Using Global Code Motions to Improve the Quality of Results for High-Level Synthesis

Sumit Gupta, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau

***Abstract*—The quality of synthesis results for most high-level synthesis approaches is strongly affected by the choice of control flow (through conditions and loops) in the input description. This leads to a need for high-level and compiler transformations that overcome the effects of programming style on the quality of generated circuits. To address this issue, we have developed a set of speculative code-motion transformations that enable movement of operations through, beyond, and into conditionals with the objective of maximizing performance. We have implemented these code transformations, along with supporting code-motion techniques and variable renaming techniques, in a high-level synthesis research framework called *Spark*. *Spark* takes a behavioral description in ANSI-C as input and generates synthesizable register-transfer level VHDL. We present results for experiments on designs derived from three real-life multimedia and image processing applications, namely, the MPEG-1 and -2 and GNU image manipulation program applications. We find that the speculative-code motions lead to reductions between 36% and 59% in the number of states in the finite-state machine (controller complexity) and the cycles on the longest path (performance) compared with the case when only nonspeculative code motions are employed. Also, logic synthesis results show fairly constant critical path lengths (clock period) and a marginal increase in area.**

***Index Terms*—Code motions, embedded systems, high-level synthesis, parallelizing compilers, speculation.**

I. Introduction

Recent years have seen the widespread acceptance and use of language-level modeling (such as VHDL and Verilog) of digital designs. Increasingly, the typical design process starts with design entry in a hardware description language at the register-transfer level (RTL), followed by logic synthesis. Furthermore, with the advent of systems-on-a-chip, system level behavioral modeling in high-level languages is being used for initial system specification and analysis. All of these factors have led to a renewed interest in high-level synthesis from behavioral descriptions [1]–[8].

However, current synthesis efforts have several limitations: synthesizability is guaranteed on a small, constrained subset of the input language and the language level optimizations are few and their effects on final circuit area and speed are not well understood. The quality of synthesis results (in terms of circuit delay and area) is adversely affected by the presence of conditionals and loops. Designers are often given minimal control over the transformations that effect these results. All these factors continue to limit the acceptance of high-level synthesis tools among designers.

To alleviate the problem of poor synthesis results in the presence of complex control flow in designs, there is a need for high-level and compiler transformations that can optimize the synthesis results irrespective of the choice of control flow in the input description. To address this issue, several scheduling algorithms have been proposed that employ beyond-basic-block code motion techniques such as speculation to extract the inherent parallelism in designs and increase resource utilization [4]–[7].

Generally, speculation refers to the unconditional execution of operations that were originally supposed to have executed conditionally. However, we found that there are situations when there is a need to move operations *into* conditionals. This may be done by *reverse speculation*, where operations before conditionals are duplicated into *subsequent* conditional blocks and, hence, executed conditionally, or this may be done by *conditional speculation*, wherein an operation from after the conditional block is duplicated *up* into *preceding* conditional branches and executed conditionally. Another code-motion technique we developed, called *early condition execution*, evaluates conditional checks as soon as their data dependencies are satisfied. In this way, all of the operations in the branches of the conditional are ready to be scheduled immediately.

A number of similar code transformations have been proposed for compilers as well. Whereas compilers often pursue maximum parallelization by applying speculative-code motions, in high-level synthesis, such code transformations have to be selected and guided based on their effects on the control, interconnect, and area costs. An important contribution of our work is in heuristics that select the code transformations so as to improve the overall synthesis results. In some cases, our heuristics actually end up increasing the number of operations by duplicating them into conditional blocks.

We, thus, present a priority-based global-list scheduling heuristic that directs these code motion transformations and obtains significant reductions in schedule lengths and controller complexity. Area overheads are kept in check by using a resource binding technique that minimizes interconnect [9]. We implemented the code-motion transformations, the scheduling heuristic, and control and binding passes in a modular and extensible high-level synthesis framework called *Spark* [10]. We also modified several parallelizing compiler transformations [11], [12] for high-level synthesis and implemented them within our framework. *Spark* produces synthesizable RTL VHDL and, thus, enables evaluation of the effects of several coarse and fine-grain optimizations on logic synthesis results. The input language for *Spark* is ANSI-C, currently with the restrictions of no pointers, no unstructured jumps (gotos), and no function recursion.

The rest of this paper is organized as follows. The next section reviews previous related work. In Section III, we present the intermediate representation used by *Spark*. In Section IV, we describe a set of speculative code motions that are useful in high-level synthesis. Next, we present the *Spark* framework and, in Section VI, we describe the scheduling heuristic. In Section VII, we present scheduling and logic synthesis results for experiments using the code motions.

II. Related Work

Early high-level synthesis work focused on data-flow designs and applied optimizations, such as algebraic transformations, retiming, and code motions across multiplexers for improved synthesis results [13]–[16]. Recent work has demonstrated the effectiveness of speculation in improving schedule lengths for designs with control flow. CVLS [4] uses condition vectors to improve resource sharing among mutually exclusive operations. Radivojevic *et al.* [17] and Haynal [6] present exact symbolic and automata-based formulation for designs with control flow. The "Waveschedule" approach [18] employs speculative execution to minimize the expected number of cycles in the schedule of a design. Santos *et al.* [19] and Rim *et al.* [20] support generalized code motions during scheduling of designs with control flow.

Manuscript received October 6, 2001; revised May 20, 2002 and February 1, 2003. This work was supported by the Semiconductor Research Corporation under Task I.D. 781.001. This paper was recommended by Associate Editor R. Camposano.

