Symbolic Debugging of Globally Optimized Behavioral Specifications

Inki Hong[†], Darko Kirovski[†], Miodrag Potkonjak[†], and
Marios C. Papaefthymiou[‡]
[†] Computer Science Department, University of California, Los Angeles
[‡] Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, USA

Abstract
Symbolic debuggers are system development tools that can accelerate the validation speed of behavioral specifications by allowing a user to interact with an executing code at the source level. In response to a user query, the debugger must be able to retrieve and display the value of a source variable in a manner consistent with what the user expects with respect to the source statement where execution has halted. However, when a behavioral specification has been optimized using transformations, values of variables may either be inaccessible in the run-time state or inconsistent with what the user expects.

We address the problem that pertains to the retrieval of source values for the globally optimized behavioral specifications. We describe how transformations affect the retrieval of source values. We present an approach for a symbolic debugger to retrieve and display the value of a variable correctly and efficiently in response to a user inquiry about the variable in the source specification. The implementation of the new debugging approach poses several optimization tasks. We formulate the optimization tasks and develop heuristics to solve them. We demonstrated the effectiveness of the proposed approach on a set of designs.

1 Introduction

Functional debugging of hardware and software systems has been recognized as a labor-intensive and expensive process. This situation is likely to become even worse in the future, since the key technological trends indicate that the percentage of controllable and observable variables in designs will steadily decrease. For example, the designers of a modern superscalar microprocessor reported that the debugging process took more than 40% of the development time [Uch94].

Symbolic debuggers are system development tools that can accelerate the validation speed of behavioral specifications by allowing a user to interact with an executing code at the source level. Symbolic debugging must ensure that in response to a user inquiry, the debugger is able to retrieve and display the value of a source variable in a manner consistent with what the user expects with respect to a breakpoint of the source code. The application of code optimization techniques usually makes symbolic debugging harder. While code optimization techniques such as transformations must have the property that the optimized code is functionally equivalent to the unoptimized code, such optimization techniques may produce a different execution sequence from the source statements and alter the intermediate results. In addition, some variables in the source code may disappear in the optimized code.

Debugging the unoptimized code rather than the optimized code is not acceptable for several reasons. First, it may be the case that while an error in the unoptimized code is undetectable, the error is detectable in the optimized code. Second, optimizations may be absolutely necessary to execute a program. The code without optimizations for debugging may be unable to run on a target platform, for example, because of memory limitations or constraints imposed on an embedded system. Third, a symbolic debugger for optimized code is a means to find errors in the optimizer.

In this paper we address the problem pertaining to the retrieval of source values for the globally optimized behavioral specifications. We present a *design-for-debugging* approach for a symbolic debugger to retrieve and display the value of a variable correctly and efficiently in response to a user inquiry about the variable in the source specification. We informally define the design-for-debugging problem in the following way. We are given a design or code. The code is fully specified in any high level design specification language which will be transformed to the control-data flow graph (CDFG) of the computation. The goal of our design-for-debugging (DfD) technique is to modify the original code so that every variable of the source code is debuggable (that is, controllable and observable) in the optimized program as fast as possible. At the same time, the original code must be optimized with respect to target design metrics such as throughput, latency and power consumption. A particularly important requirement is that in response to a user inquiry about a variable in the source program, the value of the variable should be retrieved or set as fast as possible.

We define an important concept for developing a method that solves the problem. The *golden cut* is defined to be the variables in the source code which should be *correct* [Hen82] in the optimizaed program. The variables are time-dependent. A variable named x at two different locations in the source program is treated as two different variables. By default, *primary inputs* and *state* or *delay variables* are included in the golden cut. The *complete golden cut* is a golden cut with the property that all variables which appear after the cut can be computed using only the variables in the cut, excluding primary inputs and state variables. An *empty golden cut* is a golden cut with no variables except for the default primary inputs and state variables in it.

Our proposed method can be described as follows. First, we should determine a golden cut. Next, in response to a user inquiry about a source variable x_t at some point t in the source program, all the variables in the golden cut that the variable x_t depends on are determined by a breadth-first search for the *source* CDFG with reversed arcs. For those variables except the primary inputs and state variables in the golden cut, all the statements that they depend on are identified by the breadth-first search for the *optimized* CDFG with reversed arcs. Those statements in the *optimized* CDFG are executed on the multi-core system-on-silicon under debugging. From this execution, we get the values of the variables in the golden cut that the variable x_t depends on. Using these values, the variable x_t is computed by the statements in the *source* CDFG on a workstation (usually uniprocessor) which runs a debugger program.

