Improving the Observability and Controllability of Datapaths for Emulation-Based Debugging

Darko Kirovski, Miodrag Potkonjak, and Lisa M. Guerra

***Abstract*— Growing design complexity has made functional debugging of application-specific integrated circuits crucial to their development. Two widely used debugging techniques are simulation and emulation. Design simulation provides good controllability and observability of the variables in a design, but is two to ten orders of magnitude slower than the fabricated design. Design emulation and fabrication provide high execution speed, but significantly restrict design observability and controllability.**

To facilitate debugging, and in particular error diagnosis, we introduce a novel cut-based functional debugging paradigm that leverages the advantages of both emulation and simulation. The approach enables the user to run long test sequences in emulation, and upon error detection, roll-back to an arbitrary instance in execution time, and transparently switch over to simulation-based debugging for full design visibility and controllability. The new debugging approach introduces several optimization problems. We formulate the optimization tasks, establish their complexity, and develop most-constrained least-constraining heuristics to solve them. The effectiveness of the new approach and accompanying algorithms is demonstrated on a set of benchmark designs where combined emulation and simulation is enabled with low hardware overhead.

***Index Terms*— Application-specific integrated circuit (ASIC) debugging, emulation, fast prototyping, maximum feedback set, simulation.**

I. Introduction

THE KEY technological and application trends, mainly related to increasingly reduced design observability and controllability, indicate that the cost and time expenses of debugging follow sharply ascending trajectories. Two most directly related factors are rapid growth in the number of transistors per pin and increased level of hardware sharing. The analysis of physical data for state-of-the-art microprocessors (according to *the Microprocessor Report*) indicates that in less than two years (from late 1994 to mid 1996) the number of transistors per pin increased by more than a factor of two, from slightly more than 7000 to 14 100 transistors per pin. At the same time, the size of an average embedded or digital signal processing (DSP) application has been approximately doubling each year, the time to market has been getting shorter for each new product generation, and there has been a strong market need for user customization of application-specific systems. Together, these factors have resulted in shorter available debugging time for increasingly complex designs. Finally, design and computer-aided design (CAD) trends that additionally emphasize the importance of debugging include design reuse, introduction of system software layer, and increased importance of collaborative design. These factors result in increasingly intricate functional errors, often due to interaction of parts of designs written by several designers.

Such technology and design trends indicate that functional verification emerges as a dominant step with respect to time and cost in the development process. The difficulty of verifying designs is likely to worsen in the future. 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 [20]. Traditional approaches, such as design emulation and simulation, are becoming increasingly inefficient to address system debugging needs. Design emulation—implemented on arrays of rapidly prototyping modules [field programmable gate arrays (FPGA's)] or specialized hardware—is fast, but due to strict pin limitations, provides limited and cumbersome design controllability and observability. Simulation—software model of the design at an arbitrary level of accuracy—has the required controllability and observability, but is, depending on the modeling accuracy, two to ten orders of magnitude slower than emulation [18], [21].

The novel ideas proposed in this work advocate the development of a new paradigm for debugging and design-for-debugging of application-specific integrated circuits (ASIC's). The new debugging technique integrates design emulation and simulation, in a way that the advantages of the two are combined, while the disadvantages are eliminated.

The functional debugging process, depicted in Fig. 1, includes four standard debugging procedures: test input generation and execution, error detection, error diagnosis, and error correction. Long test sequences are run in emulation. Upon error detection, the computation is migrated to the simulation tool for full design visibility and controllability. To explain how execution is transferred from one execution domain to another, we introduce the notion of a *complete cut*. A complete cut is a subset of variables that fully determines the design state at an arbitrary time instance. The ability to read/write the state of a particular cut from/to the design is enabled by inserting register-to-port interconnects and appropriate scheduling statements into the initial design specification. The design techniques developed to enable migration of the execution are applied as a design post-processing step and, thus, can be used in conjunction with existing or future synthesis systems or manual design approaches.

Manuscript received July 30, 1998; revised May 7, 1999. This paper was recommended by Associate Editor R. Camposano.

D. Kirovski and M. Potkonjak are with the Computer Science Department, University of California, Los Angeles, CA 90095 USA (e-mail: darko@cs.ucla.edu).

L. M. Guerra is with Conexant Systems Inc., Newport Beach, CA 92658-8902 USA.

Publisher Item Identifier S 0278-0070(99)09467-1.

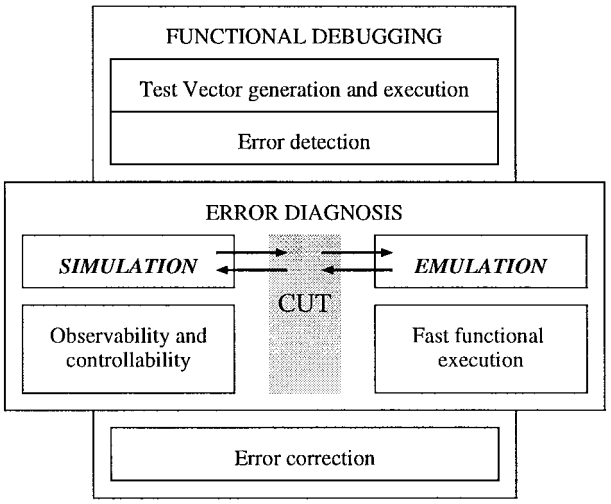

Fig. 1. The new concept of functional debugging. The running design periodically outputs the cut state, which is stored in a database. Any one of these states can be used to initialize, and then continue execution with preserved functional and timing accuracy.

The running design (simulation or emulation) periodically outputs the cut state. These states are saved by a monitoring workstation. When a transition to the alternate domain is desired, any one of the previously saved states can be used to initialize, and then continue execution in simulation or emulation with preserved functional and timing accuracy. Once the error is localized and characterized in the error diagnosis step, the emulator is updated or built-in fault tolerance mechanisms are activated.

The new debugging approach introduces a number of optimization problems involved in the design-for-debugging post-processing phase. The developed set of optimization methods aims to add minimum hardware overhead and still provide efficient integration of the two functional testing domains. The applied algorithms are constructed using the most-constrained least-constraining heuristic methodology. The efficiency of the developed algorithms is tested on a set of real-life examples, where combined simulation and emulation debugging is provided with exceptionally low implementation overhead.

The diagnosis approach and accompanying optimization issues are illustrated using a fifth-order continued fraction infinite impulse response (IIR) filter. Fig. 2(a) shows the control data flow graph for this filter. In Fig. 2(b), the assignment and scheduling of the same computation structure is depicted for an architecture that consists of one multiplier, one adder, and six registers.

The goal of the design-for-debugging step is to allocate minimal hardware resources that enable the cut state to be observed and controlled. The primary requirement of this design post-processing for debugging is to avoid changing the existing design allocation, assignment, and scheduling. In order to avoid addition of new input–output (I/O) ports, the cut should be scheduled for transfer at control steps when the states of input and output variables are not imported/exported (in this example, control steps 1–11). For controllability, during loading of the cut state, the value of each variable in the cut should be written before its first usage. For observability, during export of the cut, the state of each variable in the cut should be read before the value of the same variable for the next computation iteration or another variable will overwrite it. An integral part of any complete cut is the primary input and output of the system.

A trivial candidate for a subset of variables, which constitutes a computation complete cut, is $C = \{D1, D2, D3, D4, D5\}$ [dotted lines in Fig. 2(b)]. The state of the cut and the input completely defines the state of a particular iteration of the depicted computation. Hence, a particular cut state along with a correspondingly synchronized input sequence can be used to restart the computation correctly on an execution engine. A possible set of control steps at which the cut state can be input is $CS = \{1, 2, 3, 4, 5\}$. Since all variables in C are concurrently alive, they must be stored in five different registers. The designer requires access to read/write into these registers from the designated I/O pins. Since the cut is stored in five registers, five register-to-I/O connections have to be allocated to enable cut observability and controllability.