S. Gupta, N. Savoiu, N. Dutt, and A. Nicolau are with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA (e-mail: sumitg@cecs.uci.edu; savoiu@cecs.uci.edu; dutt@cecs.uci.edu; nicolau@cecs.uci.edu).

R. Gupta is with the Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093 USA (e-mail: rgupta@ucsd.edu).

Digital Object Identifier 10.1109/TCAD.2003.822105

A range of code transformation techniques similar to those presented in our work have also been previously developed for high-level language compilers (especially parallelizing compilers) [21]–[23]. Although the basic transformations (e.g., dead code elimination, copy propagation) can be used in synthesis as well, other transformations need to be modified for synthesis by incorporating ideas of mutual exclusivity of operations, resource sharing, and hardware cost models.

The contributions of this work include: 1) three code-motion transformations derived from speculative execution techniques that are specifically targeted for high-level synthesis; 2) a heuristic approach to drive the application of these transformations; and 3) a framework that provides a toolbox of code transformations and supporting compiler transformations. This enables the designer to apply heuristics to drive selection and control of individual transformations under realistic cost models for high-level synthesis. The synthesis framework provides a path from an input behavioral description down to RTL code that is synthesizable by logic synthesis tools.

III. Design Description Modeling

The *Spark* synthesis framework accepts a behavioral description in ANSI-C as input. However, we do not support pointers, unstructured jumps (gotos), and recursive functions for synthesis. The input C code is a sequential description of statements. Statements may be operation expressions, conditional constructs (if-then-else, switch-case), and loop constructs (for, while, do-while loops).

Traditionally, control-data flow graphs (CDFGs) [15], [24] have been the primary model for capturing design descriptions for high-level synthesis. CDFGs consist of operation and control nodes with edges for both data flow and control flow. CDFGs work very well for traditional scheduling and binding techniques. However, we found the abstraction level offered by CDFG the range of coarse-grain and fine-grain compiler transformations that we proposed. In particular, loop and conditional structures are not maintained, making it difficult to apply coarse-grain optimizations.

To enable the range of optimizations explored by our work, we use a hierarchical control-flow representation called *hierarchical task graphs* (HTGs) [11], [25] that maintains the control and loop constructs in the design such as if-then-else blocks, for-loops, and while-loops.

Of course, several other representation models have been proposed for high-level synthesis [26]–[29]. However, HTGs are a convenient representation for designs with considerable control constructs since they maintain a structured view of the design. In fact, we maintain HTGs in conjunction with control flow graphs (CFGs) and data flow graphs (DFGs). Thus, whereas CFGs (and CDFGs) are efficient for traversing the basic blocks in a design, HTGs enable higher order manipulation. For example, they enable coarse-grain code restructuring (such as that done by loop transformations [12]) and also provide an efficient way to move operations across large pieces of code [11] (see Section V-A).

In the next two sections, we present the DFGs, CFGs, and the HTGs maintained by our framework and how they all tie in together.

A. DFGs and CFGs

Data dependencies between operations create a partial ordering between the operations. If an operation op_j reads the result of another operation op_i, then a *flow dependency* is said to exist between operations op_i and op_j. Hence, op_j can start execution only after op_i has finished execution. We can define a DFG that captures flow dependencies as follows.

Definition III.1: A **DFG** is a directed acyclic graph $G_{\mathrm{DFG}}(V_{\mathrm{DFG}}, E_{\mathrm{DFG}})$, where the vertices $V_{\mathrm{DFG}} = \{\mathrm{op}_i \mid i = 1, \ldots, n_{\mathrm{ops}}\}$ are the operations in the design, and the edges E_{DFG} represent the flow data dependencies between operations. A directed edge $e_{ij} = (\mathrm{op}_i, \mathrm{op}_j)$, where $\mathrm{op}_i, \mathrm{op}_j \in V_{\mathrm{DFG}}$, exists in E_{DFG} if data produced by operation op_i is read by operation op_j.

A sequence of statements or operations from the input description with no conditionals or loops between them is aggregated into a *basic block*. Whereas the input "C" description consists only of operations that execute sequentially, the high-level synthesis scheduler can schedule operations to execute concurrently. We aggregate operations that execute concurrently into *scheduling steps* within basic blocks. These scheduling steps correspond to control steps in high-level synthesis [15] and to VLIW instructions in compilers [25].

This can be formally stated as follows.

Definition III.2: A **scheduling step** $\mathrm{step}_j = \{\mathrm{op}_k \mid k = 1, 2, \ldots, n\}$ is an aggregation of n operations that execute concurrently in the same cycle or time step.

Definition III.3: A **basic block** $\mathrm{bb}_i = \{\mathrm{step}_j \mid j = 1, 2, \ldots, n_{\mathrm{steps}}\}$ is a maximal sequence of n_{steps} consecutive scheduling steps, where the flow of control enters at the beginning and leaves at the end without halting or possibility of branching except at the end.

The presence of conditional and loop constructs in a design description introduces the notion of *control flow paths*. Control flow can branch into multiple control paths at *fork* (or branch) basic blocks and merge back into a single control flow path at *join* (or merge) basic blocks. We define a CFG that captures this control-flow information as follows.

Definition III.4: A **CFG** is a directed graph $G_{\mathrm{CFG}}(V_{\mathrm{CFG}}, E_{\mathrm{CFG}})$, where the vertices $V_{\mathrm{CFG}} = \{\mathrm{bb}_i \mid i = 1, 2, \ldots, n_{\mathrm{bbs}}\}$ are the basic blocks in the design and the edges E_{CFG} represent the flow of control between the basic blocks. A directed edge $e_{ij} = (\mathrm{bb}_i, \mathrm{bb}_j)$, where $\mathrm{bb}_i, \mathrm{bb}_j \in V_{\mathrm{CFG}}$, exists in E_{CFG} if the signifies that bb_j executes after bb_i has finished execution. bb_i is denoted as a predecessor of bb_j and bb_j as a successor of bb_i. There exists a unique initial basic block $\mathrm{bb}_0 \in V_{\mathrm{CFG}}$ from which all paths in G_{CFG} originate; $\mathrm{FirstBB}(G_{\mathrm{CFG}})$ returns bb_0.

