Program Slicing for Codesign

Jeffry T Russell
Department of Electrical and Computer Engineering
University of Texas at Austin
jeff_russell@ieee.org

ABSTRACT

Program slicing is a software analysis technique that computes the set of operations in a program that may affect the computation at a particular operation. Interprocedural slicing techniques have separately addressed concurrent programs and hardware description languages. However, application of slicing to codesign of embedded systems requires dependence analysis across the hardware-software interface.

We extend program slicing for a codesign environment. Hardware-software interactions common in component-based systems are mapped to previously introduced dependences, including the interference and signal dependences. We introduce a novel *access dependence* that models a memory access side effect that results in activation of a process. A slicing algorithm that incorporates this variety of dependences is described.

1. Introduction

Program slicing is a software analysis method often used to compute the subset of program statements that may affect the computation at a particular program point. This program point, which may be defined as a statement or a particular variable used at a statement, is the *slicing criterion*. In the simple case of a single entry-single exit program, a slice is determined by finding all the transitive data and control dependences that lead to the slicing criterion. Slicing is positioned primarily as a maintenance or reuse tool for activities such as program understanding, regression testing, and function extraction from existing code.

Codesign techniques that emphasize reuse of existing subsystems can benefit from program slicing analysis as a design tool. As embedded systems density increases, system on chip solutions incorporate powerful processor cores that readily support the use of software components, typically packaged as a huge library of functions, e.g. one port of the networking subsystem for the Linux 2.4 kernel was found to contain over 200,000 lines of code in over 400 files. Often the only organization is a directory structure and only documentation comments in the code. As an interactive tool, a program slicer facilitates understanding of relevant portions of the software by directly analyzing the source code.

There is a significant body of work published on program slicing, including proposals for its application to hardware description languages. However, to the best of our knowledge, there has been no attempt to extend slicing to the hardware-software interface present in component-based embedded systems. Such an extension is needed to aid a designer in understanding the complex interaction between software device driver routines and the hardware controllers to which they interface.

The focus of this paper is to extend program slicing for use in a codesign environment. To accomplish this we identify typical dependences at the hardware-software interface, incorporate these in a graph-based representation, and then define a slicing algorithm using these dependences. We present a consistent, formal definition of the dependences based on several previous works, plus introduce the novel *access dependence*.

The remainder of the paper is organized to introduce program slicing in Section 2 and describe software-hardware interactions leading to the dependences for use in a slicing algorithm in Section 3. A case study is presented in Section 4, and finally, related slicing work is described in Section 5.

2. Background

Program slicing is a somewhat mature field, as demonstrated by published survey articles [10][15][2], reported research tools [6][1][9], and the availability of a commercial tool [2]. Program slicing was first introduced by Weiser [16] who defined a slicing criterion as any subset of program variables at a statement. The program slice consists of those statements that may affect the values of the criterion variables, including whether or not the statement executes. It is computed by iteratively solving data and control flow equations based on a control flow graph representation of a program.

An early work by Horwitz et. al [7] that formulated the slice computation as a graph reachability problem was based on a representation called a procedure dependence graph (PDG), also called a *program* dependence graph [5]. A PDG summarizes the control and data dependences of a single entry-single exit program found by analyzing its control flow graph (CFG).

In graph-based slicing, the slicing criterion is a program point, which is a node in the CFG. The slice consists of all nodes that can transitively reach the criterion node. We use the graph-based approach to define slicing for the hardware-software interface. Figure 1(a) is an intuitive example of a procedure with a slice.

2.1 Dependence Analysis

In this work, the formal representation underlying a source code specification is a control flow graph (CFG), which is defined as a flat representation, i.e. there are no basic blocks.

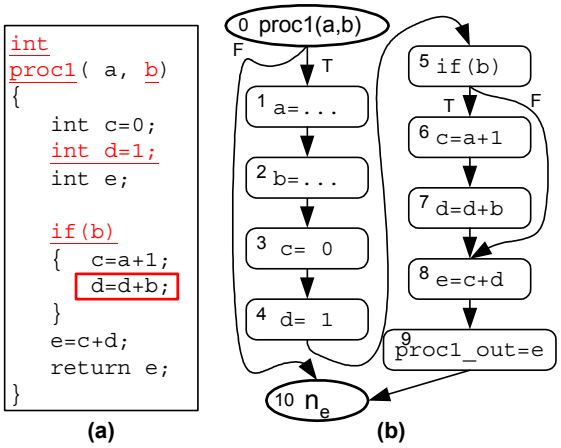

Figure 1. Procedure proc1() and its CFG. Underlined statements indicate slice from the criterion "d=d+b".