As a lower overhead alternative, consider the cut consisting of the output variables of additions $A2$, $A4$, $A6$, $A8$, and $A10$ [bold lines in Fig. 2(b)]. Only one register ($R5$) is required to hold the values of these variables since they are not alive simultaneously. In this case only one register-to-port connection is dedicated to the register that holds the cut. Cut dispensing is performed in five consecutive control steps: 2–6.

II. Background

State-of-the-art tools for system debugging have primarily concentrated on enhancing the performance of software simulation models and the design visibility in emulation, rather than trying to provide methods for their synergy. VHDL or Verilog register transfer (RT)-level simulation environments are capable of performing error tracing and timing analysis[1] and simulation backtracking[2].

Hardware emulators have been developed as early as 1979 [3], and have been under further development ever since [15]. Modern reconfigurable systems for physical emulation of electronic circuits include a data entry workstation where a user may input data representing the circuit configuration. This data is converted to a form suitable for programming an array of programmable gate elements provided with a richly interconnected architecture.[3]

The observability and controllability of variables in such systems is a great challenge for emulator developers. The developed approaches are inefficient, expensive, or both. A common approach uses the expensive, low-bandwidth, and intrusive Joint Test Action Group (JTAG) boundary scan methodology [14]. The most advanced application of JTAG circuitry has been introduced in the industry's first solution for run-time target application-host data exchange (RTDX) by Texas Instruments [19]. Software developers use C or DSP assembly code to address an internal data exchange library,

[1] http://www.interrainc.com/htmls/ver2/picasso.html.

[2] http://www.synopsys.com/products/simulation/simulation.html.

[3] FPGA-based emulation systems have been developed by a number of companies. including: Quickturn, http://www.quickturn.com/products/cobalt.htm; Ikos, http://www.ikos.com; and Axis http://www.axiscorp.com/.

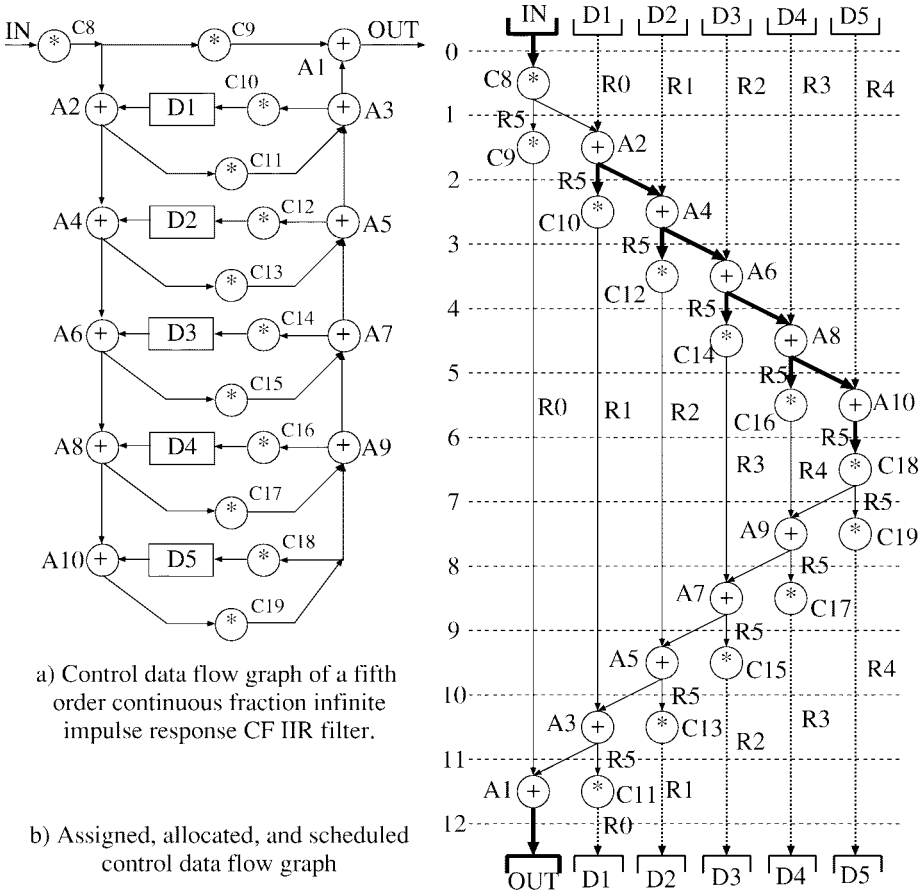

Fig. 2. Optimal cut example: (a) control data flow graph (CDFG) and (b) allocated, assigned, and scheduled CDFG for the fifth-order CF IIR filter. (b) Depicts two cuts: $C1 = \{IN, D1, D2, D3, D4, D5\}$ with dotted edges and $C2 = \{IN, A2, A4, A6, A8, A10\}$ with bold edges.

which in turn makes use of a scan-based emulator to move data on and off chip via the JTAG serial test bus.

Design controllability and observability can be obtained also by addressing user-customized SRAM memory cells (Quickturn Cobalt) or by probing nets into the FPGA testbed (Quickturn System Realizer, Mentor Graphics SimExpress[4]). While the former case raises expenses, the latter reduces visibility performance and continuity (e.g., 6048 probes are available in Synopsys SimExpress; 1024 selected signals over 4 million cycles can be stored in internal buffers of Mentor Graphics Celaro).

Novel challenges in system debugging are streamlined toward partitioning the system execution in software and hardware (Quickturn Q/Bridge and Axis Corporation), verification of emulation hardware with respect to the targeted functionality and timing [12], and signal reconstruction for increased visibility and reduced emulation bandwidth demands [13].

A. Computation and Hardware Model

Two main, often contradictory, criteria for evaluation of system and behavioral synthesis models of computations are expressiveness [6] and suitability for optimization. While high expressiveness implies wider application domain, suitability for optimization often implies efficient implementation. For the sake of conceptual simplicity, in this work, we target the synchronous data flow (SDF) model of computation [2]. This computation model is often used to facilitate optimization-intensive compilation for ASIC platforms (filtering, frequency transforms, wavelet computation structures, error-correction coding, encryption, etc.) [17]. Modern single-chip applications (for example, MPEG audio/video encoding/decoding, or wireless communication protocols and data transfer) are by default not developed based only on the SDF computation model. However, most of the subfunctions [discrete cosine transfer (DCT) and fast Fourier transform (FFT)] transforms, Huffman coding, etc.) in such applications can be modeled using the SDF computation model. The computation, that does not follow the SDF model, can be abstracted using the semi-infinite stream random-access machine (SISRAM) model. The SISRAM model is created by removing a requirement for algorithm termination from the standard RAM model [1].

It is important to stress that the cut-based debugging approach is not limited to a specific computation model. However, for each computation model, a cut definition has to be established to satisfy the generic concept of a cut: a cut at time T is defined as a subset of variables from which any other variable computed after T can be computed. A cut definition for the SISRAM computation model has been presented elsewhere [11]. In this manuscript, we describe cut selection for ASIC's that are synthesized using CDFG's—defined in Section IV-A). This simplification is assumed because of three reasons: brevity, availability of synthesis tools, and the

[4]http://www.mentorg.com/codesign/main-f/index.htm.

fact that the SDF computation model corresponds to many data-intensive multimedia, communications, and wireless applications. In our experiments, we used Silage [17] as a specification language for the ASIC implementation.

We assume fully deterministic behavior of hardware and a continuous semi-infinite operation mode (not necessarily periodic). 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 into register files. Each hardware resource can be connected in an arbitrary way to other hardware resources. We do not impose any restrictions on the number of pipeline stages of the employed functional units.