The mapping of operations in the DFG to basic blocks in the CFG is given by the following definition.

Definition III.5: There exists a many-to-one mapping of operations in the DFG $G_{\mathrm{DFG}}(V_{\mathrm{DFG}}, E_{\mathrm{DFG}})$ to the basic blocks in the CFG $G_{\mathrm{CFG}}(V_{\mathrm{CFG}}, E_{\mathrm{CFG}})$ given by $\mathrm{BB}_{\mathrm{Ops}}$: $V_{\mathrm{DFG}} \mapsto V_{\mathrm{CFG}}$. Thus, $\forall\, \mathrm{op}_i \in V_{\mathrm{DFG}}, \mathrm{BB}_{\mathrm{Ops}}(\mathrm{op}_i)$ gives the basic block $\mathrm{bb}_j \in V_{\mathrm{CFG}}$ to which operation op_i is mapped.

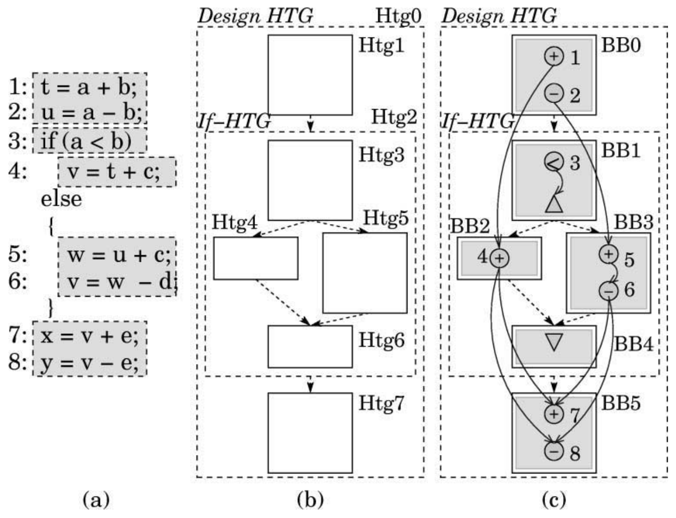
B. HTGs: A Model for Control-Intensive Designs

Whereas CFGs are useful for maintaining the flow of control between basic blocks, we employ HTGs to maintain structure of the design description. We define a HTG as follows.

Definition III.6: An **HTG** is a directed acyclic graphs $G_{\mathrm{HTG}}(V_{\mathrm{HTG}}, E_{\mathrm{HTG}})$, where the vertices $V_{\mathrm{HTG}} = \{\mathrm{htg}_i \mid i = 1, 2, \ldots, n_{\mathrm{htgs}}\}$ can be one of three types $\mathrm{Type}_{\mathrm{HTG}} = \{\mathcal{SN}, \mathcal{CN}, \mathcal{LN}\}$, corresponding to single, compound, and loop nodes.

1) **Single nodes** represent nodes that have no subnodes and are used to encapsulate basic blocks.
2) **Compound nodes** are recursively defined as HTGs, i.e., they contain other HTG nodes. They are used to represent structures like if-then-else blocks, switch-case blocks or a series of HTGs.
3) **Loop nodes** are used to represent the various types of loops (for, while-do, do-while). Loop nodes consist of a loop head and a loop tail that are single nodes and a loop body that is a compound node.

The edge set E_{HTG} in G_{HTG} represents the flow of control between HTG nodes. An edge $(\mathrm{htg}_i, \mathrm{htg}_j)$ in E_{HTG}, where



```
1: t = a + b;
2: u = a – b;
3: if (a < b)
4:    v = t + c;
   else
   {
5:    w = u + c;
6:    v = w  – d;
   }
7: x = v + e;
8: y = v – e;
```

Fig. 1. (a) Example C description. (b) Corresponding HTG representation. (c) HTG representation with the control and DFGs overlaid on top. An empty basic block bb_4 is added to the CFG in the *Join* node of the If-HTG node.

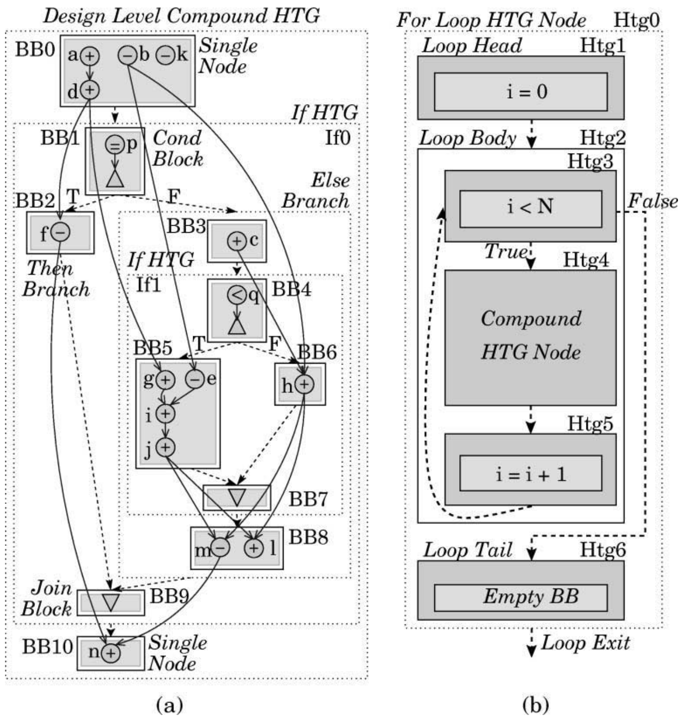


Fig. 2. (a) HTG representation of the "waka" benchmark with the control and DFGs overlaid on top. (b) HTG representation of a for-loop.

