SYMBOLIC DEBUGGING SCHEME FOR OPTIMIZED HARDWARE AND SOFTWARE

Farinaz Koushanfar†, Darko Kirovski‡, and Miodrag Potkonjak†

†Electrical Engineeing and Computer Science Departments, University of California, Los Angeles, CA 90095

‡ Microsoft Research, One Microsoft Way, Redmond, WA 98052

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 retrieves the value of a source variable in a manner consistent with respect to the source statement where execution has halted. However, when a behavioral specification has been optimized using transformations, values of variables may be inaccessible in the run-time state.

We have developed a set of techniques that, given a behavioral specification $CDFG$, enforce computation of a selected subset V_{cut} of user variables such that *(i)* all other variables $v \in CDFG$ can be computed from V_{cut} and *(ii)* this enforcement has minimal impact on the optimization potential of the computation. The implementation of the new debugging approach poses several optimization tasks. We have formulated the optimization tasks and developed heuristics to solve them. The effectiveness of the approach has been demonstrated on a set of benchmark designs.

1. INTRODUCTION

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 [Hen82]. 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 respect to a breakpoint in the source code. 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. Debugging unoptimized rather than optimized code is not acceptable for several reasons , including:

- while an error in the unoptimized code is undetectable, it is detectable in the optimized code,
- optimizations may be necessary to execute a program due to hardware limitations, and
- a symbolic debugger for optimized code is often the only tool for finding errors in an optimization tool.

We propose a *design-for-debugging* (DfD) approach that enables retrieval of source values for a globally optimized behavioral specification. The goal of the DfD technique is to modify the original code in a pre-synthesis step such that every variable of the source code is controllable and observable in the optimized program. More formally, given a source behavioral specification (represented as a control data flow graph [Rab91]) $CDFG$, the goal of the DfD approach is to enforce computation of a selected subset $V_{cut} \in CDFG$ (cut) of user variables such that:

- all other variables $V \in CDFG$ can be computed from the cut V_{cut} [Kir99], and
- the enforcement of computation of user defined V_{cut} variables has minimal impact on the optimization potential.

Finding a cut of a computation has been addressed in many debugging [Kir99] and software checkpointing [Ziv98] tools. However, symbolic debugging imposes a new constraint for the cut selection procedure: *variables enforced to be computed should not restrict the optimization process.*

The developed DfD technique analyzes the source code and selects the cut variables according to a number of heuristic policies. Each policy quantifies the likelihood that a particular variable is not computed due to a specific transformation of the computation. In order to support fully modular pre-processing, explicit computation of selected cut variables V_{cut} is enforced by assignment of each variable $v_i \in V_{cut}$ to a primary output. Thus, application of any synthesis tool would result in an optimized behavioral specification $CDFG_o$, which necessarily contains the selected cut variables $V_{cut} \in CDFG_o$. At debugging time (simulation or emulation), the symbolic debugger monitors the values of cut variables. In response to a user inquiry about a source variable $v_i \notin V_{cut}, CDFG_o$ and $v_i \in CDFG$, all the variables in the cut that the variable v_i depends on, are determined by a breadth-first search of the source $CDFG$ with reversed arcs. Using these values, variable v_i is computed using the original $CDFG$.

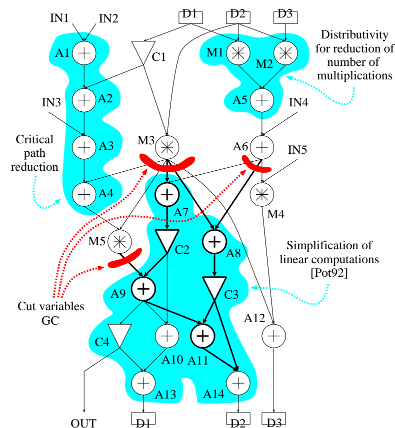

Figure 1: Trade-offs related to selection of cut variables.

The developed DfD technique poses a number of optimization tasks. Using the following motivational example, we show that de-

bugging of an optimized behavioral specification can be performed efficiently. For brevity and expressiveness, we have constructed an abstract computation, illustrated in Figure 1. Expected optimization steps are applied to shaded areas in the figure as follows:

- additions $A1$ through $A4$ can be compacted in a tree of additions for critical path reduction,
- number of multiplications can be reduced by applying the distributivity rule to multiplications $M1$ and $M2$ and addition $A5$ (area optimization), and
- all operations in the remaining shaded area can be optimized using the fast linear computation procedure[Pot92].