The design is fully specified and its functionality and realization is not disturbed by the debugging process, with the exception of enabling the user to write into specific controllable registers. In a fully specified design each operation, variable, and data transfer is scheduled and assigned to a particular instance of hardware resource in one or more control steps. In order to support debugging, we allocate additional debugging hardware to satisfy all debugging requirements. The goal is, of course, to add as little hardware as possible. In particular, we do not allow increase in the number of I/O pins, since this is a constraint that dominates other hardware constraints in modern designs.

III. The New Approach—Cut-Based Integrated Debugging

The key idea behind the cut-based approach for the integration of simulation and emulation is to leverage on the strong aspects of each of the functional execution domains. The resulting debugging technique provides fast, observable, and controllable functional execution. The essence of the idea is the establishment of the concept of a complete cut. A complete cut of a computation represents a subset of variables sufficient to correctly continue the computation regardless of the values of variables that are not part of the cut, i.e., all variables in the computation that are not part of the cut can be recomputed using only the cut variables. The full system state is a straightforward example of a state of a complete cut. However, a complete cut may be substantially smaller than the system state. Clearly, if one has complete controllability and observability over the state of all variables in the complete cut for a specific breakpoint, the computation can be continued functionally correctly from that breakpoint. A cut contains the complete information about the history of the computation process and its primary inputs until a given point in time (breakpoint). For the sake of brevity, from now on when we say cut, we mean a complete cut.

Our cut-based functional debugging approach is conducted using the following three phases of the design and debugging process.

- Design post-processing:
  a) *Phase 1:* Defining the cut from the RT-level design specification.
  b) *Phase 2:* Augmentation of the design specification with cut statements which support controllability and observability when the design is executed in a debug mode.
- Debugging process:
  a) *Phase 3:* Simultaneous and coordinated design execution of the fabricated or emulated design and the appropriate simulator for efficient debugging.

In the first phase, a computation iteration at the behavioral-level of specification is logically partitioned into two or more components such that the cut between the partitions is complete. The synthesis support for exchange of information between simulation and emulation has the following three degrees of design freedom.

- the determination of variables which form the cut;
- the determination of the exact control step when the state of a particular variable is read or replaced by a user specified state;
- the assignment of specific sets of I/O pins used to transfer variable states to or from the chip.

It is important to notice the optimization tradeoff involved in finding the optimal cut. The optimization procedure has a unique goal: to add minimum hardware resources into the initial design, while obtaining its full controllability and observability. A cut with a minimum number of variables seems to be an attractive solution. If those variables are simultaneously alive, more registers that hold the variables of the cut-set have to be provided with register-to-I/O pin interconnects. Therefore, a favorable cut is the one which consists of variables with long disjoint life-times and has the property that a small number of machine registers contain the cut variables.

Once an optimal cut is found, the next design problem is to define the sequence of control steps in which the variables are dispensed out of the chip. The freedom of transferring the cut state is limited due to the control steps when the I/O pins are busy. Due to lack of idle cycles, in some cases, all variables of the cut cannot be transferred over the I/O pins of the emulator. One straightforward solution to this problem is to allocate near-minimal buffer to hold the unscheduled variables. Potkonjak, Dey, and Wakabayashi have utilized the concepts of pipelining debugging variables for improving their scheduling and assignment freedom and use of I/O buffers for improving resource utilization of I/O pins [16]. They do not search for a complete cut of a computation. Instead, they derive provably optimal bounds for the maximum cardinality of the set of controllable and observable variables for a given design specification. Most importantly, they have developed a nongreedy heuristic minimization algorithm for I/O buffer allocation, which can be successfully used in the cut-based debugging framework.

In the second phase of the synthesis approach, the original design specification is augmented with additional resources that enable design observability and controllability. For example, the following input operation is incorporated to provide complete controllability of variable Var_1 using user specified input variable Input_1: **if (DEBUG) then** $\mathrm{Var}_1 = \mathrm{Input}_1$ in the case of pipelined functional units, their pipeline latches are not subject to inclusion into cuts. However, for programmable

platforms with states inaccessible by instructions (pipeline, branch predictors) and memory hierarchies, the problem of outputting the machine cut state becomes a time-lengthy process and is addressed in [11]. The hardware/software codesign platform presented in [11] encapsulates a complex environment with a set of programmable cores, a number of ASIC accelerators, and a memory hierarchy.

IV. Synthesis for Debugging

In this section, we overview the key optimization problems involved in integrating debugging resources into a design specification for full controllability and observability. We present a set of techniques that add minimal hardware resources to a given design specification in order to achieve the design-for-debugging objectives. The determination and integration of inserted debugging resources is performed by the following sequence of tasks.

First, the optimal cut is selected based on the analysis of the computation control data flow graph (CDFG). The goal is to identify a subset of variables that represent a CDFG cut, such that all variables in the set are stored in minimal number of registers. In addition, the computation graph and timing bounds have to allow all variables in the cut-set to be output from the chip through a designated set of I/O pins within a single or multiple computation iterations. This task is explained in detail in Section IV-B. Section IV-C describes the algorithm that searches for an optimal scheduling of cut-set variables with respect to control steps at which a subset of available I/O ports is idle. Finally, the algorithm presented in Section IV-D finds the minimal cardinality set of register-to-port interconnects that enables scheduling the cut-set variables to available ports. After cut-set variables are assigned and scheduled, the initial specification is updated with the set of resources that enable cut-set I/O. The chip is then ready to be fabricated or emulated.

A. Background Definitions

Before we present the formal description of the encountered problems and developed algorithms, we introduce a set of definitions that build the formal foundation for our debugging methodology. A **CDFG** of a computation iteration $\mathbf{i}$ is a directed graph $G(N, PI, P\text{OUT}, D, E)$ with four types of vertices: data operations N, primary inputs PI, primary outputs POUT, and state delays D; and data precedence edges E. Each *data precedence edge* has a single *source* and a single *sink* vertex. *Primary inputs* can be used only as sources to edges. A *primary output* can be used only as a sink of one edge. Each *data operation* N_i has at least one incoming and at least one outgoing data precedence edge. All edges with a common source N_i represent the *variable* V_i generated using data operation N_i. Each operation $N_i \in N$ is *labeled* with an integer L_i that specifies the number of control steps required to execute operation N_i. *State delays* are used to distinguish the computation state between two consecutive iterations. Each state delay D_i can be a sink to only one edge.

The assumed CDFG definition can be easily extended with the following two types of control edges: 1) weighted edges can represent *control precedence* information (for example, if two operations are connected with a control precedence edge weighted W, then the execution of the source operation trails the execution of the sink operation for W control steps) [17] and 2) control edges can be used to create *loop* and *if-then-else* macro constructs as presented in [5]. Since the design-for-debugging process is performed after the RT-level synthesis and, thus, after the operation scheduling, the control edges of type 1) do not impose constraints on the presented debugging methodology. The selected cuts partition all trajectories in the control flow induced by edges of type 2).

According to the allocated resources and data and control dependencies, during the behavioral design process, the CDFG is scheduled and assigned to the allocated hardware resources such that it can be executed in a particular number of control steps. The lower bound in the number of control steps required for execution on $|N|$ functional units is equivalent to the critical path of the CDFG, i.e., largest sum of operation labels along a path from a state delay in computation iteration i to a state delay in the next successive computation iteration $i+1$.

Definition 1: A schedule of variable V_i in a CDFG is determined by the control step C_i^{start} when V_i is created and the control steps C_i^{first}, C_i^{last} when V_i is used for the first and last times, respectively.

Definition 2: A register assignment of variable V_i in a CDFG is a m-to-1 mapping $V_i \rightarrow R_j$ to a register R_j from the set $R_j \in R$ of all registers in the ASIC.

Definition 3: Read life-time of variable V_i stored in register R begins at the control step when variable V_i is created until the control step when variable V_i is overwritten by another variable V_j or the next iteration value for V_i.

Definition 4: Write life-time of variable V_i stored in register R starts at the control step when variable V_i is computed and ends at the control step when variable V_i is used for the first time.