Our proposed method requires that the golden cut be chosen to result in minimum debugging time, optimal design metrics, and as complete debugging of optimized program as possible. The last requirement stems from the fact that our method executes part of the source program to get the value of a source variable in request. Because our goal is to debug the optimized program, this portion of the source program should be minimal.

1.1 Motivational Example

We illustrate the proposed method with a small motivational example shown in Figures 1, 2, and 3. The design objective is throughput optimization. The source program is shown in Figure 1. The source program consists of additions and multiplications with constants. The number of clock cycles for an iteration is 9. The number in italics next to each edge (a variable) denotes the number of operations that needs to be executed on a general purpose computer for retrieving the value of the variable. If there is no number by an edge, the value of the variable is available, because the variable is either an input (states or primary inputs) or output (states or primary outputs) variable.

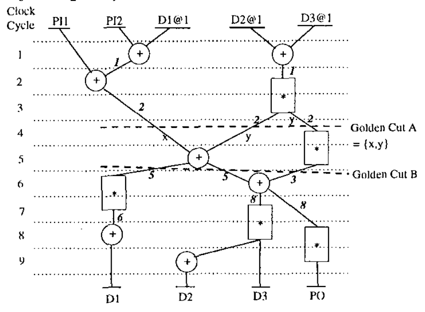

Figure 1: Part of the optimized program without considering debugging.

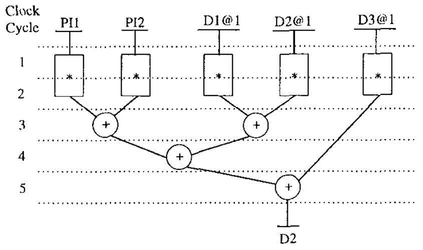

Figure 2: Part of the optimized program by our proposed design-for-debugging method.

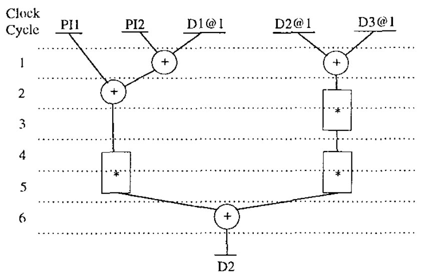

Figure 3: A motivational example for the proposed design-for-debugging method.

The original program can be optimized to execute in 5 clock cycles. Part of the optimized program (only for a state variable $D2$) is shown in Figure 2. Almost all variables in source program disappear in optimized program. For example, the variables x and y in the source program have disappeared in the optimized program. It takes 3.575 operations on average on a workstation to retrieve any intermediate variable in a source program, with the assumption that the values of all state variables for the current iteration are known. In addition to the high debugging time, debugging is performed entirely on a source program rather than its optimized version. Our proposed DfD method produces an optimized program which can execute in 6 clock cycles, while ensuring faster and more complete debugging of the optimized program. Part of the optimized program is shown in Figure 3. The golden cut chosen for our method is shown in Figure 1 labeled as *Golden Cut A*. It takes 1.125 operations on average on a workstation to retrieve any intermediate variable in a source program. If we choose the golden cut labeled as *Golden Cut B* in Figure 1, it takes 1 operation on average on a workstation to retrieve any intermediate variable in a source program while the optimized program executes in 8 clock cycles on a system-on-silicon. In this example, we have shown that the debugging of optimized program can be performed efficiently and thoroughly with minimal loss of optimization potential by the proposed DfD method.

2 Related Work

We survey the related works along two lines: CAD for debugging and symbolic debugging of optimized code. In the CAD domain recently Powley and De Groat developed a VHDL model for an embedded controller [Pow94]. The model supports debugging of the application softwares. Koch, Kebschull, and Rosenstiel [Koc95] proposed an approach for source level debugging of behavioral VHDL in a way similar to software source level debugging through the use of hardware emulation. Simulation has been used for functional debugging [Lie97]. Hennessy [Hen82] introduced the problem of debugging optimized code, defined the basic terms and presented measurements of the effects of some local optimizations. DOC [Cou88] and CXdb [Bro92] are two examples of real debuggers for optimized code which do not deal with global optimizations. Adl-Tabatabai and Gross [Adl96] discussed the problem of retrieving the values of source variables when applying global scalar optimizations. When the values of source variables are inaccessible or inconsistent, their approach just detects and reports it to a user. Our approach provides the efficient method of retrieving the values of such source variables.