We define an important concept that enables effective DfD. A *golden cut* is defined as a subset of variables in the source code, which should be *correct* [Hen82] in the optimized program. A *complete golden cut* V_{cut} is a golden cut with the property that all user variables and primary outputs in the computation can be computed using only the cut variables and the primary inputs [Kir99].

Variables of an example complete golden cut GC (output of $M3$, $M5$, and $A6$) are depicted in Figure 1. Using the variables in GC and the primary inputs, any other variable in the computation can be computed. For example, consider the input variables of addition $A14$. Bold lines in Figure 1 illustrate the sequence of operations to be executed in order to calculate the results of $A11$ and $C3$ solely by using the variables in the cut. Since the selected cut variables are not a result of operations that can be involved in the abovementioned optimizations, their selection yields efficient debugging accompanied with effective design implementation.

Highly inefficient cut can be constructed using the output variables of $M1$, $M2$, $A2$, $A9$, $C2$, and $A8$. Besides larger cardinality of the involved subset of variables, this unfortunate selection also disables all possible optimizations, thus resulting in a poor implementation. In general, all possible optimizations are not known in the pre-processing DfD phase. Therefore, we propose a cut selection process that is guided using heuristics that determine the likelihood that an operation can be involved in a transformation.

2. RELATED WORK AND COMPUTATIONAL MODEL

Transformations alter the structure of a computation in a such a way that the user specified I/O relationship is maintained. For formal definitions of all related transformations, we refer to the standard compiler reference works [Aho77]. Hennessy has categorized and presented models to describe the effects of local and global optimizations on symbolic debugging of program variables [Hen82]. Adl-Tabatabai and Gross [Adl96] discussed the problem of retrieving 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. Kirovski et al. have proposed a DfD technique that modifies the original code so that every variable is controllable and observable in the optimized program as fast as possible [Hon00]. However, they did not address the problem of a potential significant impact of their DfD technique to the optimized design metrics.

The developed debugging approach is not limited to a specific computation model. However, for each computation model, the definition of a complete golden cut has to be established to satisfy the generic concept: a golden cut at time T is defined as a subset of variables from which any other variable computed after T can be computed. For the sake of conceptual simplicity, we target the synchronous data flow (SDF) model of computation [Bha93].

3. DESIGN FOR SYMBOLIC DEBUGGING

As a compilation pre-processing step, the developed DfD technique analyzes the original $CDFG$ in order to select a complete golden cut GC which is optimization-friendly. Upon selection, the DfD procedure augments the original specification with statements that enforce computation of golden cut variables. If the DfD approach is part of an optimizing compiler, this step can be performed by marking variables. An independent modular DfD technique would achieve the same goal by specifying the golden cut variables as output variables. Once computation of the golden cut variables is assured, the modified $CDFG_m$ is processed by a synthesis tool. The result of this process is an optimized $CDFG_o$ with guaranteed existence of golden cut variables.

While monitoring code execution, the symbolic debugger scans for values of golden cut variables and stores them in designated buffers. Since the computation of a single source variable may involve values of golden cut variables from several iterations [Kir99], the depth of each buffer can be larger than one. The expectation is that the cardinality of cut variables is much smaller than the cardinality of variables in the source $CDFG$ [Kir99]. Therefore, the memory overhead for golden cut maintenance is in general low.

While debugging, at a specific breakpoint the user inquires about a source variable v_i. Initially, the symbolic debugger determines if v_i exists in the optimized $CDFG_o$. This step can be efficiently performed by keeping a list of variables that exist in both the source and optimized CDFGs. If the variable does not exist in the optimized code $CDFG_o$, then its value is computed from the golden cut. All the variables in the cut that the variable v_i depends on, are determined by a breadth-first search of the source $CDFG$ with reversed arcs. Finally, we compute variable v_i using the cut values and the statements from the original $CDFG$.

The effectiveness of the developed symbolic debugging approach depends strongly on the selection of golden cut variables. In this section, we identify the trade-offs involved in golden cut determination under different optimization constraints. Next, we establish the complexity of the cut selection problem and provide an algorithm for its solution. Finally, we discuss how certain transformations can affect the cut selection, resulting in cut invalidation.

The adopted definition of a complete golden cut [Kir99] ensures that any variable in the original specification can be computed from its cut. However, it does not guarantee that the modified specification can be optimized as effectively as the original one. To address this issue, the search for a computation cut has to reflect the trade-offs involved with potential optimizations. The developed DfD approach does not assume that a particular optimization will be performed, but heuristically quantifies the likelihood that a particular variable will disappear during the optimization process.