Definition 5: A port is a set of K I/O pins. When variable V is assigned to port P, then V is output or input in its entirety through port P in one control step.

An example of a scheduled and assigned CDFG and the accompanying definitions are depicted using Fig. 3. Registers that store the variables are $R1$, $R2$, and $R3$. Exact clock cycles when operations are executed are also depicted. For example, the last control step, when variable stored in $R1$ is used, is $C1$. Since no variable is stored in $R1$ after $C1$ until $C3$, it is said that the read life-time of the variable stored in $R1$ spreads over the entire iteration. Write life-time of a variable can be observed on the example of variable stored in $R2$. This variable is computed at control step $C1$ and used for the first time in the next consecutive control step. Hence, its write life-time includes only the control step $C1$. There are no restrictions imposed on the type of data operations. Since we target debugging designs at the behavioral level, we consider operations such as addition, subtraction, multiplication, and division.

B. Cut Selection

In this Section, we introduce two definitions of a cut of an SDF computation and present effective algorithms for cut

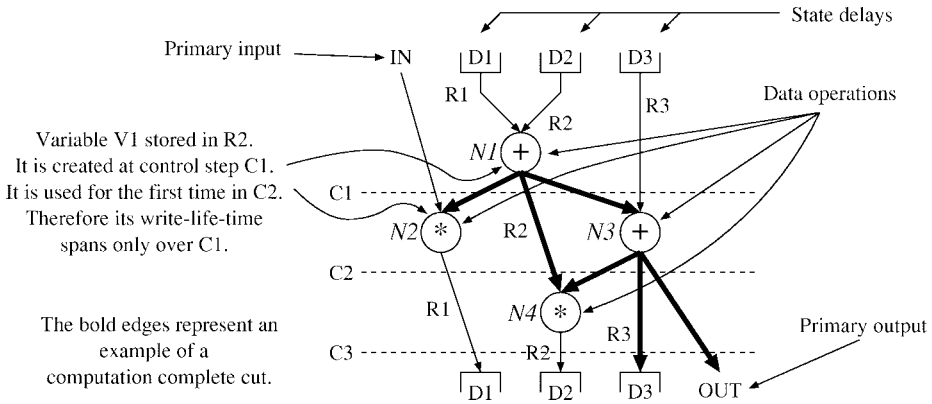

Fig. 3. An example of a scheduled and assigned control data flow graph and the accompanying definitions. Primary inputs and outputs, state delays, data operations, data precedence edges, register assignment, variable write life-time, and a complete cut example are illustrated.

selection and allocation of hardware resources that enable I/O of the cut state. The two different cut definitions enable exploration of certain tradeoffs in the cut-selection process. The first definition of a cut imposes a limitation that all contained variables must be selected from a single computation iteration. The second definition relaxes this requirement by enabling the search for a cut-set to be conducted among variables in several consecutive computation iterations. While cuts which obey the first definition require smaller trace capturing devices and induce lower computation initiation start-up times, the cuts formed according to the second definition frequently require less hardware resources.

Definition 1—Single Iteration Complete Cut: A complete cut is a set of variables generated within one computation iteration that cuts all possible paths in the computation.

Therefore, the goal of the cut-set search algorithm is to, given a computation control data flow graph, find a register subset of minimal cardinality that stores all the variables of at least one complete cut. Before we commence with the algorithm description, note that only observability-related algorithms will be presented. The algorithms that support controllability are identical with the exception that write life-times are used in place of read life-times. If the designer desires to use the same cut for observing and controlling the computation, then the cut should be determined by the controllable version of the proposed algorithms.

The initial problem formulation is determined using the standard Garey–Johnson format [7].

PROBLEM—Optimal Cut-Set for Debugging:

INSTANCE: Given a control data flow graph with read life-times of its variables, variable-to-register assignments, P ports, set S of control steps when each port is busy, and integer K.

QUESTION: Is there a subset of variables V such that each path in the control data flow graph CDFG contains at least one variable $V_i \in V$, the cardinality of the set of registers that contains each variable $V_j \in V$ equals K, and there exists such schedule that each variable $V_j \in V$ can be output through P ports at control steps not included in S?

The NP-completeness of this problem can be proven by restriction to the FEEDBACK ARC SET problem (GT8, [7, p. 192]). The restriction is made by assuming that no hardware sharing is possible, i.e., each variable V_i in the CDFG is stored in a separate register R_i. Since even the problem of proving whether an arbitrary set of variables represents a CDFG cut is of polynomial linear complexity, we transform the problem into a computationally less demanding task and apply problem partitioning and most-constrained least-constraining heuristics as the fundamental approach to search the transformed solution space. The pseudocode of the developed algorithm is presented in Fig. 4.

```
InputSensitiveGraph ISG = Construct_ISG(CDFG).
ISG = Input_Sensitive_Transitive_Closure(ISG, CDFG).
Repeat
  D = Input_Sensitive_Dominating_Set(ISG).
until Schedule(D, ListOfPorts) != EXISTS.
```

Fig. 4. Pseudocode for the cut search algorithm.

The developed algorithm constructs the solution based on the analysis of the input-sensitive (ISG) representation of the original CDFG. The ISG is built from the CDFG according to the pseudocode in Fig. 5. The idea behind this transformation is to create a graph-like structure which enables fast check whether a subset of variables is a cut. Each node in the ISG represents an operation from the CDFG, contains a single output that represents the output variable of that operation, and contains a number of inputs which represent the operands. A cut of a computation is a selection of node outputs which covers all inputs of all nodes. An important step in the algorithm is the input-sensitive transitive closure operation which builds the dependencies between operations, i.e., nodes in the ISG. This operation is described using the pseudocode in Fig. 6. An example ISG, which corresponds to the CDFG illustrated in Fig. 3, is shown in Fig. 8. The dotted edges are added while applying the input-sensitive transitive closure procedure.

Using the definition of scheduled and assigned ISG, the initial problem can be reformulated into the following standard Garey–Johnson format.

PROBLEM—Optimal Input Dominating Set of an ISG:

INSTANCE: ISG with read life-times of its variables, variable-to-register assignments, P ports, associated set S of control steps when each port is busy, and integer K.

QUESTION: Is there an input dominating set of variables V such that each input is covered with at least one variable in V, the cardinality of the set of registers that contains all

For each node $N_i \in CDFG$
  Create a node $M_i \in ISG$.
For each edge E_{N_i,N_j} directed from N_i to N_j
  Create an input port $M_{j,m}$ for M_j, where m is the index of the input port.
  For each edge E_{N_i,N_j} directed from N_i to N_j
    Create an edge $E_{M_i^o,M_{j,m}}$ which connects M_i^o and $M_{j,m}$,
    where M_i^o is the output port of M_i, and $M_{j,m}$ is m^{th} input of M_j.

Comment: Each primary input P_i of the $CDFG$ is ignored.
Schedule and assign each ISG node (variable) as its parent
$CDFG$ node (variable).

Fig. 5. Construct_ISG(CDFG)—input-sensitive graph construction pseudocode.

For each pair of edges $E_{M_a^o,M_{b,m}}, E_{M_b^o,M_{c,m}} \in ISG$ such that
  $E_{M_a^o,M_{b,m}}$ connects M_a to M_b and $E_{M_b^o,M_{c,m}}$ connects M_b to M_c
  If the difference, in control steps, between the starts of read life-times of nodes (variables)
  a and c is less or equal than the total number of control steps in one iteration
    Insert edge $E_{M_a^o,M_{c,m}}$ connecting M_a and $M_{c,m}$.

Fig. 6. Input_Sensitive_Transitive_Closure(ISG, CDFG).

Preprocessing: Set of primary output variables $V_i^{output} \in ISG$ is static part of the cut.
Variables read-alive during the entire iteration are static part of the optimal cut.

