# Symbolic Debugging of Embedded Hardware and Software

Farinaz Koushanfar, Darko Kirovski, Inki Hong, Miodrag Potkonjak, *Member, IEEE*, and Marios C. Papaefthymiou, *Member, IEEE*

***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 retrieve and display the value of a source variable in a manner consistent with user expectations 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 runtime state or inconsistent with user expectations. We address the problem that pertains to the retrieval of source values for the globally optimized behavioral specifications. We present a new approach for symbolic debugging. The implementation of the new debugging approach poses several optimization tasks. We formulate the optimization tasks and develop heuristics to solve them. We demonstrate the effectiveness of the proposed approach on a set of designs.**

***Index Terms*—Design automation, design for testability.**

## I. Introduction

FUNCTIONAL debugging of hardware and software systems has emerged as a dominant step with respect to time and cost of the development process. For example, debugging (architecture and functional verification) of the UltraSPARC-I took twice as long as its design [22]. The difficulty of verifying designs is likely to worsen in the future, since the key technological trends indicate that the percentage of controllable and observable variables in designs will steadily decrease. The Intel development strategy team foresees that a major design concern for their year-2006 microprocessor will be the need to exhaustively test all possible computational and compatibility combinations [23].

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 [7]. Symbolic debugging must ensure that in response to a user inquiry, the debugger will retrieve and display the value of a source variable in a manner consistent with user expectations with respect to a breakpoint in 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 nonoptimized code, such optimization techniques may produce a different execution sequence from the source statements and alter the intermediate results. Debugging nonoptimized rather than optimized code is not acceptable for several reasons.

1) While an error in the nonoptimized code is undetectable, it is detectable in the optimized code.
2) Optimizations may be necessary to execute a program due to memory limitations or other constraints imposed on an embedded system.
3) A symbolic debugger for optimized code is often the only tool for finding errors in an optimization tool.

There are many similarities between debugging a behavioral specification for hardware–software embedded systems and debugging source-level software. Nevertheless, there are also numerous differences that make embedded hardware and software significantly more demanding and more important. First, optimizations are much more important and, therefore, more aggressively applied to embedded systems due to their cost, speed, and power sensitivity. Therefore, a significantly smaller percentage of variables from the behavioral specification is preserved from the behavioral specification in the final implementation. This imposes additional constraints on symbolic debugging. Second, embedded systems often have numerous parts implemented using fixed-point arithmetic that are significantly more sensitive to numerical stability. This also makes debugging more important and difficult. Furthermore, the development time of embedded systems is most often significantly shorter than for software, again implying higher importance of fast symbolic debugging. Finally and most importantly, embedded systems programs often follow semantics such as synchronous data flow and network processes that imply infinite nonterminating execution of the program. The need for tracking streams of data instead of individual values makes symbolic debugging more complex and more important.

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* (DfD) 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 DfD problem in the following way. We are given a design or code. The code is fully specified in any high-level design specification language that will be transformed to the control data flow graph (CDFG) of the computation. The goal of our 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

Manuscript received September 25, 2000. This paper was recommended by Associate Editor R. Karri.

F. Koushanfar, D. Kirovski, I. Hong, and M. Potkonjak are with the Computer Science Department, University of California, Los Angeles, CA 90024 USA.

M. C. Papaefthymiou is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA.

Publisher Item Identifier S 0278-0070(01)01512-3.

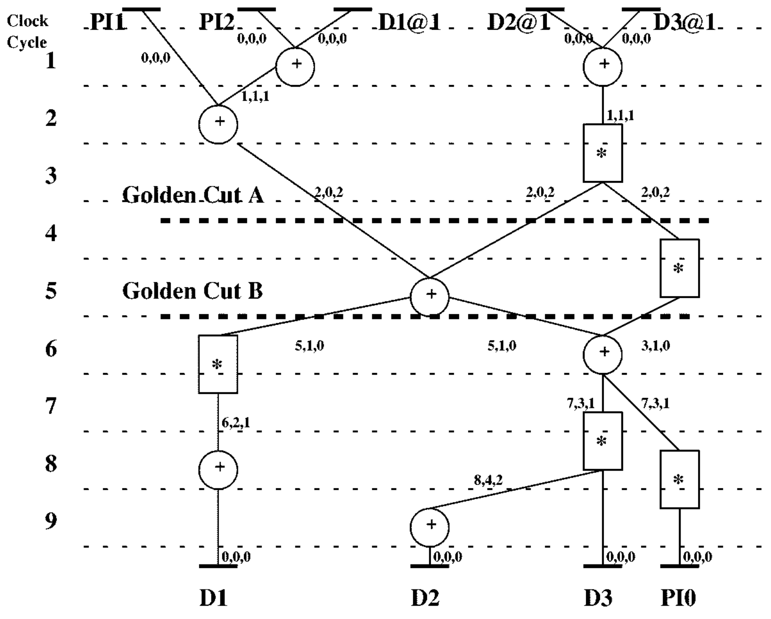

Fig. 1. Part of the optimized program without considering debugging. Three numbers shown on each edge of the CDFG correspond to the number of operations required for their computation from three golden cuts: state variables only, state variables and the golden cut A, and state variables and the golden cut B.