We propose a set of heuristics that identify variables that are likely to be used in generic, area, and throughput optimizations. Low-power constraints can be usually described as a superposition of transformations for area and throughput [Dey99]. The set of criteria for optimization-sensitive cut selection is incorporated into the search process using an objective function $\Phi(v_i, CDFG)$. This function attempts to quantify for each variable v_i the likelihood that v_i disappears during the synthesis process.

$$\Phi(v_i, CDFG) = \alpha \cdot |GC| + fanout(v_i) + testLinear(v_i) + test_\epsilon_CP(v_i) + testDistributivity(v_i) + testParallelism(v_i) + \beta \cdot \frac{max(overlap(v_i, N(v_i)))}{LT(v_i) \cdot LT(N(v_i))} + testInputsInCycles(v_i)$$

The components of the objective function return quantifiers that represent the trade-offs involved in decision making for inclusion of a variable in a complete golden cut. Values of quantifiers are determined experimentally in a learning process or according to designer's experience and optimization goals. Each component

corresponds to the following optimization objective:
$|GC|$ - *Small cardinality of the golden cut.*
$fanout(v_i)$ - *Operations with high fanout.* If the result v_i of an operation O_i is used as an operand in a relatively large number of different operations, then it is hard to apply transformations to the original $CDFG$ such that v_i disappears from it. Thus, it is highly desirable to include v_i in a complete golden cut.
$testLinear(v_i)$ - *Non-linear operations.* It has been demonstrated that a collection of linear operations (addition, subtraction, multiplication with a constant, etc.) can be transformed optimally for a particular design metric [Pot92]. Hence, operands or results of non-linear operations should be given preference. $test_\epsilon_CP(v_i)$ - *ϵ-critical path.* Transformations for almost all design metrics, frequently severely modify the critical path of the computation, as the critical path usually limits the performance of a circuit. The golden cut selection routine should avoid including variables at most ϵ operations close to the critical path.

The following optimization indicators have been considered for area minimization:
$testDistributivity(v_i)$ - *Enable distributivity.* Low priority for cut selection is given to variables that can be involved in applying distributivity among operations. Since distributivity is the key enabler of reducing expensive operations, e.g. multiplications, it is of utmost importance not to disable this transformation.
Disable scheduling expensive operations in the same control step. Special attention is paid to expensive operations that have short and overlapping life-times $LT(v_i)$ ($LT(v_i) = ALAP(v_i) - AEAP(v_i)$; AEAP - as early as possible - and ALAP - as late as possible). We denote the subset of variables with lifetimes overlapping the lifetime of v_i as $N(v_i)$. If the overlap $overlap(v_i, N(v_i))$ among them is relatively large with respect to the lifetimes of considered variables, then the cut selection procedure should avoid inclusion of v_i in the cut.
$testParallelism(v_i)$ - *Enable reduction of parallelism.* Transformations such as loop folding and loop merging modify the computation in such a way that blocks of parallel operations are merged into a single instruction block with lower degree of parallelism. Obviously, reduction of parallelism can reduce circuit's area at the expense of increased clock speeds.

We consider only one transformation for maximizing throughput: $testInputsInCycles(v_i)$ - *Number of inputs in cycles.* Cycles with higher number of primary input variables have to be carefully cut, since input operations can be commonly extracted from the loop and processed as a highly pipelined structure. This transformation can significantly increase the throughput of the system.

PROBLEM: The Complete Golden Cut.

INSTANCE: *Given an unscheduled and unassigned control data flow graph $CDFG(V, E)$ with each node v_i weighted according to $\Phi(v_i, CDFG)$ and real number K.*

QUESTION: *Is there a subset of variables GC, such that when removed from the $CDFG$ leaves no directed cycles and the sum of weights $\sum_{v_i \in GC} \Phi(v_i, CDFG)$ is smaller than K?*

The specified problem is an NP-complete problem since there is an one-to-one mapping between the special case of this problem, when the weights on all nodes are equal, and the FEEDBACK ARC SET problem [Gar79]. The developed heuristic algorithm for this problem is summarized using the pseudo-code in Figure 2. The heuristic starts by logically partitioning the graph into a set of strongly connected components (SCCs) using the breadth-search algorithm [Cor90]. All trivial SCCs which contain exactly one vertex are deleted from the resulting set since they do not form cycles. Then, the algorithm iteratively performs several processing steps on each of the non-trivial SCCs.