  Select a random subset of nodes $CUT \in ISG$, such that
  CUT covers all node inputs in ISG. Set best solution $CUT* = CUT$.
  If there does not exist a schedule such that CUT can be output
  through P ports at control steps not included in S do **Preprocessing**.
  Repeat $GLOBAL$ times
    Unselect random subset of nodes $\in CUT$ such that at least one node input remains uncovered.
    Randomly select a subset of nodes *subCUT* from (ISG - CUT) which covers
    the uncovered set of inputs. Merge *subCUT* and CUT.
    If Cost(CUT) < Cost($CUT*$)
      If there exists a schedule such that CUT can be output through P ports set $CUT* = CUT$.
        Repeat $LOCAL$ times
          $CUT+ = CUT*$
          Unselect random subset of nodes $\in CUT+$ such that at least one node input remains uncovered.
          Randomly select a subset of nodes *subCUT* from (ISG - $CUT+$) which covers
          the uncovered set of inputs. Merge *subCUT* and $CUT+$.
          If Cost($CUT+$) < Cost($CUT*$)
            If there exists a schedule such that CUT can be output
            through P ports set $CUT* = CUT+$.

Return: $CUT*$ as the optimal input dominating set (optimal cut-set).

Fig. 7. Input_Sensitive_Dominating_Set(ISG).

variables from V equals K, and there exists a schedule such that all variables in V can be output through P ports at control steps not included in S?

The NP-completeness of this problem can be proved by reduction to the GRAPH DOMINATING SET problem (GT2, [7, p. 190]). The restriction simplifies the ISG in such a way that each node has only one input and each variable $V_i \in$ ISG is stored in a dedicated register R_i.

We developed a novel heuristic algorithm for this problem. The pseudocode of the proposed heuristic technique is presented in Fig. 7. The algorithm generates a large number of candidate node subsets which have the property of being cuts. Although, the cuts are generated probabilistically uniformly with respect to the search time, the most-constrained least-constraining objective function that evaluates the candidate cut-sets is run-time dependent. In the beginning, the algorithm favors most-constrained solutions with few storing registers. As the search progresses, the cost dominating factor becomes the cumulative length of read life-times of all selected variables (such variables are least-constraining). At that point variables with nonoverlapping read life-times are also favored. This approach enables more assignment freedom for the final variable-to-port scheduling.

The main disadvantage of defining a cut according to *the first definition* is the fact that it does not have the flexibility of outputting a computation cut over a consecutive span of computation iterations. The following technique of selecting a computation cut enables this property and reduces the amount of hardware resources augmented for cut I/O.

Definition 2—Multiple Iteration Complete Cut: A complete cut is a subset of variables which bisects all cyclic paths in the control data flow graph of a computation.

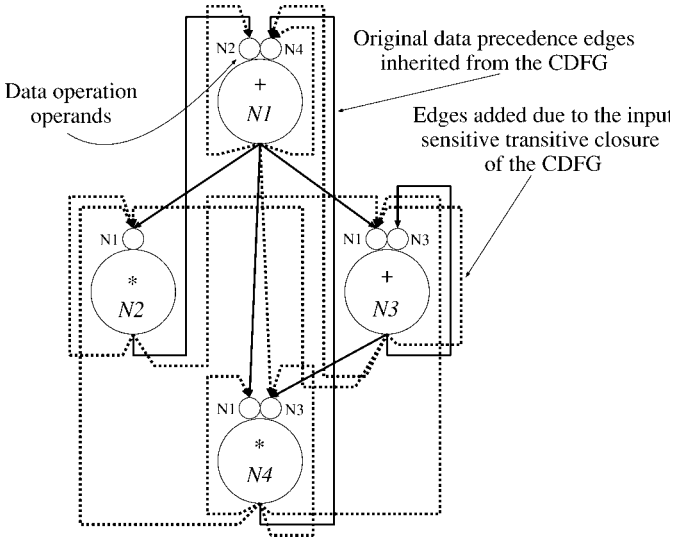

Fig. 8. Example of an ISG which corresponds to the CDFG shown in Fig. 3. Each node corresponds to a data operation N_i in the original CDFG and has a set of inputs which correspond to the operands of N_i. The edges in the graph are either inherited from the original CDFG or created using the input-sensitive transitive closure procedure.

The targeted optimization problem of finding a cut which is contained in minimal number of registers and a schedule according to which all variables of the cut can be output, can be defined using the standard Garey–Johnson format.

PROBLEM—Optimal Cut-Set for Debugging (II):

INSTANCE: Given a control data flow graph with read lifetimes of its variables, variable-to-register assignments, P ports, associated set S of control steps when each port is busy, and integer K.

QUESTION: Is there a subset of variables V, such that when removed from the CDFG leaves no directed cycles in the CDFG, the cardinality of the set of registers that contains all variables from V equals K, and there exists such schedule that each variable $V_j \in V$ can be output through P ports at control steps not included in S?

The specified problem is an NP-complete problem since there is an one-to-one mapping between the special case of this problem, when all operations in the computation are executed exactly the same number of times, and the FEEDBACK ARC SET problem (GT8, [7, p. 192]). 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 (SCC's) using the depth-first search algorithm [1]. 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 SCC's 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 nontrivial SCC's.

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. For each edge, a list of registers that store the compacted variables is maintained.

In the next step, an objective function decides which node (variable) in the current set of SCC's is to be deleted. The function analyzes, for the deletion of each vertex, the cardinality of the newly created set of SCC's, the registers that are storing the variables in these SCC's, the length of the read lifetimes of variables in the SCC's, and the vertex cardinalities of the new set of SCC's. The vertex that results in the smallest objective function is deleted 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, candidate node deletion evaluation, node deletion, and graph updating is repeated while the set of nontrivial SCC's in the graph is not empty. The set of nodes (variables) deleted from the computation represents the final cut-set selection.

Consider the example shown in Fig. 10. The CDFG of the third-order Gray–Markel ladder IIR filter, shown in Fig. 11, has only one nontrivial SCC. The graph compaction step is explained using Fig. 10(a)–(c). Initially, vertex B is merged with vertex A, which imposes variable W to be merged with variable V. Next, the shaded nodes in Fig. 10(b) are merged as well as the corresponding variables. The node compaction process results in a SCC presented in Fig. 10(c). Fig. 10(e) illustrates the resulting set of SCC's, after node M is deleted from the SCC depicted in Fig. 10(d).

Finally, let's compare the cuts retrieved according to the two different definitions on the example of a Gray–Markel ladder filter. In Fig. 11, the variables of a cut-set that corresponds to the first definition are represented as bold dotted lines. In order to define a single iteration, the output variables of adders $A6$, $A8$, and $A9$ stored in registers $R4$, $R2$, and $R3$, respectively, are output as a cut. Such a set of variables fully determines the state of the machine.

Consider the cut that corresponds to the second definition. It contains three variables: the outputs of adders $A1$, $A3$, and $A5$, all stored in register $R1$. This subset of variables bisects all cyclic paths in the CDFG; by deleting the edges in the CDFG which represent these variables, all cyclic paths are removed. In order to use these variables in restarting verification, cut values from three consecutive iterations are required before the machine state is correctly restored.