A single entry, single exit procedure **P** is represented as a *control flow graph* which is a directed graph $G=(N,E)$ where the set of nodes N contains two special nodes $n_s \in N$ and $n_e \in N$, and $\forall n \in N$ there is a walk from start node n_s to end node n_e that includes n. The nodes represent operations of **P**. Edges represent flow of control and are labeled with the condition that determines when the flow is active. The start node is a control predicate that models activation of the procedure. A CFG is shown in Figure 1(b), where the start node is labeled with the procedure name. The CFG is extended from the source code specification to include synthetic assignment operations for each formal parameter, and an assignment to a temporary variable to represent the function return value.

A node *d dominates* a node *m* in a control flow graph *G* if every path from the start node n_s to node *m* goes through *d*. A node *m* is *post-dominated* by a node *p* in directed graph *G* if every path from *m* to n_e (not including *m*) contains *p*. Referring to Figure 1(b), node 5 is immediately post-dominated by node 8, node 6 does not post-dominate a node and node 7 does not dominate a node.

Let *G* be a CFG. Let n_1 and n_2 be nodes in *G*. Node n_2 is *control dependent* on n_1 if and only if

1. there exists a path P from n_1 to n_2 with any internal node $n \in$ P post dominated by n_2; and
2. node n_1 is **not** post-dominated by n_2

Note that n_1 will always have two or more outbound control flow edges in the CFG if it is the source of a dependence relation. For the CFG in Figure 1(b), nodes 6 and 7 are control dependent on node 5, but node 8 is not control dependent on node 5 since node 8 post-dominates node 5.

Let *G* be a CFG. Let n_1 and n_2 be nodes in *G*. Node n_2 is *data flow dependent* on n_1 if and only if

1. there exists a walk *W* from n_1 to n_2 with such that no interior node $n \in W$ defines variable ***v***; and
2. variable ***v*** is defined by n_1 and used by n_2.

The procedure dependence graph (PDG) is a procedure representation that makes explicit both the data and control dependences for each operation [5]. Formally, the *procedure dependence graph* for a procedure **P** is a directed graph $G=(N,E)$. The nodes *N* represent the operations of the procedure, i.e. the nodes are the same as a control flow graph of procedure **P**. The edges *E* represent dependences between operations. Figure 2 shows a PDG. The open ended arrows are data dependences.

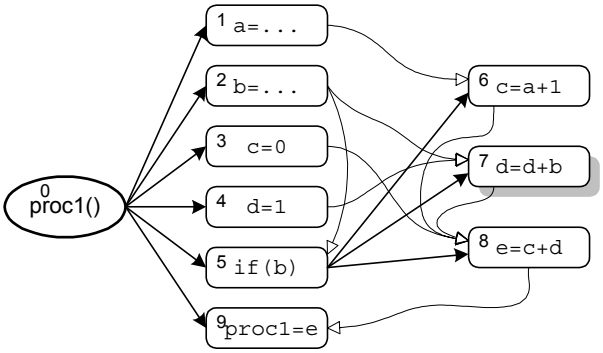

Figure 2. A program dependence graph (PDG) for proc1().

2.2 System Dependence Graph

A multiple procedure program is represented by a collection of program dependence graphs with edges between them [7]. The *system dependence graph* (SDG) for a program **Prog** is a directed graph $G=(G_{PDG}, E_{Inter})$ consisting of a set of procedure dependence graphs, G_{PDG}, and a set of augmenting edges that express interprocedural relationships between the PDGs, E_{Inter}.

To support the linking of procedures, each function call operation in a CFG is hierarchically extended into several operations as demonstrated in Figure 3. The *call* node acts as a control predicate to all other detailed operations and models control flow transfer to the called procedure. Parameters are passed using temporary variables assigned at *actual-in* nodes and output values are returned at *actual-out* nodes. Any global variables used in a called routine are modeled as input and output variables so that data dependences can be easily tracked. We assume all function calls return, so there is no need for a return edge.