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 that should be *correct* [7] in the optimized 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 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 in 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 in the *optimized* CDFG with reversed arcs. Those statements in the optimized CDFG are executed on the multicore system-on-silicon under debugging. From this execution, we get the values of the variables in the golden cut on which the variable $x_t$ depends. 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 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.

## *A. Motivational Example—Symbolic Debugging with Fast Variable Computation*

We illustrate the proposed method with a small motivational example shown in Figs. 1, 2, and 3. The design objective is throughput optimization. The source program is shown in Fig. 1. The source program consists of additions and multiplications with constants. The number of clock cycles for an iteration is nine. 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.

The original program can be optimized to execute in five clock cycles. Part of the optimized program (only for a state variable $D2$) is shown in Fig. 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. The proposed DfD method produces an optimized program which can execute in six clock cycles while ensuring faster and more complete debugging of the optimized program. Part of the optimized program is shown in Fig. 3. The golden cut chosen for our method is shown

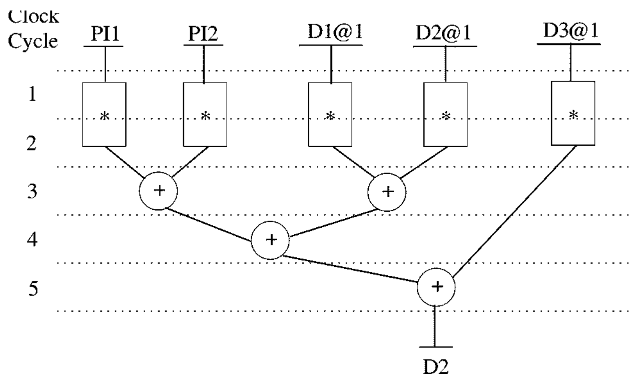

Fig. 2. Part of the optimized program by our proposed DfD method.

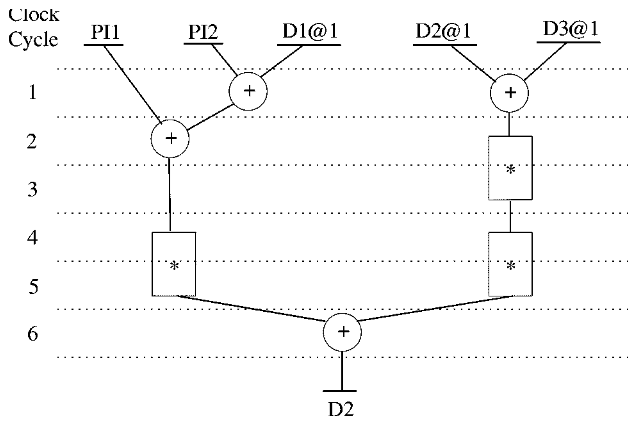

Fig. 3. Motivational example for the proposed DfD method.

in Fig. 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 Fig. 1, it takes one operation on average on a workstation to retrieve any intermediate variable in a source program while the optimized program executes in eight clock cycles on a system-on-silicon.

## *B. Motivational Example—Symbolic Debugging with Minimal Impact on Computation Performance*

Using the following motivational example, we show that debugging of an optimized behavioral specification can be performed efficiently and thoroughly with minimal loss of optimization potential by the proposed DfD method for minimal impact on computation performance. For brevity and expressiveness, we have constructed an abstract computation illustrated in Fig. 4, which demonstrates the tradeoffs involved in selecting cut vafriables for symbolic debugging. The CDFG of the constructed computation is depicted in Fig. 4. Expected optimization steps are applied to shaded areas in the figure as follows.

1) Additions $A1$ through $A4$ can be compacted in a tree of additions for critical path reduction (throughput optimization).
2) The number of multiplications can be reduced by applying the distributivity rule to multiplications $M1$ and $M2$ and addition $A5$ (area optimization).