C. Variable Scheduling

The design has to be able to output all variables in the cut-set through a limited number of I/O ports within a single or multiple computation iterations. Since the procedure which checks whether this is achievable is invoked every time a candidate cut-set is found, we propose a most-constrained least-constraining heuristic technique to quickly provide an answer to this question. If the answer is positive, a search for the minimal cardinality set of register-to-port interconnects is performed. The interconnects are such that cut-set variables can be output through the ports at idle control steps. In this Section we present the algorithm for the first subproblem. The problem can be formulated using the following format.

```
Create a set SCC = ComputeScc(CDFG(V, E)) of strongly connected components [Aho83]
 For each SCC_i ∈ SCC
   If |SCC_i| = 1 delete SCC_i from SCC
Repeat LOOPS times
 Repeat
   CUT = null
   While SCC ≠ empty
    For each SCC_i ∈ SCC
     GraphCompaction(SCC_i)
     For each node V_{i,j}
      S = ComputeScc(SCC_i − V_{i,j})
      OF(S) = (1 + α) Σ_{i=1}^{|S|} (|S_i| · Edges(S_i) · LifeTime(S_i)) · Registers(S)^4),
      where α is random number α ∈ {0, 1/|SCC|^2}, LifeTime(S_i) returns the read life-time of variables in S_i,
      and Registers(S) returns the number of registers which store all variables in S
     End For
     Select vertex V_{i,j} which results in minimal OF(S(E_{i,j}))
     Delete V_{i,j} from SCC_i
     SCC = S(V_{i,j})
     For each SCC_i ∈ SCC
      If |SCC_i| = 1 delete SCC_i from SCC
     End For
     CUT = CUT ∪ V_{i,j}
    End For
   End While
 until Schedule(D, ListOfPorts) != EXISTS.
If |CUT| < |BESTCUT| then BESTCUT = CUT
Return BESTCUT

Procedure GraphCompaction(SCC_i)
  For each vertex V_i ∈ SCC_i
    If V_i has exactly one input edge E_{j,i} with a source in vertex V_j
      For each edge E_{i,k}
        Create edge E_{j,k}
        Delete E_{i,k}
      End For
      Delete E_{j,i} and V_i
```

Fig. 9. Pseudocode for optimal cut-set for debugging (II) search.

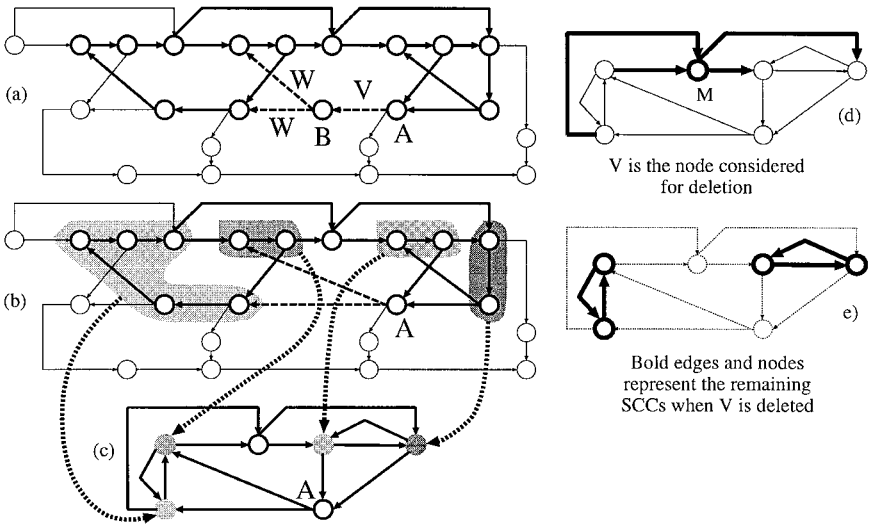

Fig. 10. Finding the cut-set of the third-order Gray–Markel ladder IIR filter. Subfigures (a)–(c) demonstrate the node merger procedure. Subfigures (d)–(e) illustrate the removal of a node from the set of SCC's and its inclusion in the set of selected cut variables.

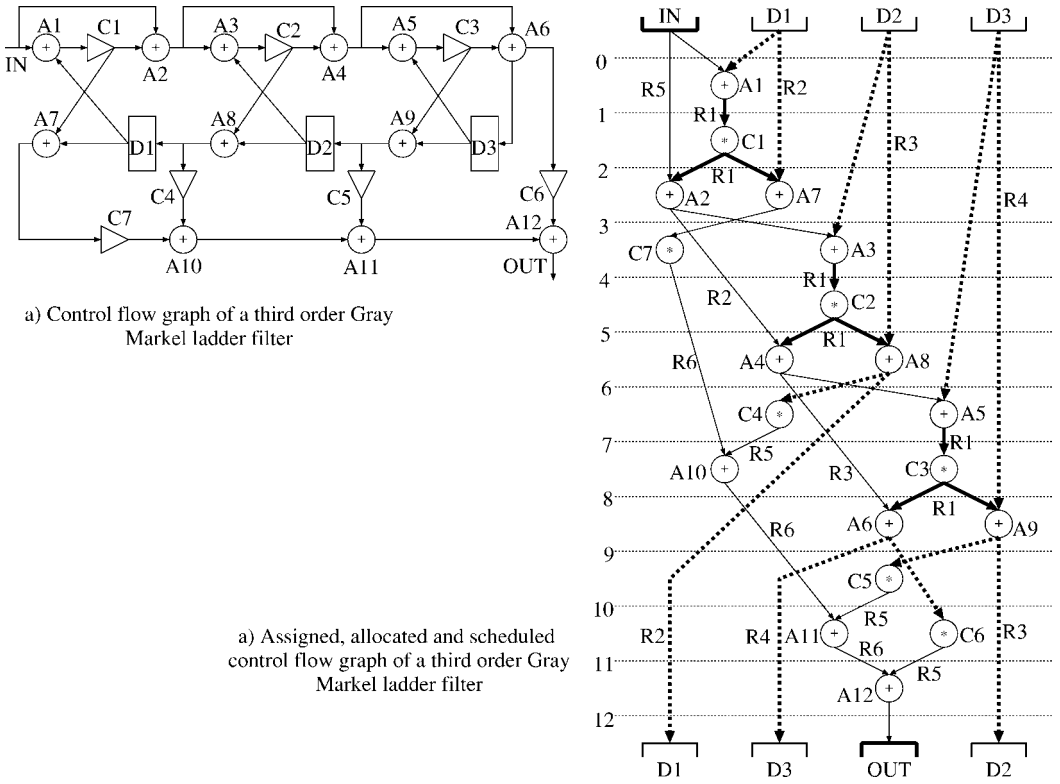

Fig. 11. (a) The unscheduled and (b) scheduled and assigned control data flow of a third-order Gray–Markel ladder filter.

Repeat until no additional variable can be scheduled
  For each variable V_i
    If V_i can be scheduled only in one control step C_j,
    schedule V_i to C_j and any port P_x available at C_j.
  For each control step C_i
    Set C_i^{var} as number of all variables which are read-alive at C_i.
  Schedule variable V_i with $max(Cost(V_i)) = max(\frac{\sum_{j=AllControlSteps}(C_j^{var} \cdot ReadAlive(V_i,C_j)}{ReadLifeTime(V_i)-1})$
  to the control step C_k which has $min(C_k^{var})$ and any port P_x still available at C_k.

$ReadAlive(V_i, C_j)$ returns 1 if V_i is read-alive at C_j.
$ReadLifeTime(V_i)$ is the number of control steps for which V_i is read-alive.

Fig. 12. Pseudocode for the cut-set output scheduling heuristic.

PROBLEM—Output Scheduling of a Set of Variables in a CDFG:

INSTANCE: Set of variables V, each with its read life-time, P ports and associated set S of control steps when each port is busy.

QUESTION: Is there a schedule such that all variables can be output through P ports at control steps not included in S?

The NP-completeness of this problem is proved by restriction to the SEQUENCING WITH RELEASE TIMES AND DEADLINES problem (SS1, [7, p. 236]). The restriction is imposed by selecting only those variables in V that are not in the set of state (delay) variables D. The heuristic developed for this problem schedules variables using a greedy strategy. First, the constraint of each control step C_i in the scheduled and assigned CDFG is calculated as the number C_i^{var} of variables being read-alive during that control step. For each variable V_i its constraint is computed as a sum of C_j^{var}, where j is in the set of control steps when V_i is read alive. Consequently, the most constrained variable is assigned to the least constraining control step. The process of computing constraints and scheduling variables to distinct control steps is iterated until all variables in the cut-set are not output scheduled. Pseudocode of the proposed most-constrained least-constraining heuristic is presented in Fig. 12.