In the SDG, there are three types of edges linking the hierarchically extended call site to the called procedure: (1) A *call edge* connects a call node to the entry node of the called

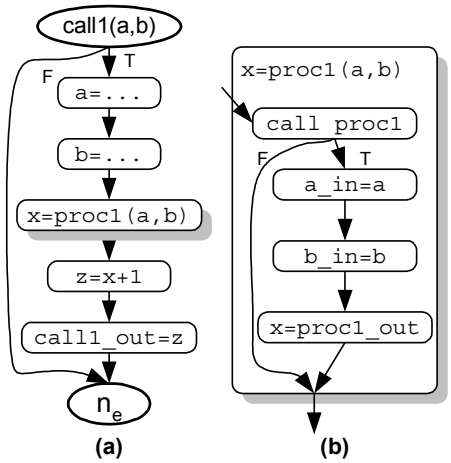

Figure 3. Hierarchical detail of a function call.

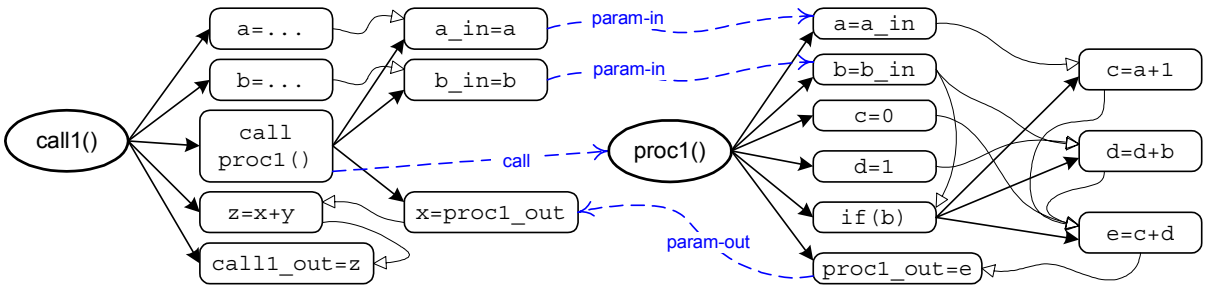

Figure 4. Function call edges in a SDG.

procedure, (2) a *param-in edge* to pass values and (3) a *param-out edge* to return values. See the example in Figure 4.

2.3 Program Slicing

A *slicing criterion* is a program point *p*, which is a node in the SDG. A *program slice* is a subgraph of the SDG that contains all nodes that may influence the computation at the slicing criterion. The slice for a criterion with multiple program points is computed as a union of slices, one for each node in the criterion.

For a program consisting of a single procedure, the SDG is a single procedure dependence graph (PDG). In this simple intraprocedural case, the slice is found by analyzing the PDG for transitive flow and control dependences from the slicing criterion. Consider the slice first shown in Figure 1 as a source code example. The corresponding PDG is shown in Figure 2, and the criterion is node 7. The dependence edges are followed (backwards) from the criterion to find nodes in the slice: the two data dependences yield nodes 2 and 4, and the control dependence edge yields node 5. All inbound edges are followed from these nodes, which all yield node 0.

In a multiple procedure SDG, the slice is found by transitively following all control, flow, and param-out edges. A goal in program slicing is to produce a more *precise slice*, which is a slice that more closely reflects the feasible paths in a program (compared to a conservative analysis that considers all paths). The call and param-in edges that are encountered are followed taking into account the calling context to improve precision. If a param-out edge had been previously followed, then the current path is in a function call. When encountered, only the matching call or param-in edge back to the call site is followed. However, if the current walk has not descended into a function call, then all call and param-in edges are followed since there is no calling context.

3. Hardware-Software Interface

The application of program slicing is important for embedded system design, especially for component-based systems. The functionality of hardware or software components can be represented as a collection of communicating processes, where each process is represented by a control flow graph. A single process specification may consist of several CFGs that represent distinct procedures that are linked via procedure calls.

The operating system is an important part of a component-based embedded system. It abstracts the system hardware into a common programming model, effectively decoupling application software from the hardware components. It also provides a multitasking runtime environment such that multiple software processes can be executed (apparently) in parallel. This enables use of common libraries, third party software modules, and applications that are portable across hardware platforms.

A key hardware component is the I/O controller (IOC) that provides an interface to the outside world. The IOC has a programming interface that is used to control data transfers between memory buffers and the external medium. An embedded system that makes use of IOCs requires software modules called device drivers that interface the hardware to the operating system.

We focus on the device driver and IOC interaction as the representative hardware-interface. The programming interface of an IOC can be viewed as a collection of memory locations and procedures.