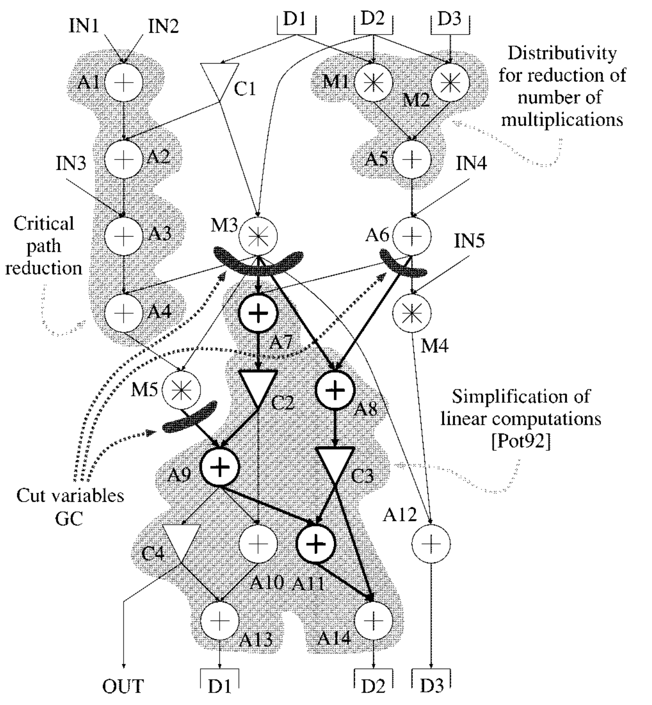

Fig. 4. Example of tradeoffs involved in selection of cut variables such that optimization potential of the computation is not impacted.

3) All operations in the remaining shaded area can be optimized according to the transformations for achieving fast linear computation [18].

We define an important concept that enables effective DfD. A *golden cut* is defined as a set of variables in the source code, which should be *correct* [7] in the optimized program. A *complete golden cut* $V_{\text{cut}}$ 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. Alternatively, a *complete golden cut* is a set of variables which bisects all cyclic paths in the CDFG of a computation [10].

Variables of an example complete golden cut *GC* (output of $M3$, $M5$, and $A6$) are depicted in Fig. 4. 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 Fig. 4 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 above mentioned optimizations, their selection yields efficient symbolic debugging accompanied with effective design implementation.

Conversely, a highly inefficient cut can be constructed using the output variables of $M1$, $M2$, $A2$, $A9$, $C2$, and $A8$. Besides larger cardinality of the involved set 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 preprocessing DfD phase. Therefore, we propose in this paper a cut selection process, which is guided using heuristics that determine the likelihood that an operation can be involved in a transformation.

## II. Related Work

We survey the related works along two lines: computer-adided design (CAD) for debugging and symbolic debugging of optimized code. In the CAD domain, recently Powley and De Groat [19] developed a very high-speed integrated-circuit hardware description language (VHDL) model for an embedded controller . The model supports debugging of the application software. Koch *et al.* [12] 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 [16]. Hennessy [7] introduced the problem of debugging optimized code, defined the basic terms, and presented measurements of the effects of some local optimizations. DOC [4] and CXdb [2] are two examples of real debuggers for optimized code that do not deal with global optimizations. Adl-Tabatabai and Gross [1] 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. CAD-related debugging efforts include [10], [11], [13].

## III. Computational and Hardware Model

We represent a computation by a hierarchical CDFG consisting of nodes representing data operators or subgraphs and edges representing the data, control, and timing precedence [20]. The computations operate on periodic semiinfinite streams of inputs to produce semiinfinite streams of outputs. The underlying computational model is homogeneous synchronous data flow model [14], which is widely used in computationallly 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 register-transfer 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 that enables controllability in the “debugging” mode. The following input operation is incorporated to provide complete controllability of a variable Var1 using user specified input: Input1: if(Debug) then $\mathrm{Var1} = \mathrm{Input1}$.

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 that 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.

## IV. Design for Symbolic Debugging

In this section, we present two symbolic debugging algorithms. The first, presented is Section IV-A, targets fast variable computation. The motivation for this approach is that embedded hardware–software often have infinite-stream semantics (e.g., synchronous data-flow) and, therefore, require long simulation times. In this situation, it is important to continuously calculate values of the user specified variables in the initial specification. This calculation is conducted most often on personal workstations which has relatively limited computationall resources compared to the dedicated embedded hardware.

The second debugging approach, presented in Section IV-B, aims to enable symbolic debugging while preserving all the potential of the computation to be optimized using transformations. Transformations are often the best way to optimize embedded hardware–software. Therefore, the two approaches are independent and in some sense orthogonal. They can be combined by forming a composite objective function for both goals. Instead of combining them in arbitrary way without specific design goals, we decided to evaluate their individual effectiveness.

Note that both proposed symbolic debugging techniques provide complete information about all variables in the initial behavioral specification.

### A. Selection of Optimal Golden Cuts for Fast Variable Computation

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 store 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.

Our proposed method requires that the golden cut should be chosen to result in minimum debugging time, optimal design metrics, and 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, 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

```
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 $|c_{E'}(e)| \le k$
        break
    Mark as "green" the edges with $l \le |c_{E'}(e)| \le k$.
    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 $E'$
```

Fig. 5. Pseudocode of the basic heuristic for the golden cut problem. $l$ and $k$ are lower and upper boundaries, respectively, where edge can be selected for CG.