$\mathrm{htg}_i, \mathrm{htg}_j \in V_{\mathrm{HTG}}$, signifies that htg_j executes after htg_i has finished execution. Each node htg_i in V_{HTG} has two distinguished nodes, $\mathrm{htg}_{\mathrm{Start}}$ and $\mathrm{htg}_{\mathrm{Stop}}$, belonging to V_{HTG} such that there exists a path from $\mathrm{htg}_{\mathrm{Start}}$ to every node in htg_i and a path from every node in htg_i to $\mathrm{htg}_{\mathrm{Stop}}$.

The $\mathrm{htg}_{\mathrm{Start}}$ and $\mathrm{htg}_{\mathrm{Stop}}$ nodes for all compound and loop HTG nodes are always single nodes. The $\mathrm{htg}_{\mathrm{Start}}$ and $\mathrm{htg}_{\mathrm{Stop}}$ nodes of a loop HTG node are the loop head and loop tail respectively and those of a single node are the node itself. For the rest of this paper, we will denote the top-level HTG corresponding to a design as the ***Design HTG***. The design HTG is constructed by creating a compound node corresponding to each control construct in the design.

Consider the sample "C" code in Fig. 1(a). The HTG representation for this code is given in Fig. 1(b). The HTG representation consists of a compound design HTG (htg_0) that encapsulates the HTG nodes htg_1 to htg_7 and the control flow edges between the HTG nodes (shown by dashed arrows). The if-then-else control construct from the source code is encapsulated in the compound If-HTG node htg_2. The If-HTG node consists of a single node for the conditional check, a compound node for the true/then branch, a compound node for the false/else branch and a single node for the Join node (containing only an empty basic block as explained below). The $\mathrm{htg}_{\mathrm{Start}}$ node for an If-HTG is the single node with the conditional check and the $\mathrm{htg}_{\mathrm{Stop}}$ node is the Join node; in Fig. 1(b), these correspond to htg_3 and htg_6 respectively. Similarly, htg_1 and htg_7 are the $\mathrm{htg}_{\mathrm{Start}}$ and $\mathrm{htg}_{\mathrm{Stop}}$ nodes of the design HTG htg_0.

In Fig. 1(c), we show how the CFG and DFG can be overlaid onto the HTG graph. Basic blocks are denoted by shaded boxes within the HTG nodes (bb_0 to bb_5) and operations are denoted by circular nodes with the operator sign within (operations 1 to 8). Dashed lines denote control flow between HTG nodes and solid lines denote data flow between operations. A fork in the control flow (i.e., a Boolean condition check) is denoted by a triangle (Δ) and a merge by an inverted triangle (∇).

Note that, for clarity, the true and false branches in Fig. 1(c) are shown as a single node encapsulating a basic block. In practice, the single node is then encapsulated in the compound HTG node that forms the true or false branch.

Also, during the construction of HTGs, we add empty "Join" basic blocks where multiple control flow path merge. In Fig. 1(c), basic block bb_4 corresponds to a *Join* basic block that is encapsulated in the single Join node htg_6. Join HTG nodes serve as the $\mathrm{htg}_{\mathrm{Stop}}$ nodes of compound HTG nodes and enable an easier and more structured approach to the hierarchical composition of nodes. Detailed notes on HTG construction are presented in [25].

Fig. 2(a) illustrates the HTG for the synthetic benchmark "waka" [4] along with the control flow and DFGs. This design contains an If-HTG node, whose false/else branch contains another If-HTG node. bb_0 to bb_{10} denote basic blocks.

We show a For-loop HTG in Fig. 2(b). The For-loop HTG htg_0 consists of three subnodes: 1) *Loop head* (htg_1): consists of a single node with an optional initialization basic block; 2) *Loop body* (htg_2): a compound HTG node containing a single HTG node (htg_3) for the conditional check basic block and a compound HTG node (htg_4) for the main body of the loop, and an optional single node (htg_5) for the loop index increment basic block; 3) *Loop tail/exit* (htg_6): a single node with an empty basic block. There is a backward control flow edge from the end of the loop body to the conditional check single node. Maintaining the loop hierarchy allows us to treat the back edges as implicit self-loops on composite nodes [25]. Therefore, at any hierarchy level, the HTG is a directed acyclic graph.

The relationship between HTGs and CFGs is captured by the following definition.

Definition III.7: Given $G_{\mathrm{CFG}}(V_{\mathrm{CFG}}, E_{\mathrm{CFG}})$ and $G_{\mathrm{HTG}}(V_{\mathrm{HTG}}, E_{\mathrm{HTG}})$, there exists a one-to-one mapping of the basic blocks in G_{CFG} to the single nodes in G_{HTG} given by: $\mathrm{Htg}_{\mathrm{BB}} : V_{\mathrm{CFG}} \mapsto V_{\mathrm{HTG}}$. Thus, $\forall \mathrm{bb}_j \in V_{\mathrm{CFG}}$ there exists a $\mathrm{htg}_i \in V_{\mathrm{HTG}}$, such that $\mathrm{Type}_{\mathrm{HTG}}(\mathrm{htg}_i) = \mathcal{SN}$.