We assume all communication is via shared memory, which may reside on a variety of physical components, e.g. the CPU, main memory, or the IOC itself. The software procedures use normal load and store operations to access shared memory, including registers on the IOC. Though implementations typically use pointer de-references to access IOC registers, we represent software access to shared memory as a variable access in the case study. The hardware has equivalent operations to access shared memory.

The IOC behavior is modeled as a collection of parallel processes specified as procedures. A procedure may be a non-halting process that controls the overall component and interacts with the registers. Other procedures may react to input, e.g. a hardware procedure that is activated based on receiving a signal or as a side effect from a memory access.

A register on an IOC can act as a communication channel similar to a function call. The software puts a message in the channel, i.e. stores a value in a register, which causes a hardware process to activate with the message as a "parameter". The software control flow continues to completion while the hardware process executes in parallel. This type of access is called signaling.

Hardware processes may also activate one another using a communication channel, i.e. using signals. Hardware may also activate a software process through an interrupt signal.

A similar, but distinct, interaction involves a software load or store to a specific memory location that actives a process. The hardware process is specified to be sensitive to an access to a particular register, and the software activates it as a side effect to the variable access. This leads to our novel access dependence.

3.1 Interface Dependences

In this section, we define three dependences that may exist between software and hardware processes: interference, signal, and the novel access dependences.

An interference dependence is a data dependence resulting from the definition and use of variables that are common to parallel executing statements [11].

Let G_1 and G_2 be CFGs with a shared variable $\boldsymbol{v}$. Let n_1 be a node G_1 and n_2 be a node in G_2. Node n_2 is *interference dependent* on n_1 if and only if

1. n_1 and n_2 may potentially execute in parallel; and
2. node n_1 defines v and node n_2 uses $\boldsymbol{v}$.

If a communication channel exists between processes such that an assignment to the channel results in the activation of a process, then there is a signal dependence [4]. Let G_1 and G_2 be CFGs with a common communication channel $\boldsymbol{w}$. Let n_1 be a node in G_1. The CFG G_2 is *signal dependent* on n_1 if and only if

1. G_1 and G_2 may execute in parallel; and
2. node n_1 writes a message to $\boldsymbol{w}$ such that G_2 may be activated.

The last type of interaction is one that we propose to model the side effect of a memory access that activates a process. Let G_1 and G_2 be CFGs. Let n_1 be a node in G_1. Let variable $\boldsymbol{v}$ be a shared variable to which G_1 can explicitly access. The CFG G_2 is *access dependent* on n_1 if and only if

1. G_1 and G_2 may execute in parallel; and
2. node n_1 uses or defines a variable $\boldsymbol{v}$; and
3. a use or definition of $\boldsymbol{v}$ may activate G_2 .

The key difference of the proposed access dependence is that no explicit communication channel exists between the procedures. The process activation is a side effect of the operation that accesses the special memory location.

For our use in a codesign environment, we expect that the designer (or component provider) annotate the CFGs with these dependences: interference, signal, and access. This is assumed both from the point of view that SDG generation is an orthogonal issue from slicing [13], and it is a practical matter that software and hardware components are specified in different source languages. Based on practical experience, it is suggested that the IOC supplier provide no more than an abstract CFG model; one that is useful for analysis but which hides details of its intellectual property.

3.2 Slicing Algorithm

The proposed algorithm for slicing shown in Table 1 is designed for conceptual simplicity, not implementation efficiency. We assume the SDG consists of multiple procedures that may execute in parallel, and there are no procedure calls across processes. The types of edges in the SDG are: control, flow, call, param-in, param-out, interference, signal, and access. The interprocedural edges (param-in, param-out, and call) are uniquely labeled for each procedure call operation, i.e. the calling context.

A marked node means it is part of the slice, and a visited node means all inbound edges have been considered for the current calling context. The set of visited nodes is tracked separately for each calling context.

A call context stack is associated with each node in the algorithm to track the particular sequence of procedure calls. When the algorithm descends into function calls (param-out edge) the current calling context, i.e. the edge label, is pushed on call stack. The algorithm ascends from a function call (call or param-in edge) if the current calling context matches the edge label, at which time the calling context is popped off stack. Initially the stack is empty, and an empty stack matches all labels. This occurs when the criterion is in the same procedure whose start node is reached, yet there are inbound call edges to follow. Since no calling context exists, all potential function calls are followed. When a new process is entered (interference, signal, or access edge), the call stack is reset, since calls only occur within a process.