The algorithm is described using Fig. 13. There are 12 variables in the cut-set, three output ports which are all busy during control step C_4. The schedule is found using the described heuristic by sequentially assigning variables to ports as depicted. First, variables V_1, V_8, and V_{12} are scheduled to C_5, and V_{11} to C_3 since they do not have a choice. In the next step, since all ports are used at C_5, we schedule V_9 to C_1 and V_6 to C_3. Next, variable V_{10} is the most-constrained according to the formula in the pseudocode so we schedule it to its least-constraining control step C_2. Then V_7 and V_5 have

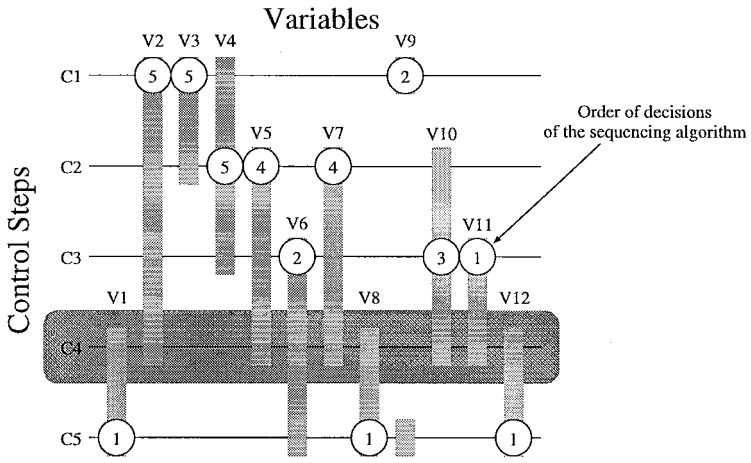

Fig. 13. Example of output scheduling.

```
Interconnects = |R|
Repeat LOOPS times
  Repeat Interconnects times
    Connect register R with max(ProbabilisticCost(R)) to port P with max(ProbabilisticCost(P))
  End Repeat
  If BipartiteMatching(variables, ports, interconnects) == EXISTS
    Decrease(Interconnects)
End Repeat
```

Fig. 14. Pseudocode for the variable-to-port scheduling heuristic.

no choice and have to be scheduled at C_2. Finally, V_2, V_3, and V_4 are scheduled to C_1 and C_2 according to the already described principles.

D. Variable-to-Port Scheduling

The second phase of the synthesis for the debugging process has as an input the selected cut-set from the first phase and the information about the read life-times of each variable as well as its storing register. The available number of output ports is also available. The key design question is to assign each cut-set variable to a specific output port in such a way that the number of connections between registers and I/O ports is minimal. We present the problem in the standard Garey–Johnson format.

PROBLEM—Optimal Output Scheduling of a Set of Variables in a CDFG for Debugging:

INSTANCE: An ordered set of variables V, each with its read life-time and designated register, P ports, associated set S of control steps when each port is busy, and integer K.

QUESTION: Is there a schedule such that all variables can be output from the chip through P ports at control steps not included in S, and the cardinality of register-port connections equals K?

If the optimization demand in the problem is ignored, the proof that this problem is NP-complete is equivalent to the NP-completeness proof for the scheduling problem described in Section IV-C. We have partitioned this problem into two fully modular optimization subproblems. Initially, register-to-port assignment is performed such that optimization requirements are met and variables of the proposed cut-set can be output through all designated ports. In the second phase, for each port P_i, all variables assigned to P_i are scheduled for output. The pseudocode for the combined algorithm is presented in Fig. 14.

For the assignment problem, the developed heuristic iteratively tries to assign as few as possible most-constrained registers to least-constraining ports such that the final variable-to-port scheduling is achievable. Objective function that quantifies the constraint of register R is given as

$$\text{Cost}(R) = \sum_{V_i \in R} \sum_{j=\text{All Control Steps}} \frac{C_j^{\text{var}}}{\text{ReadLifeTime}(V_i) - 1}$$

where ReadLifeTime(V_i) returns the number of control steps for which V_i is read-alive. Similarly, the objective function that quantifies the constraint of port P_x is

$$\text{Cost}(P_x) = \sum_{j=\text{All Variables}} \cdot \frac{\text{ReadLifeTime}(V_j) \cdot \text{Assigned}(V_j, P_x)}{1 + \text{Number of Registers Already Assigned to} P_x}$$

where Assigned(V_j, P_x) returns one if V_j is already assigned to P_x. Otherwise, it returns zero.

In the conducted experiments we used a probabilistic version of the described heuristic which iteratively generated randomized solutions. The solutions were still guided by the described objective functions augmented with a certain random offset.

Once registers are assigned to ports, variable-to-port scheduling is performed only for variables stored in registers as-

TABLE I
APPLICATION OF THE DESIGN-FOR-DEBUGGING STEP TO A SET OF STANDARD BENCHMARKS FOR ESTIMATION OF HARDWARE OVERHEAD

Design		Structure					Complete Cut - Def.1			Complete Cut - Def.2		
Description	Hyper optim.	Control steps	Critical path	Variables	Registers	States	Variables	Registers	Ports added	Variables	Registers	Ports added
8th Order	NO	18	18	35	19	8	8	1	0	8	1	0
Continued	NO	36	18	35	19	8	8	1	0	8	1	0
Fraction	YES	4	4	49	30	8	8	6	2	8	6	2
IIR Filter	YES	8	4	49	29	8	10	5	1	10	5	1
Linear GE	NO	12	12	48	19	5	8	3	0	8	3	0
Controller 1	NO	24	12	48	23	5	13	1	0	13	1	0
	YES	6	6	48	27	5	5	5	1	5	5	1
	YES	12	6	48	26	5	8	4	0	8	4	0
Wavelet	NO	16	16	31	20	15	15	1	0	1	1	0
Filter	NO	32	16	31	20	15	15	1	0	1	1	0
	YES	1	1	31	31	15	15	15	15	1	1	1
	YES	2	1	31	31	15	15	15	7	1	1	0
Modem	NO	10	10	33	16	8	12	2	0	4	1	0
Filter	NO	20	10	33	15	8	12	1	0	4	1	0
	YES	4	4	47	29	8	8	6	2	8	6	2
	YES	8	4	47	27	8	11	4	1	8	4	1
Volterra	NO	12	12	28	15	4	4	1	0	4	1	0
2nd	NO	12	24	28	15	4	4	1	0	4	1	0
order	YES	6	6	28	19	4	4	1	0	4	1	0
filter	YES	6	12	28	17	4	4	1	0	4	1	0
Volterra	NO	20	20	50	22	6	6	1	0	6	1	0
3rd order	NO	20	40	50	22	6	6	1	0	6	1	0
nonlinear	YES	8	8	50	31	6	6	1	0	6	1	0
filter	YES	8	16	50	27	6	6	1	0	6	1	0
Controller	NO	15	15	114	38	14	14	6	0	11	3	0
VSTOL	NO	30	15	114	37	14	14	4	0	11	2	0
aircraft	YES	6	6	114	46	14	14	9	2	14	9	2
	YES	12	6	114	44	14	14	5	1	14	5	1
Digital to	NO	132	132	354	167	74	76	3	0	2	1	0
Analog	NO	132	264	354	171	74	77	2	0	2	1	0
Converter	YES	5	5	398	189	74	76	29	18	2	1	0
	YES	10	5	398	178	74	76	28	8	2	1	0
Motorola	NO	134	132	217	121	133	133	67	0	1	1	0
C-133	NO	268	268	217	128	133	133	67	0	1	1	0
filter	YES	1	1	217	217	133	133	133	133	1	1	0
	YES	2	1	217	129	133	133	67	67	1	1	0
Long	NO	2566	2566	1082	1056	1024	1027	5	0	2	1	0
Echo	NO	5132	2566	1082	1061	1024	1027	4	0	2	1	0
Canceler	YES	1088	1088	1107	1064	1024	1028	6	0	2	1	0
	YES	2176	1088	1107	1059	1024	1027	5	0	2	1	0

signed to a single port. This problem is equivalent to the problem of *maximum bipartite matching* ([4, p. 601]) which can be efficiently solved in polynomial time. A bipartite graph G for each port P is constructed. The first partition of G contains a node for each control step in the computation iteration for which port P is not busy. A node in the other partition is created for each variable assigned to port P. Edges are drawn between each node which represents variable V and all nodes in the first control-step partition associated with control steps for which V is read-alive. To efficiently solve this problem we use the Ford–Fulkerson method which runs in $O(VE)$, where V is the number of vertices and E is the number of edges in graph G [4].