We capture the input description using a design graph that is defined as follows.

Definition III.8: A **design graph**, $\mathcal{DG}(V_D, E_D, \mathrm{BB}_{\mathrm{Ops}}, \mathrm{Htg}_{\mathrm{BB}})$, is a layered graph,[1] where the vertex set $V_D = \{V_{\mathrm{DFG}} \cup V_{\mathrm{CFG}} \cup V_{\mathrm{HTG}}\}$ consists of three layers of nodes corresponding to the nodes of the graphs G_{DFG}, G_{CFG}, and G_{HTG} and the edge set $E_D = \{E_{\mathrm{DFG}} \cup$

[1]A **k-layered graph** is a connected graph in which the vertices are partitioned into k sets $L = l_1, \ldots, l_k$ and edges run between the vertices of successive layers, l_i and l_{i-1}.

$E_{\mathrm{CFG}} \cup E_{\mathrm{HTG}}\}$ consists of the edges corresponding to the edges of the graphs $G_{\mathrm{DFG}}, G_{\mathrm{CFG}}$, and G_{HTG}. $\mathcal{DG}$ also contains the mapping $\mathrm{BB}_{\mathrm{Ops}}$ between the operations in G_{DFG} and the basic blocks in G_{CFG} and the mapping $\mathrm{Htg}_{\mathrm{BB}}$ between the basic blocks in G_{CFG} and the single nodes in G_{HTG}.

For clarity, in the rest of this paper, we make several simplifications in the figures used for the examples. We omit the single HTG node that encapsulates basic blocks. Control flow edges in HTG representations are shown to originate from basic blocks and terminate at basic blocks (i.e., these represent the edges from the CFG).

IV. Code Motions in High-Level Synthesis

Computationally expensive portions of several classes of applications—particularly multimedia and image processing applications—are characterized by the presence of a considerable number of unpredictable branches. These control constructs limit the amount of instruction-level parallelism that can be exploited from the input description [30], [31]. There are usually not enough operations available for execution to utilize all the resources in each cycle or scheduling step. Hence, there are a number of idle resources in a basic block.

A resource is said to be *idle* in a scheduling step if there is no operation scheduled to execute on that resource in that scheduling step (the converse of an idle resource is a *busy* resource). Idle resources can be utilized by moving and scheduling operations from subsequent or preceding basic blocks. The candidate operations for these *code motions* are operations whose data dependencies are satisfied, but the conditions under which they execute may not have been evaluated. One of the key enabling transformations for such type of code motions is speculation.

A. Using Speculation in High-Level Synthesis

Speculative execution or speculation refers to the execution of an operation before the branch condition that controls the operation has been evaluated.[2] In our approach to speculation for high-level synthesis, we store the result of a speculated operation in a new register. If the condition that the operation was to execute under evaluates to true, then the stored result is committed to the variable from the original operation, else the stored result is discarded.

We demonstrate speculation by the example in Fig. 3. In Fig. 3(a), variables d and g are calculated based on the result of the calculation of the conditional c. Since the operations that produce d and g execute on different branches of a conditional block, these operations are *mutually exclusive*. Hence, these operations can be scheduled on the same hardware resource with appropriate multiplexing of the inputs and outputs, as shown by the circuit in Fig. 3(a).

Now, consider that an additional adder is available. Then, the operations within the conditional branches can be calculated *speculatively* and concurrently with the calculation of the conditional c, as shown in Fig. 3(b). The corresponding hardware circuit is also shown in this figure. Based on the evaluation of the conditional, one of the results will be discarded and the other committed. It is evident from the corresponding hardware circuits in Fig. 3(a) and (b) that as a result of this speculation, the longest path gets shortened from being a sequential chain of a comparison followed by an addition to being a parallel computation of the comparison and the additions.

This example also demonstrates the additional costs of speculation. Speculation requires more functional units and potentially more storage for the intermediate results. Uncontrolled aggressive speculation can also lead to *worse* results due to multiplexing and control overheads. On the other hand, judicious use of speculation can improve resource utilization.

[2]*Data speculation* is another type of speculation in which an operation is executed with potentially incorrect operand values.

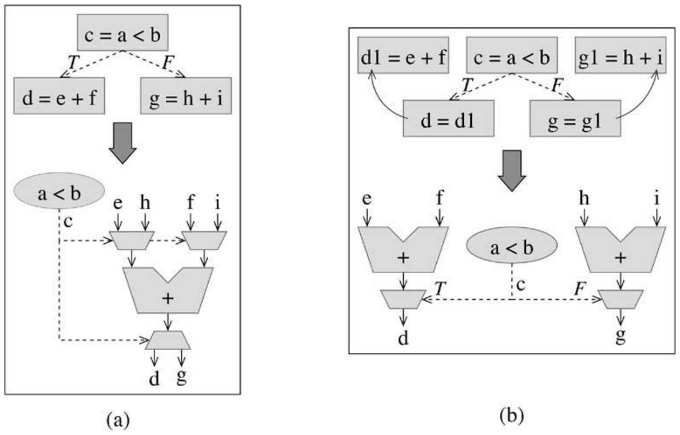


Fig. 3. Extracting the inherent parallelism in a design by speculating the addition operations. This requires an additional resource, but leads to a reduction in the longest path.