Table 1. Worklist algorithm to compute a slice.

```
algrithm MarkSdg
input     Crit : node in Sdg, the slicing criterion
global    Sdg : procedure dependence graph for a procedure
declare   Visited[] : Sets of nodes visited, one set per call context
          WorkList : Set of Sdg nodes
          p, m, n : Nodes in Sdg.
          .stack : a node call context stack, initially empty.
          .context() : node method, returns current call context
          .push() : node method, puts new call context on stack
          .pop() : node method, removes call context from stack
          θ(v) : function that returns the process containing v.
begin MarkSdg
    WorkList := Crit  //initialize with slicing criterion
    while Worklist ≠ ∅ do
        Select and remove node n from WorkList
        Mark n  // part of slice
        Add n to Visited[n.context()]  // visited node for context
        foreach m ∉ Visited[n.context()] with edge (m,n) do
            if (m,n) ∈ {control, flow } then
                m.stack := n.stack
                Insert m into Worklist
            elseif (m,n) ∈ {call , param-in } then
                if  n.current() is empty then
                    Reset m.stack  // no calling context
                    Insert m into Worklist
                else
                    if  (m,n) labled with n.current() then
                        m.stack := n.stack
                        m.pop()
                        Insert m into Worklist
                    endif
                endif
            elseif (m,n) ∈ { param-out } then
                m.stack := n.stack
                m.push( (m,n) label) // saves call context
                Insert m into Worklist
            elseif (m,n) ∈ {interference } then
                if  ( θ(m), θ(n) ) is valid thread order then
                    Reset call stack for m
                    Insert m into Worklist
                endif
            elseif (m,n) ∈ {signal } then
                Reset m.stack  // no calling context
                Insert m into Worklist
            elseif (m,n) ∈ {access } then
                Mark m // in slice, but not completely visited
                foreach p with edge (p,m) of type {control }do
                    Reset p.stack  // no calling context
                    Insert p into Worklist
                endfor
            endif
        endfor
    endwhile
end MarkSdg
```

Table 2. Summary of dependence edge traversal.

Edge	Predicate to follow edge	Next set of edges to follow	Traversal state change
Control	None	All	None
Flow	None	All	None
Call	Valid call context	All	Call depth decrease
Param-in	Valid call context	All	Call depth decrease
Param-out	None.	All	Call depth increase
Inter-ference	Feasible process order	All	New process
Signal	None	All	New process
Access	None	Control only	New process

Additionally, when a process boundary is crossed following an interference edge, the process order is validated to insure a feasible execution is under consideration. The dependence edges and the actions within the algorithm are summarized in Table 2.

Note that our newly proposed access dependence may appear similar to a signal dependence when computing a slice, but only control dependences are followed from the source node (of the access dependence).

4. Case Study

Our case study demonstrates interprocedural dependences based on the transmit path for a 550 UART (universal asynchronous receiver/transmitter), an IOC component widely used both as a discrete chip and as a core in SoC designs [17]. We present an abstract set of partial CFGs in Figure 5 that represents the operations involved in the transmission of a single byte.

The CFG in Figure 5(a) is a portion of a setup software routine that writes a value to the MCR register with the bit set to enable auto flow control. The software procedure **xmit_char()** with input parameter **c** is shown in Figure 5(b). The procedure implements a busy wait on the LSR register to check the THRE bit, which indicates the transmit hold register is empty. When the bit is set, THR register is written with the byte to be sent, **c**.

The THR register is a shared variable. When accessed from outside the IOC, it has the side effect of firing the **thr_write()** procedure in Figure 5(c). This is an access dependence. This procedure clears the bit that indicates the THR is empty, and then fires the **send()** procedure in Figure 5(d) using a signal, i.e. the **enable** operation. This is a signal dependence.

The **send()** procedure reads the shared variables THR and MCR, which are interference dependent on the assignment operations in the software procedures. Likewise, the LSR register is written by the IOC which creates another interference dependence to **xmit_char()** which reads the variable. Note that the interference dependence from the MCR write in **setup()** impacts the flow of control in the hardware procedure **send()**.

The SDG that represents this system is shown in Figure 5(e). Space precludes a detailed example of a slice, but any node in the SDG can be used as a criterion to compute a slice according to the algorithm in Table 2.