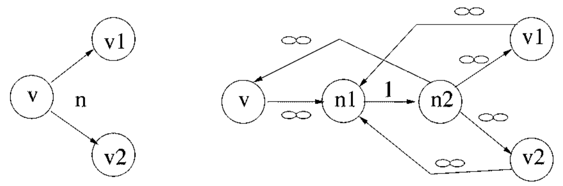

Fig. 6. Modeling a hyperedge in flow network.

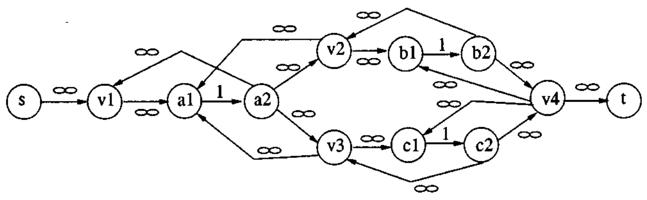

Fig. 7. Construction process of a flow network for the "green" subgraph: the flow network. Original hypergraph consists of four nodes: v1, v2, v3, and v4, where v1 sends data to v2 and v3; v2 and v3 send data to v4.

golden cut should be large enough 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 set 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 set of nodes consisting of nodes on paths from all edges in $E'$ to $e$.

We define hyperedge as collection of edges that go from a node to other nodes. The reason behind this definition is that these edges correspond to the same variable in the computation.

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 Fig. 5. 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 [3]. Using the method proposed by Yang and Wong [21], the flow network for the subgraph is constructed as the following.

1) For each hyperedge $n = (v; v_1, \ldots, v_m)$ 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 Fig. 6).
2) 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.

The construction process for an example graph is shown in Fig. 7. A minimum cut of the constructed flow network can be found using various approaches such as the $O(|V||E|)$-time algorithm in [21]. We use linear programming for solving a flow network by relying on a public domain package `lp_solve` [17]. 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. Trivial solution is the solution where $l$ is set uniformly to zero, i.e., we always take golden cut as far as allowed by the user imposed constraint. In order to avoid greedy and suboptimal solutions, we consider edges in a belt of size $l$ around the user-specified distance for establishing golden cut. Experimental evaluation indicates that $l = 2$ works well for all examples.

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 Fig. 5 as a search engine. The heuristic described in Fig. 5 is modified such that the constant $l$ is not fixed and its value is randomly chosen between one and $k$ for each pipeline stage insertion. Let $\mathrm{Pipeline}(H, k)$ 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 $\mathrm{Pipeline}(H, k)$ is described in Fig. 8.

The DfD algorithm for fast variable computation can be explained at the intuitive level in the following way. The basic idea is to start from the boundary of computation defined by its states and primary inputs. Since we want to calculate all variables in

```
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
```

Fig. 8. Pseudocode of the iterative improvement heuristic for the golden cut problem. Parameter $c$ denotes the number of times of iterative improvement attempts. In our experimentation, we used $c = 10$ because we observed that higher values induce longer runtime with no improvements to the quality of results.

the initial behavioral specification, the next component of cut must be placed on edges (variables) within this range. The trivial and locally optimal solution is to place the cut as far as possible. This is not necessarily a good decision because this could induce a very large cut. Another extreme solution is to place the cut along the min cut of the edges within the range. Since there are numerous polynomial time algorithms for this task, this approach provides provably minimal cut. However, this cut could be very close to the previous cut, with the eventual result that a very large number of edges is placed in the cuts. In order to strike the right balance, we heuristically evaluate several options that consider alternative solutions within the two extremes. We leverage on the min-cut max-flow algorithm to optimally locate minimal cuts when we place a variety of constraints where the cut can be placed within the allowed range. In order to further enhance the performances of the algorithm and avoid local minima, we employ the basic optimization strategy within the iterative improvement paradigm.

One can envision many sophisticated techniques for calculating $l$. For example, $l$ can be taken from local min cut to the furthest user-specified distance for each cut. In practice, however, we observe that simple constant bound $l = 2$ works well for all examples.

## B. Optimization-Friendly Selection of Optimal Golden Cuts

The performance effectiveness of the developed symbolic debugging approach depends strongly on the selection of golden cut variables. In this section, we identify the tradeoffs 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.

*Definition of a Complete Golden Cut:* Every cycle in the CDFG must have at least one vertex in the golden cut. A complete cut is a set of variables which bisects all cyclic paths in the CDFG of a computation.