At the beginning of each iteration, to reduce the solution search space, a graph compaction step is performed. In this step each path $P : A \rightsquigarrow B$ which contains only vertices $V \in P, V \neq A$ with exactly one variable input is replaced with a new edge $E_{A,B}$ which connects the source A and destination B and represents an arbitrary selected edge (variable) of the same path. Nodes A and B inherit the maximum weight among its current weight and all the nodes removed from the CDFG due to the compaction process using edge $E_{A,B}$.

In the next step, the algorithm decides which node in the current set of SCCs is to be deleted. The algorithm makes its decision based on the cardinality of the newly created set of SCCs and the sum of objective functions of the currently selected cut. The vertex that results in the largest overall objective function is removed from the set of nodes as well as all adjacent edges. The deleted vertex is added to the resulting cut-set. The process of graph compaction, evaluation of node deletion, node deletion, and graph updating is repeated until the set of non-trivial SCCs in the graph is empty. The set of nodes deleted from the computation represents the final cut-set selection.

```
Create a set SCC = ComputeScc(CDFG(V,E)) of SSCs [Cor90]
For each SCC_i ∈ SCC
  If |SCC_i| = 1 delete SCC_i from SCC
-----------------------------------------------------------
CUT = null
While SCC ≠ empty
  For each SCC_i ∈ SCC
    GraphCompaction(SCC_i)
    For each node v_j ∈ SCC_i
      S = ComputeScc(SCC_i − v_j)
      OF(S, v_j) = Σ_{i=1}^{S} Φ(v_j, CDFG)
    Select vertex v_j which results in max OF(S, v_j)
    Delete v_j from SCC_i
    SCC = S(V_j)
    For each SCC_i ∈ SCC
      If |SCC_i| = 1 delete SCC_i from SCC
    CUT = CUT ∪ v_j
-----------------------------------------------------------
Procedure GraphCompaction(SCC_i)
  For each vertex v_j ∈ SCC_i
    If v_j has exactly one input edge E_{j,k} with a source in v_k
      For each edge E_{k,m}
        Create edge E_{j,m} and delete E_{k,m}
      Weight(v_j) = max(Weight(v_j), Weight(v_k))
      Delete v_k
```

Figure 2: Pseudo-code for the developed algorithm for The Complete Golden Cut problem.

Consider the example shown in Figure 3. The CDFG of the third order Gray-Markel IIR filter, shown in Figure 3a, has one non-trivial SCC. The graph compaction step is explained in Figure 3b, where vertex B is merged with vertex A as well as variable W is merged with variable V. In Figure 3c an example of node deletion is described. The deleted node creates two smaller SCCs.

Once the DfD procedure modifies the source code $cdfg_m = DfD(cdfg)$, a synthesis tool ST is applied in order to generate the final optimized specification $cdfg_o = ST(cdfg_m)$. In general, the synthesis tool should have the freedom to perform arbitrary transformations on the source computation. The question that can be posed is: does there exist such a set of transformations ST, which translates the source specification $cdfg_m$ with an enforced complete golden cut GC into a new specification $cdfg_o$, where the enforced cut GC is not a complete cut?

Several examples of computation structures of different implementations of the same computational functionality (for example: the Gray-Markel ladder, cascade, and parallel IIR filters) clearly indicate that there exist such transformations that enforce a given cut in one specification not to satisfy the cut properties in the transformed specification. However, the sophistication of such algorithmic transformations is far from being met by any published synthesis tool. Therefore, it is not expected that the structure of the computation is changed drastically during optimization.

There exist transformations performed by common compilers (e.g. loop, folding, and unfolding), which modify the loop structure of the computation. However, all of these transformations preserve the completeness of a cut selected in the DfD phase.

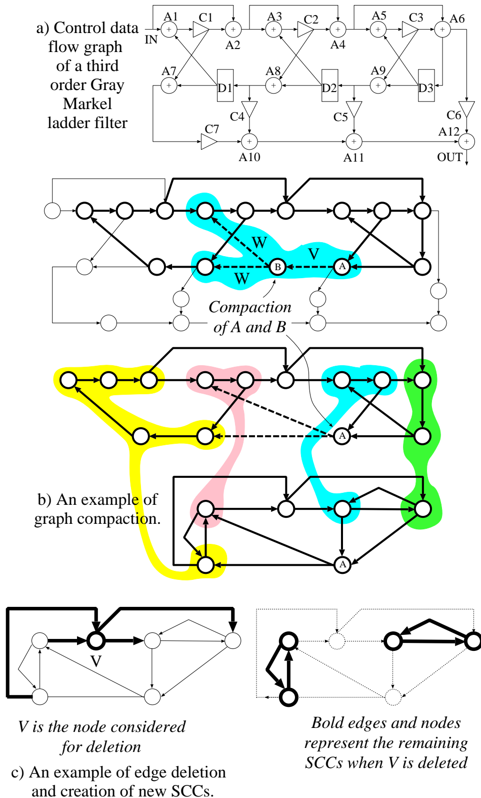

Figure 3: Performing the steps of a single iteration of the cut-set selection procedure.