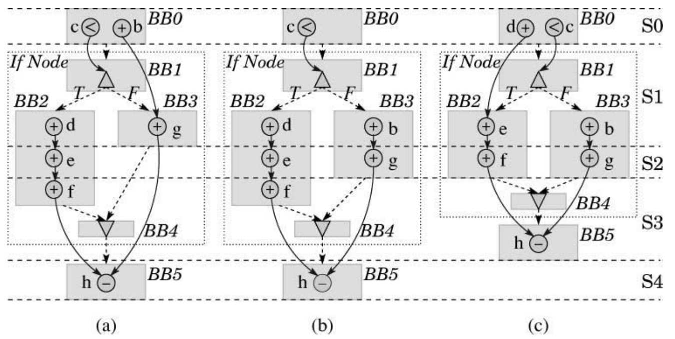


Fig. 4. Reverse Speculation. (a) An example design. (b) Operation b is reverse speculated into the branch of the If-HTG that uses its result, i.e., the *false* branch. (c) Operation d is now speculated into basic block bb_0. This reduces the schedule length by one cycle.

B. Reverse Speculation

Reverse speculation refers to moving an operation op_i from its basic block bb_j into the successors of bb_j. We employ this code motion to duplicate operation op_i into the branches of an If-HTG when the If-HTG is the successor of bb_j. Reverse speculation has been referred to as *lazy* execution [20] and *duplicating down* in past literature [32].

Reverse speculation is useful in instances where an operation inside a branch of an If-HTG is on the longest path through the design, whereas an operation before the If-HTG is not. We demonstrate this by an example in Fig. 4(a). In this design, operation b, that is on the shorter dependency path ($\langle b, g, h\rangle$), is placed in the basic block before the If-HTG, whereas operation d, that is on the longer dependency path ($\langle d, e, f, h\rangle$), is placed in the *true* branch of the If-HTG. If we reverse speculate operation b into the conditional branches, as shown in Fig. 4(b), the adder in basic block bb_0 is left idle (see next paragraph for an explanation of why b is not duplicated). This enables us to speculatively execute operation d in bb_0, as shown in Fig. 4(c). The dashed lines in Fig. 4 demarcate the state assignments ($S0$ through $S4$) for the three designs. Clearly, the final design in Fig. 4(c), after reverse speculation of b and speculation of d, requires one state less than the original design in Fig. 4(a).

Note that, while applying reverse speculation in the example above, a data dependency analysis determines that the result of operation b is used only in the *false* branch of the If-HTG. Hence, instead of duplicating b into both branches, we move b only into the false branch of the If-HTG, as shown in Fig. 4(b). In the general case, reverse speculation leads to duplication of the operation into both the branches of

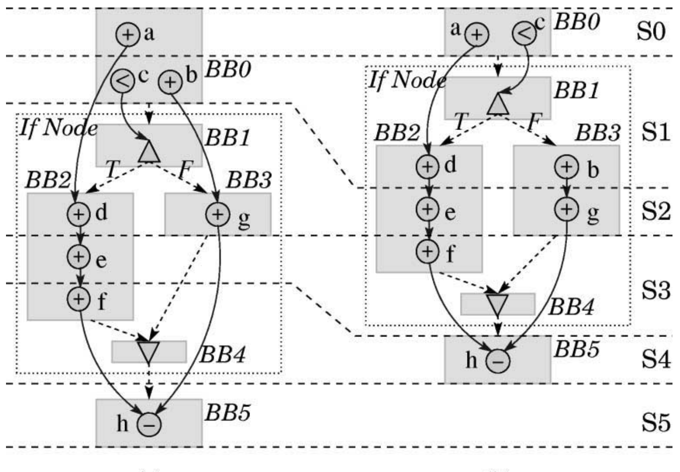


Fig. 5. Early condition execution. (a) Original design. (b) Comparison operation c is scheduled as soon as possible to evaluate the conditional check early. All unscheduled operations before the conditional checks are reverse speculated into the branches of the If-HTG.

an If-HTG. It is also important to make a distinction between moving operations into a later scheduling step and the downward operation *duplication* done by reverse speculation. When an operation encounters a fork node while being moved down, it has to be duplicated into all the control paths that lead out of the fork node (unless its result is not needed in one of the branches).

C. Early Condition Execution

We employ reverse speculation by using another novel transformation called *early condition execution*. This transformation attempts to schedule operations such that the conditional check can be evaluated or scheduled as soon as possible. Any operations before the conditional check that are unscheduled are moved into the branches of the If-HTG by reverse speculation. Evaluating a conditional check early using early condition execution resolves the control dependency for operations within branches of the If-HTG. These operations are, thereby, available for scheduling sooner.

Early condition execution is demonstrated by an example in Fig. 5(a). In this example, comparison operation c computes a conditional that is checked in basic block bb_1 (the Boolean conditional check is denoted by a triangle). We can schedule this comparison operation concurrently with operation a in state $S0$ in basic block bb_0, as shown in Fig. 5(b). Now, the conditional check in basic block bb_1 can be executed "early" in state $S1$. However, operation b in basic block bb_0 has not been scheduled as of yet. Therefore, this operation is reverse speculated into basic block bb_3 (and not into bb_2 since its result is used only in bb_3). These code motions lead to an overall shorter schedule length, as shown by the state assignments in Figs. 5(a) and (b).

D. Conditional Speculation

Often design descriptions have instances where there are idle resources in the scheduling steps of the basic blocks that comprise the branches of an If-HTG. Speculating out of If-HTGs also leaves resources idle in the basic blocks of the conditional branches. To utilize these idle resources, we propose *duplicating* operations that lie in basic blocks after the conditional branches up into the basic blocks that comprise the conditional branches. We call this code motion *conditional speculation*. This is similar to the duplication-up code motion used in compilers and the node duplication transformation discussed by Wakabayashi *et al.* [4].