The definition of a complete golden cut has been adopted from [10], where cuts are used to transfer minimal computation states from simulation to emulation engines. Such a definition of a cut ensures that any variable in the original specification can be computed from its cut. Necessary and sufficient condition that all the variables can be computed in finite number of steps is that the graph does not contain directed cycles. 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 tradeoffs involved with potential optimizations. The developed DfD for minimal impact on computation performance 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 [5]. The set of criteria for optimization-sensitive cut selection is incorporated into the search process using an objective function $\Phi(v_i, \mathrm{CDFG})$. This function attempts to quantify for each variable $v_i$ the likelihood that $v_i$ disappears during the synthesis process

$$\begin{aligned}\Phi(v_i, \mathrm{CDFG}) = \alpha \cdot |\mathrm{GC}| \\ + \mathrm{fanout}(v_i) \\ + \mathrm{testLinear}(v_i) \\ + \mathrm{test\_}\epsilon\mathrm{\_CP}(v_i) \\ + \mathrm{testDistributivity}(v_i) \\ \times \mathrm{testParallelism}(v_i) \\ + \mathrm{testInputsInCycles}(v_i).\end{aligned}$$

The components of the objective function return quantifiers that represent the tradeoffs 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. In our experiments, we have used the meta-algorithmics parameter tuning procedure [9]. Each component corresponds to the following generic optimization objectives.

- $|\mathrm{GC}|$—*Small cardinality of the golden cut:* Any additional constraints imposed on computation will reduce the optimization potential. Therefore, we would like to add as few as possible constraints.
- $\mathrm{fanout}(v_i)$—*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.
- $\mathrm{testLinear}(v_i)$—*Nonlinear operations:* It has been demonstrated that a collection of linear operations (addition, subtraction, multiplication with a constant, etc.) can be transformed optimally and very effectively for a particular design metric [18]. Therefore, operands or results of nonlinear operations should be given preference for inclusion in a golden cut.
- $\mathrm{test\_}\epsilon\mathrm{\_CP}(v_i)$—*$\epsilon$-Critical path:* During transformations for almost all design metrics, the critical path of the computation is frequently severely modified. This reasoning stems from the fact that the critical path usually limits performance of a circuit. Therefore, the golden cut selection

routine should avoid including variables at most $\epsilon$ operations close to the critical path. In our experiments, we used $\epsilon = 20\%$.

The following optimization indicator has been considered for area minimization.

- $\text{testDistributivity}(v_i)$—*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, such as multiplications or divisions, it is of utmost importance not to disable this transformation.

In our experiments, we have considered only one transformation for maximizing throughput.

- $\text{testInputsInCycles}(v_i)$—*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.

The problem of finding a complete golden cut that obeys the requirements of all optimization goals can be defined formally using the following standard format.

**PROBLEM: The Complete Golden Cut.**

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

**QUESTION:** *Is there a set of variables GC such that when removed from the CDFG, leaves no directed cycles and the sum of weights* $\sum_{v_i \in GC} \Phi(v_i, \text{CDFG})$ *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 [6]. The developed heuristic algorithm for this problem is summarized using the pseudocode in Fig. 9. The heuristic starts by logically partitioning the graph into a set of strongly connected components (SCCs) using the breadth-search algorithm [3]. This algorithm has complexity $O(V + E)$, where $V$ is the number of vertices and $E$ is the number of edges in a graph. All trivial SCCs that 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 nontrivial SCCs.

At the beginning of each iteration, to reduce the solution search space, a graph compaction step is performed. In this step, each path $P : A \rightsquigarrow B$ that 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 (variable) 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

```
Create a set $SCC = ComputeScc(CDFG(V,E))$ of strongly
connected components [3]
For each $SCC_i \in SCC$
  If $|SCC_i| = 1$ delete $SCC_i$ from $SCC$

CUT = null
While $SCC \neq empty$
  For each $SCC_i \in SCC$
    $GraphCompaction(SCC_i)$
    For each node $v_j \in SCC_i$
      $S = ComputeScc(SCC_i - v_j)$
      $OF(S, v_j) = \sum_{i=1}^{S} \Phi(v_j, CDFG)$
    End For
    Select vertex $v_j$ which results in max $OF(S, v_j)$
    Delete $v_j$ from $SCC_i$
    $SCC+ = S(V_j)$
    For each $SCC_i \in SCC$
      If $|SCC_i| = 1$ delete $SCC_i$ from $SCC$
    End For
    $CUT = CUT \cup v_j$
  End For
End while

Procedure $GraphCompaction(SCC_i)$
  For each vertex $v_j \in SCC_i$
    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 $E_{j,m}$
        Delete $E_{k,m}$
      End For
      $Weight(v_j) = max(Weight(v_j), Weight(v_k))$
      Delete $v_k$
  End For
```

Fig. 9. Pseudocode for the developed algorithm for the complete golden cut problem.

function is removed from the set of nodes as well as all adjacent edges. The deleted vertex is added to the resulting cutset. The process of graph compaction, evaluation of node deletion, node deletion, and graph updating is repeated until the set of nontrivial SCCs in the graph is empty. The set of nodes (variables) deleted from the computation represents the final cutset selection.

Consider the example shown in Fig. 10. The CDFG of the third-order Gray–Markel ladder infinite-impulse response (IIR) filter, shown in Fig. 10(a), has only one nontrivial SCC. The graph compaction step is explained in Fig. 10(b), where vertex $B$ is merged with vertex $A$ as well as variable $W$ is merged with variable $V$. In Fig. 10(c) an example of node deletion is described. The deleted node creates two smaller SCCs.

