the slicers on a number of benchmark programs and record their execution time. Our evaluation shows that among all the three slicers, the first two are very precise (have same precision) relative to the third one which is very fast even though it looses precision. The first slicer performs better than the second one when we compute one slice of an input ALF program.

1.4 Future Work

The current implementation of SLV-based slicing did not implement all the concepts mentioned in [3]. We would like to make a full implementation of SLV-based slicing and make an empirical evaluation in order to find the strengths and weaknesses of SLV-based slicing. We would also like to make an study on the incremental approach of building SLV-based slicing.

1.5 Publication

This work [3] will be published in the proceedings of the PEPM'15 ACM SIGPLAN Workshop/Symposium in Mumbai, India, January 13-14, 2015 and will be available in the ACM digital library.

References

[1] Jan Gustafsson. SWEET manual, 2011. `www.mrtc.mdh.se/projects/wcet/sweet/manual/`.

[2] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. ALF (ARTIST2 Language for Flow Analysis) specification. Technical report, Mälardalen University, Västerås, Sweden, January 2011.

[3] Björn Lisper, Abu Naser Masud, and Husni Khanfar. Static Backward Demand-Driven Slicing. In *Proceedings of the 2015 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2015, Mumbai, India, January 13-14, 2015*. ACM Press, January 2015.

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