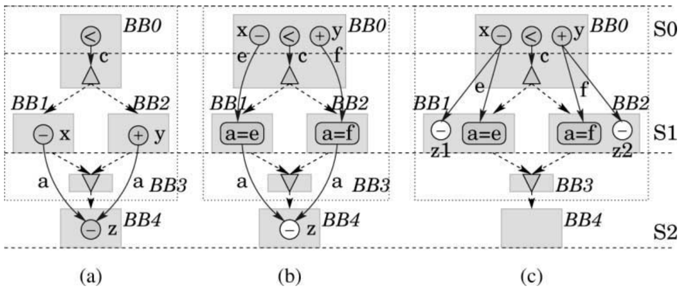


Fig. 6. (a) HTG representation of an example. (b) Operations x and y are speculated leaving resources idle in the conditional branches. (c) Operation z is CS into the conditional branches (bb_1 and bb_2).

We demonstrate conditional speculation by the example in Fig. 6(a). In this example, operations x and y both write to the variable a in the conditional branches bb_1 and bb_2. Consider that this design is allocated one adder, one subtracter, and one comparator. Then, operations x and y can be speculatively executed as shown in Fig. 6(b). The speculation of these operations leaves the resources in basic blocks bb_1 and bb_2 idle. Hence, the operation z that lies in basic block bb_4 can be duplicated up or *conditionally speculated* (CS) into both branches of the If-HTG and scheduled on the idle adder, as illustrated in Fig. 6(c). Operation z is dependent on either the result of operation x or operation y, depending on how the condition evaluates (since operation z is dependent on the variable a). Hence, the duplicated operations $z1$ and $z2$ directly read the results of operations x and y, respectively. We have also shown the state assignments ($S0, S1$, and $S2$) for the three designs using dashed lines in Fig. 6. Clearly, for this example, this set of code motions leads to a design that requires one less state to execute.

Note that, correctness issues place a number of constraints on the kind of code motions that can be done. We have omitted these for brevity, but they are detailed in [33] and are also dealt with to some extent in [11] and [25].

V. *SPARK* HIGH-LEVEL SYNTHESIS FRAMEWORK

We have implemented the various speculative code motions in a modular and extensible high-level synthesis framework called *Spark*. *Spark* provides a range of coarse-grain and fine-grain transformations and has been designed to aid in experimenting with new transformations and heuristics that optimize the quality of synthesis results, in terms of circuit delay and area. Fig. 7 provides an overview of the *Spark* framework. As shown in this figure, besides the input description, *Spark* also takes additional information as input, such as a hardware resource library, resource and timing constraints, and user directives for the various heuristics and transformations.

The design flow through the *Spark* framework is as follows. *Spark* accepts a behavioral description of a design in ANSI-C, creates the intermediate representation that comprises of HTGs, CFGs, and DFGs, runs a data dependency analysis pass, schedules the design, binds the resources, performs control synthesis, and, finally generates an output in RTL VHDL.

The core of the synthesis framework has a *transformations toolbox* that consists of a data-dependency analysis pass, the *Trailblazing* parallelizing code motion technique [11], dynamic renaming of variables, the basic operations of loop pipelining (or software pipelining), and supporting compiler passes such as constant propagation and dead code elimination. Passes from the toolbox are called by a set of heuristics that guide how the code refinement takes place. Since the heuristics and the underlying transformations that they use are fairly independent, writing new heuristics can be as simple as making calls to the toolbox. The use of transformations can be controlled by the designer

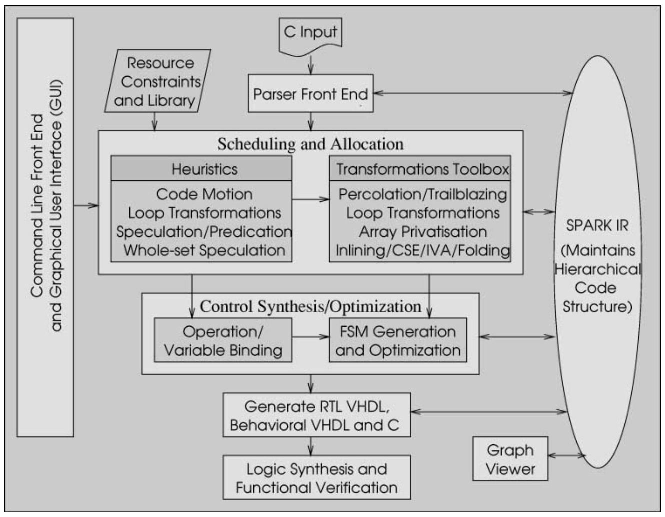

Command Line Front End and Graphical User Interface (GUI)
Resource Constraints and Library
C Input
Parser Front End
Scheduling and Allocation
Heuristics
Code Motion
Loop Transformations
Speculation/Predication
Whole-set Speculation
Transformations Toolbox
Percolation/Trailblazing
Loop Transformations
Array Privatisation
Inlining/CSE/IVA/Folding
SPARK IR (Maintains Hierarchical Code Structure)
Control Synthesis/Optimization
Operation/ Variable Binding
FSM Generation and Optimization
Generate RTL VHDL, Behavioral VHDL and C
Logic Synthesis and Functional Verification
Graph Viewer

Fig. 7. *Spark* high level synthesis framework. A complete synthesis system that provides a path from an architectural description in "C" to synthesizable RTL VHDL.

using scripts, hence, enabling experimentation with different transformations and heuristics.

A scheduling heuristic employs the techniques from the transformations toolbox to create a scheduled design (described in Section VI). The scheduling phase is followed by a *resource-binding and control-synthesis* phase. This phase binds operations to functional units, ties the functional units together (interconnect binding), allocates and binds storage (registers), generates the steering logic, and generates the control circuits to implement the schedule. The focus of our resource-binding approach is to minimize the interconnect (multiplexers and demultiplexers) between functional units and registers [9]. After binding, we generate a finite-state machine (FSM) controller for the scheduled and bound design.