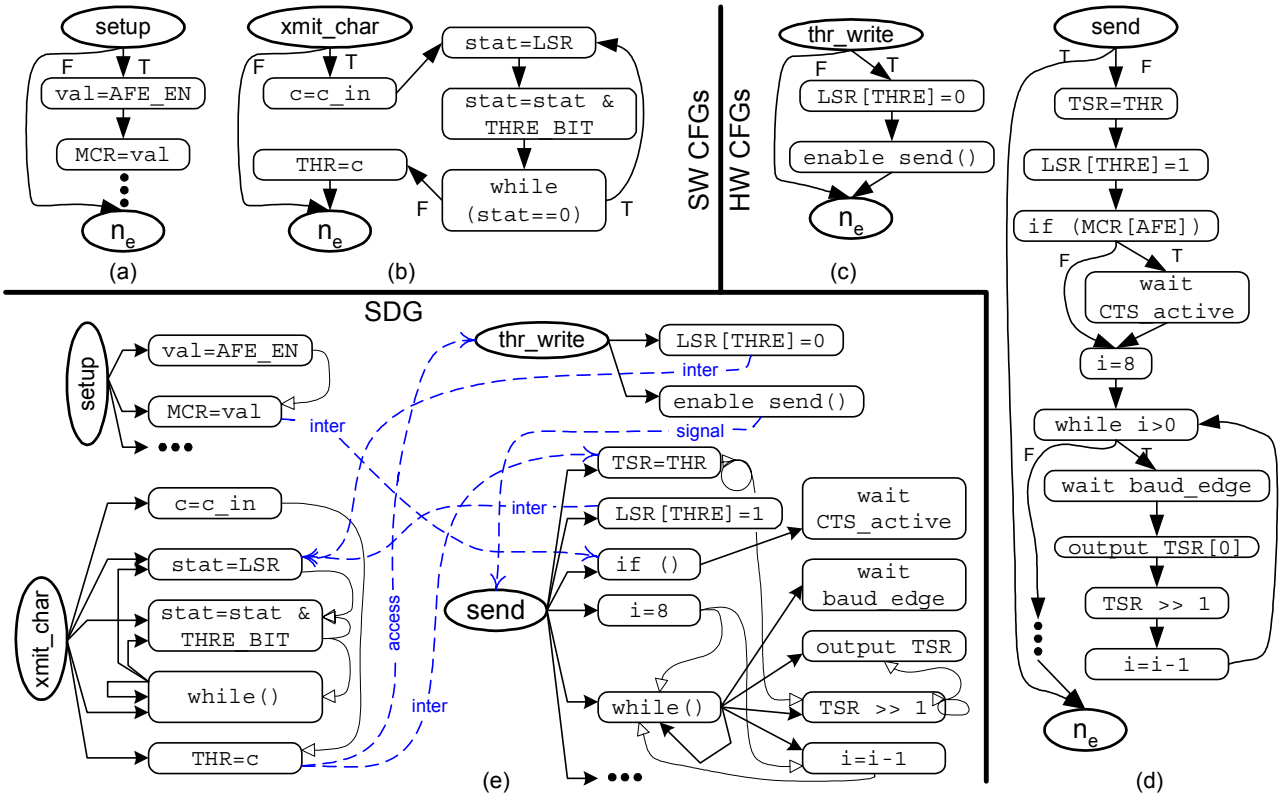

Figure 5. The case study CFGs for the device driver (a-b) and IOC (c-d) with the combined SDG(e) .

This case study demonstrates the expressiveness of the three interprocess dependences of interference, signal, and access, in addition to normal data and control dependences, to model the typical interactions across the hardware-software interface that is a central focal point of codesign activity.

5. Related Work

Program slicing is a source code analysis that was introduced as a dataflow equation problem by Weiser [16]. Interprocedural precision was improved with graph-based slicing introduced by Horwitz et. al using the system dependence graph [7]. Our base definitions, assumptions, and vocabulary are based on this work. Several works have proposed efficient algorithms for specific dependences [1][6][7][14] [13][12][11], but our algorithm purposefully excludes such advanced techniques to keep the concepts understandable.

Slicing for concurrent programs defines the interference dependence [11][12], though the previous work defines it for a single PDG. A threaded CFG was defined such that all parallel threads, which we call processes, were explicitly indicated in a single CFG, which can be analyzed to find interference dependences as well as feasible process execution order. Our technique is more general but requires a designer to explicitly indicate interference dependences and feasible process ordering between software and hardware procedures, since they do not share a common specification domain.