The presented algorithm assumes that the required number of I/O ports is sufficient to enable the I/O of the cut state. Since this number is often small and not known at design time, we perform an exhaustive binary search for the smallest number of I/O ports, which can satisfy the scheduling constraints. This search is performed as an outer shell to the scheduling and variable-to-port matching heuristic.

V. EXPERIMENTAL RESULTS

In order to evaluate the developed debugging technique and accompanying algorithms, we have applied them on several benchmark designs. The examples were collected from the following technical manuscripts: eighth-order continued fraction IIR filter, linear GE controller, Volterra filters, long echo canceler, wavelet filter, modem filter, Motorola C133 filter [8], [16], [17], and a real-life avionics VTSOL controller [10]. For all experiments, HYPER was used as a behavioral compiler to obtain register-transfer level (RTL) implementations [17]. Design-for-debugging analysis was performed to determine the cuts and the extra hardware overhead needed to support the proposed debugging technique. Table I, column 1, lists the set of designs evaluated. For each design, an optimized (first and second rows) and a nonoptimized version (third and fourth rows) were used. Additionally, on each version, both tight and more relaxed performance constraints were assumed. Optimized versions were obtained by applying scripts for speed optimizations [9]. Available control step budgets, equal

to the computation's critical path and twice that amount, were used for the tight and more relaxed performance constraints, respectively.

Table I, columns 3–7, describe the behavioral structure of the designs in the form of available control steps, critical path, number of variables, registers used in the RTL implementation, and computation states. Columns 8–10 and 11–13, display the structural properties of cuts obtained using the first and second cut definition, respectively: the number of cut variables, number of registers used by the cut variables (each of which require register-to-port connections), and additional ports needed. The experimental results point to the advantage of selecting computation cuts according to the second definition for designs which do not have large numbers of strongly connected components because of smaller cut-set cardinalities. This advantage comes at the expense of longer startup sequences during computation initialization. For example, in order to initialize correctly the computation for all design cases of the Motorola C-133 filter, its cut variable (according to the second cut definition) has to be input/output throughout 133 consecutive computation iterations. In the attempt to input/output the cut in a single iteration, 67 (or in the extremely constrained case, 133) different variables have to be transferred through the I/O pins of the ASIC.

In 21 out of 80 cases, the available I/O ports were sufficient to support full observability and controllability. For example, the nonoptimized modem filter shown in row 9 used two registers and required no extra ports to support its 12 variable cut. However, in several cases, extra ports were needed to obtain a design implementation, which was enabled for cut-based debugging. Extra ports were needed only for the highly optimized designs with exceptionally high sampling rates. In particular, the optimized wavelet filter and digital-to-analog converter have cuts of 15 and 76 variables, respectively. To support the required I/O of these variables, an additional 15 and 18 ports, respectively, were needed, since existing ports were already fully utilized for I/O of the primary input and output. It is important to stress, that such design constraints are rarely imposed on the behavioral compilers.

VI. Conclusion

The run-time of a design simulation results in several orders of magnitude slower functional execution with respect to emulation or fabrication. Design emulation and implementation significantly restrict design controllability and observability during functional debugging. We introduce a new cut-based functional debugging paradigm which integrates design emulation and simulation in such a way that advantages of both domains are fully utilized and result in a design approach which enables fast debugging with complete observability and controllability. We have identified the associated optimization synthesis tasks, established their computational complexity, and developed most-constrained least-constraining heuristics to solve them. The experimental results clearly indicate the power of the approach and new debugging tool on a number of real-life design examples with minimal hardware overhead.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms.* Reading, MA: Addison-Wesley, 1983.

[2] S. S. Bhattacharyya and E. A. Lee, "Scheduling synchronous dataflow graphs for efficient looping," *J. VLSI Signal Processing,* vol. 6, no. 3, pp. 271–288, 1993.

[3] J. Cocke, R. L. Malm, and J. J. Shedletsky, "Logic simulation machine," U.S. Patent 4 306 286, issued 1981.

[4] T. H. Cormen, C. E. Leisserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA: MIT Press, 1990.

[5] G. De Micheli, *Synthesis and Optimization of Digital Circuits.* New York: McGraw-Hill, 1994.

[6] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, Design of embedded systems: Formal models, validation, and synthesis, *Proc. IEEE,* vol. 85, pp. 366–390, Mar. 1997.

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

[8] L. Guerra, M. Potkonjak, and J. Rabaey, "High level synthesis for reconfigurable datapath structures," in *Proc. Int. Conf. Computer-Aided Design,* 1993, pp. 26–29.

[9] I. Hong, D. Kirovski, and M. Potkonjak, "Potential-driven statistical ordering of transformation," in *Proc. Design Automation Conf.,* 1997, pp. 347–352.

[10] R. A. Hyde and K. Glover, "The application of scheduled $H_{|\infty}$ controllers to a VSTOL aircraft," *IEEE Trans Automat. Contr.,* vol. 38, pp. 1021–1039, July 1993.

[11] D. Kirovski, M. Potkonjak, and L. M. Guerra, "Functional debugging of systems-on-chip," in *Proc. Int. Conf. Computer-Aided Design,* 1998, pp. 525–528.

[12] D. L. Liu, J.-T. Li, T. B. Huang, and K. S. K. Choi, "Method and apparatus for debugging reconfigurable emulation systems," U.S. Patent 5 425 036, issued 1995.

[13] J. Marantz, "Enhanced visibility and performance in functional verification by reconstruction," in *Proc. Design Automation Conf.,* 1998, pp. 164–169.

[14] C. Maunder, "JTAG, the Joint Test Action Group," *Inst. Elect. Eng. Colloquium New Ideas in Testing,* pp. 6/1–6/4, 1986.

[15] M. Poret and J. McKinley, "In-circuit emulator," U.S. Patent 4 674 089, issued 1987.

[16] M. Potkonjak, S. Dey, and K. Wakabayashi, "Design-for-debugging of application specific designs," in *Proc. Int. Conf. Computer-Aided Design,* 1995, pp. 295–301.

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

[18] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," *IEEE Parallel Distributed Technol.: Syst. Applicat.,* vol. 3, no. 4, pp. 34–43, winter 1995.

[19] G. L. Swoboda, M. D. Daniels, and J. A. Coomes, "Emulation devices, systems and methods utilizing state machines," U.S. Patent 5 329 471, issued 1994.

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

[21] V. Zivojnovic and H. Meyr, "Compiled HW/SW co-simulation," in *Proc. Design Automation Conf.,* 1996, pp. 690–695.

Darko Kirovski, for a biography, see p. 1326 of the September 1999 issue of this Transactions.

Miodrag Potlonjak, for a biography, see p. 1326 of the September 1999 issue of this Transactions.

Lisa M. Guerra received th B.S. degree in electrical engineering from Stanford University, Stanford, CA, in 1990 and the Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley in 1996.

Since 1996, she worked for several years on system design methodologies and system-on-chip verification at Conexant Systems, Newport Beach, CA. She is currently a Software Startup Engineer. She holds one U.S. patent.

Dr. Guerra was an AT&T Bell Labs and an Office of Naval Research scholar.