3 Computational and Hardware Model

We represent a computation by a hierarchical control data flow graph (CDFG) consisting of nodes representing data operators or sub-graphs, and edges representing the data, control, and timing precedences [Rab91]. The computations operate on periodic semi-infinite streams of inputs to produce semi-infinite streams of outputs. The underlying computational model is homogeneous synchronous data flow model [Lee87] which is widely used in computationally intensive applications such as image and video processing, multimedia, speech and audio processing, control, and communications.

We do not impose any restriction on the interconnect scheme of the assumed hardware model at the RT-level. Registers may or may not be grouped in register files. Each hardware resource can be connected in an arbitrary way to another hardware resource. The initial design is augmented with additional hardware which enables controllability in the "debugging" mode. The following input operation is incorporated to provide complete controllability of a variable *Var1* using user specified input variable: Input1: if(Debug) then Var1 = Input1.

The problem of setting breakpoints is handled in the following way. A breakpoint can be set in any variable such that the execution of the program must stop immediately after performing the operation producing the breakpoint variable. Since the optimized code instead of the source code is running usually on multiprocessors, the problem of determining when to stop the execution of the optimized code for a breakpoint set in the source code is not straightforward. If the variable set as a breakpoint exists in the optimized code, the execution of the optimized code stops immediately after the control step which produces the variable. If not, we stop the execution of the optimized code immediately after the control step producing any variable which exists in both the source and optimized codes and depends on the breakpoint variable. If any one of the variables depending on the breakpoint variable is computed, then the breakpoint variable has already been computed.

4 Design for Symbolic Debugging

In response to a user inquiry about a source variable x in the source CDFG, we first need to determine if the variable x exists in the optimized CDFG. This step can be efficiently performed by keeping a list of variables that exist in both the source and optimized CDFGs. If the variable x exists in the optimized CDFG, we need to confirm if the value of the variable x is still stored in a register. Due to register sharing, the register holding the variable x may be storing a different variable at the time of the inquiry. This can be handled by checking the schedule of variables for registers. At the time of the inquiry, only the variables stored in the registers are available. If any one of the answers is negative, then the variable needs to be computed from the golden cut.

4.1 Selection of Optimal Golden Cuts

Our proposed method requires that the golden cut should be chosen to result in minimum debugging time, optimal design metrics and as complete debugging of optimized program as possible. The last requirement stems from the fact that our method executes part of the source program to get the value of a source variable in request. Because our goal is to debug optimized program, the part of the source program should be minimal. Several conflicting requirements about a golden cut can be identified. First, a golden cut should be as small as possible in order to minimize the disruption of the optimization potential of optimization techniques. Second, a golden cut should not be too small in order to minimize the debugging time. For example, an empty golden cut is the smallest golden cut that will minimize the disruption of the optimization potential, but it will result in an optimized code with long debugging time. Finally, a golden cut should be as large as possible to ensure the complete debugging of the optimized code. This requirement is satisfied by the golden cut with all the variables in the source CDFG, which results in no optimization potential to be realized. Therefore, a golden cut should be chosen by balancing all these conflicting requirements.

We consider the problem of finding the smallest complete golden cut such that every source variable can be computed by at most k operations starting from the golden cut. More formally, the problem can be defined as the following:

Problem: Given a directed acyclic hypergraph $H(V, E)$, find the smallest subset of edges, E' such that for every edge $e \in E$, a *cone* c of e with respect to E' has at most k nodes, where a *cone* c of e with respect to E' is a subset of nodes consisting of nodes on paths from all edges in E' to e.

The source program can be described by a directed acyclic hypergraph due to the requirement that a complete golden cut be chosen within one iteration of the computation. Note that the source and optimized programs in the motivational example are described by a directed acyclic hypergraph.

The pseudocode of the basic heuristic for the golden cut problem is provided in Figure 4. Intuitively the heuristic inserts "pipeline stages" in the hypergraph H so that the number of edges with pipeline registers is minimized and the size of the *cone* for each edge is less than or equal to k. The pipeline stages are inserted in sequence. Once a stage is inserted, it stays fixed.