4. EXPERIMENTAL RESULTS

Proper evaluation of the proposed debugging techniques is a complex problem due to a great variety of optimization steps which can be undertaken during design optimization using transformations. In order to address this concern, we have applied the new technique on more than hundred designs from the Hyper and Mediabench benchmark suites. We have used the following transformations: associativity, commutativity, distributivity, zero and inverse element laws, retiming, pipelining, loop unfolding and folding, constant propagation, substitution of constant multiplication with shifts and additions, and common subexpression elimination and replication. *On the overwhelming number of designs, our technique did not incur any cost, regardless of targeted optimization goals: area, throughput, or power.*

The detected exceptions are tabulated in Table 1. On these examples, we applied retiming for joint optimization of latency and throughput, and then, maximally fast script for linear computations. The designs augmented with additional debugging constraints were able to produce the best combination of latency and throughput. However, on some of them notable area overhead was induced due to the added constraints. Closer analysis of this examples indicates that the symbolic constraints induced a need for computation of additional variables used only for debugging purposes. It can be concluded that although it is possible to find examples with additional overhead due to enforced computation of the golden cut, such cases occur rarely.

Design	ICP	OCP	GC	IArea	OArea	Area OH
dist	7	4	4	7.96	8.23	3.5%
chemical	6	3	3	25.56	26.33	3%
5WDF	17	5	5	81.55	85	4%
avenhaus	11	5	4	49.90	60.38	21%
10IIR	12	5	5	55.42	68.15	23%
band-pass	20	5	6	172.45	214	24%
modem	25	6	11	238.01	330.3	40%
DAC	58	3	3	42.99	43.09	0.2%

Table 1: Comparison of areas of designs optimized with and without the DfD phase. ICP - initial critical path; OCP - critical path after optimization; GC - cardinality of the complete golden cut; IArea - optimized design area without DfD; OArea - optimized design area with DfD; Area OH - the area overhead.

5. CONCLUSION

In response to a user query, a symbolic debugger retrieves the value of a source variable. However, when a behavioral specification is optimized using transformations, values of variables may either be inaccessible in the run-time state. We proposed a set of techniques that, given a behavioral specification, enforce computation of a selected subset of user variables such that all other variables can be computed from this subset. The effectiveness of the proposed approach has been demonstrated on a set of benchmark designs.

6. REFERENCES

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

[Aho77] A.V. Aho and J.D. Ullman. Principles of Compiler Design. Addison-Wesley, Reading, MA, 1977.

[Bha93] S.S. Bhattacharyya and E.A. Lee. Scheduling synchronous dataflow graphs for efficient looping. Journal of VLSI Signal Processing, vol.6, (no.3), pp.271-88, 1993.

[Cor90] T.H. Cormen, et al. Introduction to algorithms. McGraw-Hill, 1990

[Dey99] S. Dey, et al. Controller-based power management for control-flow intensive designs. TCAD, vol.18, (no.10), pp.1496-508, 1999.

[Dun92] P. Duncan, et al. Hi-Pass: A Computer Aided Synthesis System for Fully Parallel Digital Signal Processing ASICs. ICASSP, pp. V-605-608, 1992.

[Gar79] M.R. Garey and D.S. Johnson. Computers and intractability. W. H. Freeman, San Francisco, CA, 1979.

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

[Kir99] D. Kirovski, et al. Improving the Observability and Controllability of Datapaths for Emulation-based Debugging. TCAD, 1999.

[Hon00] I, Hong et al. Symbolic Debugging of Globally Optimized Behavioral Specifications. ASP-DAC, 2000.

[Pot92] M. Potkonjak and J. Rabaey. Maximally fast and arbitrarily fast implementation of linear computations. ICCAD, pp.304-8, 1992.

[Rab91] J. Rabaey, et al. Fast Prototyping of Datapath-Intensive Architectures. Design and Test of Computers, vol.8, (no.2), pp.40-51, 1991.

[Ziv98] A. Ziv and J. Bruck. Analysis of checkpointing schemes with task duplication. Trans. on Computers, vol.47, (no.2), pp.222-7, 1998.