There is little work published regarding the application of program slicing to hardware description languages. The basics of applying slicing to VHDL descriptions were addressed by Iwaihara et. al [8] who introduced a signal dependence. They defined it as a dependence that activates an operation, while we define it as activating an entire procedure. Furthermore, their explanation seems to extend the definition to include shared variable flow dependence similar to our interference dependence.

The work by Clarke et. al [4] defines slicing for VHDL by mapping the VHDL constructs to a software control flow graph representation for which slicing tools exist [2], i.e. one that handles C programs. They map a signal dependence to a function call and introduce a synthetic master process that continuously invokes the non-halting VHDL procedures.

6. Conclusion

Program slicing was introduced using a graph-based representation of dependences call the System Dependence Graph (SDG). A set of typical interactions was defined for a software-hardware interface of a component-based system. The dependences involved in such interactions were defined with the purpose of slicing. This includes a slightly different definition for a *signal dependence* compared to previous work, as well as the novel *access dependence* to model a memory access with the side effect of firing a process.

A worklist algorithm that is conceptual clear, but not particularly efficient, was presented to demonstrate the use of the dependences to compute a slice. A partial example system based on a real IOC was described to demonstrate how the system specification appears as a SDG that can be used for slicing into both hardware and software procedures. Our future work includes the application of program slicing to identify evaluation scenarios that can be used as a basis for performance estimation in high-level codesign activities.

7. REFERENCES

[1] G. Agrawal, L. Guo, Evaluating explicitly context-sensitive program slicing, PASTE '01, June 2001.

[2] P. Anderson, R. Teitelbaum, Software inspection using CodeSurfer, *Proc of Workshop on Inspection in Software Engineering (CAV 2001)*, Paris, July 18-23, 2001.

[3] G. Canfora, A. Cimitile, A. De Lucia, G. Di Lucca, Software salvaging based on conditions, *Proc. Int. Conf. on Software Maintenance*, 1994, pp 424-433

[4] M. Clarke, P. Fujita, S. Rajan, T. Reps, S. Shankar, T. Teitelbaum, "Program slicing of hardware description languages", Proc. 10th Adv. Res. Work. Conf. Correct Hard. Design and Ver. Methods, Bad Herrenalb, Germany, 1999.

[5] J. Ferrante, K. Ottenstein, J. Warren, The program dependence graph and its use in optimization, *ACM Trans. Prog. Lang. and Sys*, v. 9, n. 3, pp. 319-349, July 1987.

[6] M. Harrold, N. Ci, Reuse-driven interprocedural slicing, *Proc 1998 Int. Conf. Software Engineering*, 1998, pp. 74-83.

[7] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, *ACM Trans. on Progr. Lang. Systems*, vol 12, no 1, Jan 1990, pp. 26-60.

[8] M. Iwaihara, M. Nomura, S. Ichinose, H. Yasuura, "Program slicing on VHDL descriptions and its applications", *Proc. 3rd Asian Pacific Conf. Hardware Description Languages*, Bangalore, Jan 1996, pp. 132-139.

[9] D. Jackson, E. Rollins, A new model of program dependencies for reverse engineering, *Proc. ACM Conf. on Foundations of Software Engineering*, Dec 1994.

[10] M. Kamkar, An overview and comparative classification of program slicing techniques, Journal Systems Sofware, 1995, Elsevier Science Inc, v 31:197-214,

[11] J. Krinke, Static slicing of threaded programs, ACM Workshop on Program Analysis for Software Tools and Engineering, 1998.

[12] M. Nanda, S. Ramesh, Slicing concurrent programs, *Proc. Int. Symp. Software Testing and Analysis*, 2000, pp 180-190

[13] T. Reps, G. Rosay, Precise interprocedural chopping, *Proc 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1995, pp. 41-52.

[14] S. Sinha, M. Harrold, G. Rothermel, System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow, *Proc. 21st Int. Conf. on Software Engineering*, May 1999.

[15] F. Tip, A survey of program slicing techniques, Journal of Programming Languages, v. 3, no. 3, pp 121-189, Sept 1995.

[16] M. Weiser, "Program slicing", *IEEE Trans Soft. Eng.*, v. 10, no. 4, July 1984, pp 352-357.

[17] Texas Instruments, TL16C550C data sheet, SLLS177F, March 2001.