Let $|c_{E'}(e)|$ denote the size of the cone for the edge e with respect to E'. When calculating $|c_{E'}(e)|$, we need to traverse the graph once for each edge. Thus, $O(|V||E|)$ steps are required for each pipeline stage insertion. A minimum cut for the subgraph with only green edges and their incident nodes can be optimally computed in polynomial time by a maximum flow algorithm, based on the Max-flow min-cut theorem [Cor90]. Using the method proposed by Yang and Wong [Yan94], the flow network for the subgraph is constructed as the following:

- For each hyperedge $n = (v; v_1, \cdots, v_l)$ in the subgraph, add two nodes n_1 and n_2 and connect an edge (n_1, n_2). For each node u incident on the hyperedge n, add two edges (u, n_1) and (n_2, u). Assign unit capacity to the edge (n_1, n_2) and infinite capacity to all other added edges. (see Figure 5)
- A "dummy" source node s and a "dummy" sink node t are added to the subgraph. From the source node, we add edges with infinite capacity to all the source nodes in the original subgraph. We also add edges with infinite capacity from all the sink nodes in the original subgraph to the sink.

```
Given: a directed acyclic hypergraph $H(V, E)$
    and constants $l$ and $k$

$E' = \emptyset$
Repeat
    Calculate $|c_{E'}(e)|$ of all edges
    after the most recently inserted pipe stage.
    If all $|c_{E'}(e)| \leq k$
        break
    Mark as "green" the edges with $l \leq |c_{E'}(e)| \leq k$.
    Construct a flow network for the subgraph with
    only green edges and their incident nodes.
    Find a minimum cut of the flow network using
    a maximum flow algorithm.
    $E' \leftarrow E' \cup$ {edges of the new cut}.
Return $E'$
```

Figure 4: The pseudocode of the basic heuristic for the golden cut problem.

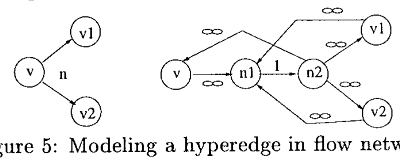

Figure 5: Modeling a hyperedge in flow network.

The construction process for an example graph is shown in Figure 6. A minimum cut of the constructed flow network can be found using various approaches such as the $O(|V||E|)$-time algorithm in [Yan94]. We use linear programming for constructing a flow network by relying on a public domain package `lp_solve` [LP]. All the "saturated" edges in the constructed flow network are added to the golden cut. To avoid trivial solutions, we use the lower bound l. The constant l is experimentally determined so that high quality golden cuts are obtained.

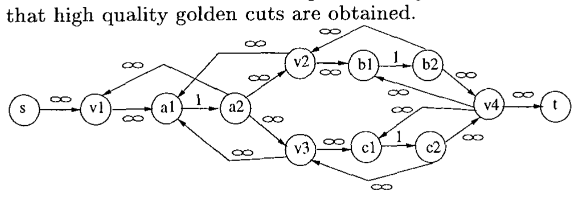

Figure 6: The construction process of a flow network for the "green" subgraph: the flow network.

Of course, the previous insertions of the pipeline stages will affect the quality of the subsequent insertions. Therefore, to further improve the heuristic, we employ the iterative improvement using the heuristic slightly modified from one described in Figure 4 as a search engine. The heuristic described in Figure 4 is modified such that the constant l is not fixed and its value is randomly chosen between 1 and k for each pipeline stage insertion. Let $Pipeline(H, k)$ be the modified heuristic for the hypergraph H with a constant k. Let $|E'|$ be the number of edges in the golden cut E'. The iterative improvement heuristic based on the heuristic $Pipeline(H, k)$ is described in Figure 7.

```
Given: a directed acyclic hypergraph $H(V, E)$
    and constant $k$

Minimum Cut = $\infty$
Repeat
    $E' = Pipeline(H, k)$
    If $|E'| <$ Minimum Cut
        Minimum Cut = $|E'|$
        Golden Cut = $E'$
Until no improvement in $c$ consecutive iterations
Return Golden Cut
```

Figure 7: The pseudocode of the iterative improvement heuristic for the golden cut problem.

5 Experimental Results

We applied our approach to design for symbolic debugging on a set of 10 small industrial examples as well as two large design examples. The smaller designs include a set of Avenhaus, Volterra, and IIR filters, an audio D/A converter, and an LMS audio formatter. Table 1 presents the experimental results for the small designs. We define *query time* as an expected time to retrieve any variable in the source program. The time is measured as average number of operations that needs to be executed for retrieving the value of a variable. Table 1 is obtained from the constraint that the value k for the linear program is set such that the final *query time* is 50%, 25%, or 12.5% of the initial *query time*). The average golden cut size with respect to the number of variables was 4.99%, 10.49%, and 19.26%, respectively.