*1) Discussion of Cut Validity After Applying Transformations:* Once the DfD for minimal impact on computational performance procedure modifies the source code $\text{cdfg}_m = \text{DfD}(\text{cdfg})$, a synthesis tool *ST* is applied in order to generate the final optimized specification $\text{cdfg}_o = \text{ST}(\text{cdfg}_m)$. 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* that translates the source specification $\text{cdfg}_m$ with an enforced complete golden cut *GC* into a new specification $\text{cdfg}_o$, where the enforced cut *GC* is not a complete cut? This question can be answered from two perspectives.

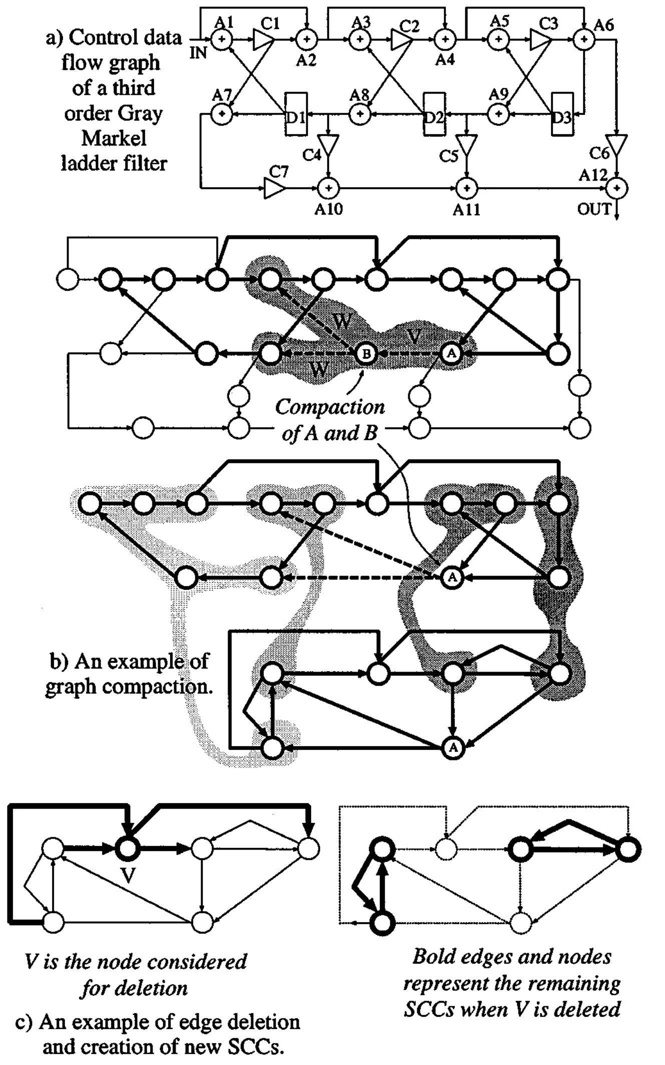
a) Control data flow graph of a third order Gray Markel ladder filter
IN
A1
C1
A2
A3
C2
A4
A5
C3
A6
A7
D1
A8
D2
A9
D3
C4
C5
C6
C7
A10
A11
A12
OUT
W
V
B
A
Compaction of A and B
b) An example of graph compaction.
V
V is the node considered for deletion
Bold edges and nodes represent the remaining SCCs when V is deleted
c) An example of edge deletion and creation of new SCCs.

Fig. 10. Performing the steps of a single iteration of the cutset selection procedure.

1) Several examples of computation structures of different implementations of the same computational functionality (for example: the Gray–Markel ladder, cascade, parallel, elliptic, and direct-form IIR filters) clearly indicate that, generally, 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.
2) There exist transformations performed by common compilers (such as loop fusion, splitting, 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.

## V. Experimental Results

This section summarizes our evaluation of two proposed DfD techniques. Section V-A provides information about the effectiveness of symbolic debugging technique for fast variable calculation. Section V-B presents our findings about the impact of symbolic debugging technique for minimal impact on computation performance.

TABLE I
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, Respectively, of Initial Query Time

| 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 |

### *A. Symbolic Debugging with Fast Variable Computation*

We applied our approach to design for symbolic debugging on a set of ten small industrial examples as well as two large design examples. The smaller designs include a set of Avenhaus, Volterra, and IIR filters, an audio digital-to-analog converter, and a least mean square audio formatter. Table I 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 I 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 kb/s. Table II 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.

### *B. Symbolic Debugging with Minimal Impact on Computation Performance*

Proper performance evaluation of the proposed debugging techniques is a complex problem due to a great variety of optimization steps that can be undertaken during design optimization using transformations. In addition, it is well known that the effectiveness of transformations is greatly dependent or the order in which they are applied. The situation is further complicated by a great variety of designs. For example, design can vary tremendously in their size, type of used operations, cycle, and in general topology structure.