The two large designs include the JPEG codec from the Independent JPEG Group and the European GSM 06.10 provisional standard for full-rate speech transcoding, prI-ETS 300036, which uses residual pulse excitation/long term prediction coding at 13 kbit/s. Table 2 presents the experimental results for the large designs. For the same set of *query time* constraints, the average golden cut size with respect to the number of variables was 2.83%, 6.07% and 12.72%, respectively. None of the examples resulted in runtimes of the linear programmer larger than a minute.

6 Conclusion

We addressed the problem related to the retrieval of source values for the globally optimized behavioral specifications. We explained how transformations affect the retrieval of source values. We presented an approach for a symbolic debugger to retrieve and display the value of a variable correctly and efficiently in response to a user inquiry about the variable in the source specification. The implementation of the new debugging approach posed several optimization tasks. We formulated the optimization tasks and developed efficient algorithms to solve them. The effectiveness of the proposed approach was demonstrated on a set of designs.

Design	Variables in CDFG	G. Cut Size 1	G. Cut Size 2	G. Cut Size 3
12th order IIR	56	3	5	9
Avenhaus direct	40	2	5	9
Avenhaus cascade	34	2	4	8
Avenhaus parallel	39	2	5	9
Avenhaus continued	35	2	5	9
Avenhaus ladder	50	3	6	11
DAC	354	7	15	28
2nd order Volterra	29	2	4	7
3rd order Volterra	50	3	5	9
LMS formatter	464	9	21	45

Table 1: Golden Cut Sizes 1, 2, and 3 are obtained for values of k in the linear program, such that the final query time is 0.5, 0.25, and 0.125 of initial query time, respectively.

Design	Variables in CDFG	G. Cut Size 1	G. Cut Size 2	G. Cut Size 3
JPEG encoder	4806	120	234	501
JPEG decoder	4269	105	229	453
GSM encoder	3291	98	206	417
GSM decoder	2556	87	199	439

Table 2: Golden Cut Sizes 1, 2, and 3 are obtained for the value k in the linear program, such that the final query time is 0.5, 0.25, and 0.125 of initial query time, respectively.

References

[Adl96] A.-R. Adl-Tabatabai and T. Gross. Source-level debugging of scalar optimized code. SIGPLAN Notices, vol.31, (no.5), May 1996. p.33-43.

[Bro92] G. Brooks, G.J. Hansen, and S. Simmons. A new approach to debugging optimized code. SIGPLAN Notices, vol.27, (no.7), July 1992. p.1-11.

[Cor90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. Introduction to algorithms. McGraw-Hill, c1990

[Cou88] D.S. Coutant, S. Meloy, and M. Ruscetta. DOC: a practical approach to source-level debugging of globally optimized code. SIGPLAN Notices, vol.23, (no.7), July 1988. p.125-34.

[Hen82] J. Hennessy. Symbolic debugging of optimized code. ACM Transactions on Programming Languages and Systems, vol.4, (no.3), July 1982. p.323-44.

[Koc95] G. Koch, U. Kebschull, and W. Rosenstiel. Debugging of behavioral VHDL specifications by source level emulation. European Design Automation Conference, 1995. p.256-61.

[Lee87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. Proceedings of the IEEE, vol.75, (no.9), Sept. 1987. p.1235-45.

[Lie97] C. Liem, et al. System-on-a-chip cosimulation and compilation. IEEE Design & Test of Computers, vol.14, (no.2), April-June 1997. p.16-25.

[LP] ftp://ftp.es.ele.tue.nl/pub/lp_solve

[Pow94] G.S. Powley and J.E. DeGroat. Experiences in testing and debugging the i960 MX VHDL model. Proceedings of VHDL International Users Forum, 1994. p.130-5.

[Rab91] J.M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath-intensive architectures. IEEE Design & Test of Computers, vol.8, (no.2), June 1991. p.40-51.

[Uch93] K. Uchiyama, et al. The Gmicro/500 superscalar microprocessor with branch buffers. IEEE Micro, vol.13, (no.5), Oct. 1993. p.12-22.

[Yan94] H. Yang and D.F. Wong. Efficient network flow based min-cut balanced partitioning. IEEE/ACM International Conference on Computer-Aided Design. 1994. p.50-5.