TABLE II
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, RESPECTIVELY, OF INITIAL QUERY TIME

| 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 |

In order to address this concern, we have applied the new technique on more than hundred designs from the Hyper [8] and Mediabench [15] 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. In addition, we have used several popular scripts for transformation ordering, such as one which guarantees the maximal throughput when applied to linear computations. *On the overwhelming number of designs, our technique did not incur any cost, regardless of targeted optimization goals: area, throughput, or power*.

On two designs (noise-sharper and modem), the DfD procedure induced rather high overhead. This is mainly a consequence of the fact that the designs are very amenable for very aggressive optimization. Addition of any debugging constrains reduces this potential. Furthermore, on these twjo designs, heuristic scheduling and assignment techniques performed really well on the initial specification, but not so well on the modified representation. The Hyper high-level synthesis system [20] uses randomized scheduling and allocation algorithms; when applied on large numbers of design, some statistical outflier effects are inevitable.

The detected exceptions are shown in Table III. 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 these examples indicates that the symbolic constraints induced a need for computation of additional variables used only for debugging purposes. The used combination of transformations drastically changed the structure of computations such that the initial selection of cut resulted in a need for significant additional computation. 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 and they are commonly associated with application of rather complex and sophisticated transformation scripts for optimization of complex objective functions. Such design objectives are desired rarely in modern design practice.

Both symbolic debugging techniques are faster on average than the Hyper scheduling and assignment procedures. While DfD for fast calculation has just somewhat lower runtime than the Hyper synthesis procedures, the second technique is almost always at least an order of magnitude faster. We conducted extensive experimentation on the weight factors used for the objective function in DfD procedure for aggressive optimization. We varied the weight factors for each parameter by scaling each of them by random factors in range 0.1 to 10. The procedure had very consistent performances except for two cases. These two cases occur when the cardinality of golden cut and test linear are assigned small weights. Small weight factors for these two component often seriously reduce the effectiveness of transformations. On the other hand, setting very high values for these two components does not have a significant impact on the overall effectiveness.

TABLE III
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—IS THE OVERHEAD IN AREA INCURRED DUE TO PREPROCESSING FOR SYMBOLIC DEBUGGING

| 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% |
| 7IIR | 10 | 4 | 7 | 42.58 | 51.95 | 22% |
| avenhaus | 11 | 5 | 4 | 49.90 | 60.38 | 21% |
| 10IIR | 12 | 5 | 5 | 55.42 | 68.15 | 23% |
| 11IIR | 17 | 5 | 5 | 66.26 | 75.53 | 14% |
| band-pass | 20 | 5 | 6 | 172.45 | 214 | 24% |
| noise-shaper | 29 | 6 | 9 | 233.97 | 325.7 | 39% |
| modem | 25 | 6 | 11 | 238.01 | 330.3 | 40% |
| DAC | 58 | 3 | 3 | 42.99 | 43.09 | 0.2% |

Fundamentally, a sound way to analyze the experimental results when numerous alternatives are available is to statistically sample the design space for variety of design and transformation orders. While we conducted numerous experiments, we did not try to analyze them statistically because the performances of both DfD techniques was very consistent. In addition, except in a few designs that are shown in Table III, we did not observe significant overhead.

## VI. CONCLUSION

We addressed the problem related to the retrieval of source values for the globally optimized behavioral specifications. 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.

## REFERENCES

[1] A.-R. Adl-Tabatabai and T. Gross, "Source-level debugging of scalar optimized code," *SIGPLAN Not.*, vol. 31, no. 5, pp. 33–43, May 1996.
[2] G. Brooks, G. J. Hansen, and S. Simmons, "A new approach to debugging optimized code," *SIGPLAN Not.*, vol. 27, no. 7, pp. 1–11, July 1992.
[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1990.

[4] D. S. Coutant, S. Meloy, and M. Ruscetta, “DOC: A practical approach to source-level debugging of globally optimized code,” *SIGPLAN Not.*, vol. 23, no. 7, pp. 125–134, July 1988.

[5] S. Dey, A. Raghunathan, N. K. Jha, and K. Wakabayashi, “Controller-based power management for control-flow intensive designs,” *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 1496–1508, Oct. 1999.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979, p. 192.

[7] J. Hennessy, “Symbolic debugging of optimized code,” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 323–344, July 1982.

[8] A. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen, “Optimizing power using transformations,” *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 12–31, Jan. 1995.

[9] D. Kirovski and M. Potkonjak, “System-level synthesis of low-power hard real-time systems,” in *Proc. Design Automation Conf.*, June 1997, pp. 697–702.

[10] D. Kirovski, M. Potkonjak, and L. M. Guerra, “Improving the observability and controllability of datapaths for emulation-based debugging,” *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 1529–1541, Nov. 1999.

[11] ——, “Cut-based debugging for programmable systems-on-chip,” *IEEE Trans. VLSI Syst.*, vol. 8, pp. 40–51, Feb. 2000.

[12] G. Koch, U. Kebschull, and W. Rosenstiel, “Debugging of behavioral VHDL specifications by source-level emulation,” in *Proc. Eur. Design Automation Conf.*, Sept. 1995, pp. 256–261.

[13] F. Koushanfar, D. Kirovski, and M. Potkonjak, “Symbolic debugging scheme for optimized hardware and software,” in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 2000, pp. 40–44.

[14] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proc. IEEE*, vol. 75, pp. 1235–1245, Sept. 1987.

[15] C. Lee, M. Potkonjak, and W. H. Mangion-Smith, “MediaBench: A tool for evaluating and synthesizing multimedia and communication systems,” in *Proc. Int. Symp. Microarchitecture*, 1997, pp. 330–335.

[16] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, and A. Jerraya, “System-on-a-chip cosimulation and compilation,” *IEEE Des. Test Comput.*, vol. 14, pp. 16–25, Apr.–June 1997.

[17] M. R. C. M. Berkelaar. LP Solve User’s Manual, version 1.5. Eindhoven Univ. Technol. [Online]. Available: ftp://ftp.es.ele.tue.nl/pub/lp_solve

[18] M. Potkonjak and J. Rabaey, “Maximally and arbitrarily fast implementation of linear and feedback linear computations,” *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 30–43, Jan. 2000.

[19] G. S. Powley and J. E. DeGroat, “Experiences in testing and debugging the i960 MX VHDL model,” in *Proc. VHDL Int. Users Forum*, May 1994, pp. 130–135.

[20] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, “Fast prototyping of datapath-intensive architectures,” *IEEE Des. Test Comput.*, vol. 8, pp. 40–51, June 1991.

[21] H. Yang and D. F. Wong, “Efficient network flow based min cut balanced partitioning,” in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 1994, pp. 50–55.

[22] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein, “System design methodology of UltraSPARC-I,” in *Proc. Design Automation Conf.*, June 1995, pp. 7–12.

[23] A. Yu, “The future of microprocessors,” *IEEE Micro*, vol. 16, pp. 46–53, Dec. 1996.

**Farinaz Koushanfar** received the B.S. degree in electrical engineering from Sharif University of Technology, Tehran, Iran, in 1998 and the M.S. degree in electrical engineering from the University of California, Los Angeles, in 2001. She is currently working toward the Ph.D. degree in electrical engineering and computer science at the University of California, Berkeley.

Her current research interests include design and debugging of embedded systems, wireless distributed embedded systems, and, in particular, wireless ad hoc sensor networks, sensor network optimizations, and computational security.

Ms. Koushanfar is a recipient of the National Science Foundation Graduate Research Fellowship.

**Darko Kirovski** received the M.S. and Ph.D. degrees from the University of California, Los Angeles, in 1997 and 2000, respectively.

He is currently a Researcher with Microsoft Research, Redmond, WA. His current research interests include various aspects of design and debugging of system-on-chip, intellectual property protection, and content protect systems.

Dr. Kirovski received a Microsoft Research Graduate Fellowship and a DAC Graduate Fellowship.

**Inki Hong** received the M.S. degree in computer science from Stanford University, Stanford, CA, in 1994, and the Ph.D. degree in computer science from the University of California, Los Angeles, in 2001.

He is currently a Senior Research and Design Engineer with the Physical Synthesis Group at Synopsys, Inc., Mountain View, CA. He has authored or coauthored more than 20 papers. His current research interests include system-level synthesis, physical synthesis, intellectual property protection, and verification.

**Miodrag Potkonjak** (S’90–M’91) received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1991.

In 1991, he joined C&C Research Laboratories, NEC USA, Princeton, NJ. Since 1995, he has been with the University of California, Los Angeles, where he is currently a Professor with the Computer Science Department. His current research interests include communication systems design, embedded systems, computational security, and intellectual property protection.

Prof.. Potkonjak received the National Science Foundation CAREER Award, the OKAWA Foundation Award, the University of California at Los Angeles TRW SEAS Excellence in Teaching Award, and a number of best paper awards.

**Marios C. Papaefthymiou** (M’91) received the Ph.D. degree in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA, in 1993.

He is an Associate Professor of Electrical Engineering and Computer Science and Acting Director of the Advanced Computer Architecture Laboratory at the University of Michigan, Ann Arbor. His current research interests include several aspects of very large scale integration systems design with an emphasis on timing and energy-related problems. He is also active in the field of parallel and distributed computing.

Prof. Papaefthymiou received a Best Paper Award at the 1995 ACM/IEEE Design Automation Conference, an ARO Young Investigator Award, an National Science Foundation CAREER Award, and several IBM Partnership Awards. He is an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and for the IEEE TRANSACTIONS ON COMPUTERS.