The back-end of the *Spark* framework consists of an RTL VHDL generator. This RTL VHDL belongs to the subset of VHDL that is synthesizable by commercial logic synthesis tools and, hence, the *Spark* framework integrates into the standard synthesis design flow. This completes the path from architectural design and specification in a high level language such as "C" to synthesizable RTL VHDL code and then down to the synthesized net-list.

A. Trailblazing: Hierarchical Code-Motion Technique

The speculative code motions employed by *Spark* are enabled by the *Trailblazing* code motion technique [11]. Trailblazing exploits information about the hierarchical structure of the design that is maintained by HTGs [25] to perform efficient code motions across large pieces of code. The Trailblazing algorithm returns a set of trails or control flow paths that an operation will have to follow in order to move from one basic block to another. To move the operation along these control flow paths, the Trailblazing algorithm employs the use of the speculative code motions presented earlier in this paper. Details about the Trailblazing algorithm are discussed in [11].

To understand the hierarchical moves performed by Trailblazing, consider the example in Fig. 8. In this example, we want to move the operation Op: $y = e + f$ from basic block bb_2 to basic block bb_1. While moving this operation, Trailblazing encounters the $\mathrm{htg}_{\mathrm{Stop}}$ node of an If-HTG node. It checks if the moving operation has any dependencies with the If-HTG node. Since, in this example, there are no dependencies, operation Op is moved across the If-HTG node to bb_1, without visiting each subnode of the If-HTG, as shown in Fig. 8(b).

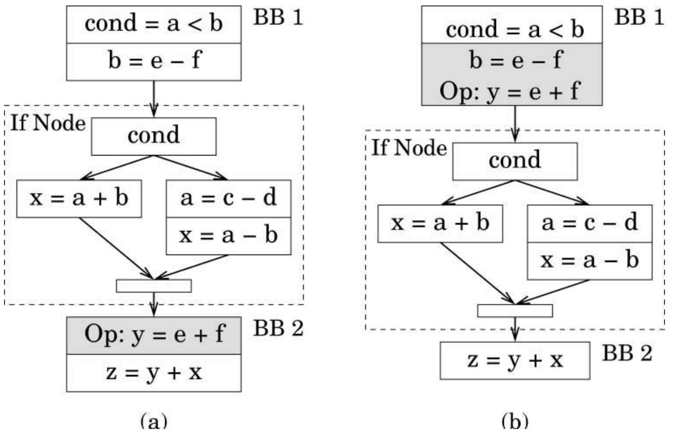
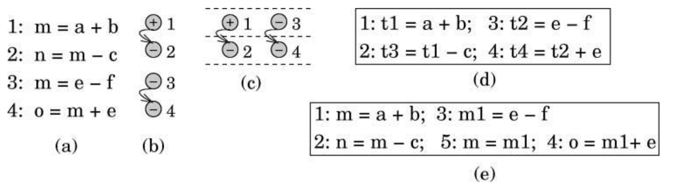


Fig. 8. Trailblazing. Operation $op1$ is moved from basic block bb_2 to basic block bb_1 across the if-then-else HTG node without visiting each basic block inside the node.


```
1: m = a + b
2: n = m – c
3: m = e – f
4: o = m + e
```
(a)

```
1: t1 = a + b;  3: t2 = e – f
2: t3 = t1 – c; 4: t4 = t2 + e
```
(d)

```
1: m = a + b;  3: m1 = e – f
2: n = m – c;  5: m = m1; 4: o = m1+ e
```
(e)

Fig. 9. (a) Example C description. (b) Corresponding DFG. (c) DFG after scheduling: Ops 1 and 3 and ops 2 and 4 are scheduled concurrently. (d) Scheduled output code generated if only flow data dependencies are maintained. (e) Output code generated if nonflow data dependencies are also maintained. Concurrent operations are written in the same line.

B. Eliminating Data Dependencies by Dynamic Renaming

There are four types of data dependencies that can exist between operations [32]: 1) *flow* (variable read after write); 2) *anti* (write after read); 3) *output* (write after write); and 4) *input* (read after read). High-level synthesis approaches have traditionally chosen to retain only flow data dependencies. However, this means that the variable names from the original description are discarded, thereby, impairing the ability to correlate the input description with the intermediate representation and the final output code generated after synthesis. This makes it difficult to visualize the effects of applying the various transformations.

To understand the need for maintaining nonflow data dependencies, consider the sample "C" description in Fig. 9(a) and its corresponding DFG in Fig. 9(b). One possible schedule is shown in Fig. 9(c), where operations 1 and 3 and operations 2, and 4 are scheduled concurrently. The output code corresponding to this scheduled design, when nonflow data dependencies are not maintained, is given in Fig. 9(d). In this output code, we have to create new variables that store the result of each operation in the scheduled design. Clearly, it takes some effort to correlate the operation statements in this output code with the operation statements in the input code (especially operations 2 and 4).

If we maintain nonflow data dependencies as well, we can generate the output code given in Fig. 9(e). In this code, the variables that each operation writes to are maintained as per the original code. By inspecting this code, we see that concurrent execution of operations 1 and 3 requires renaming the result variable of operation 3 to $m1$. Operation 5 is inserted in the code to copy the new variable $m1$ to the variable m from the original code, as shown in Fig. 9(e). Thus, operation 4 can be executed concurrently with operations 2 and 5 using the new variable $m1$ after employing dynamic variable renaming.

Although nonflow-data dependencies can restrict code motions, these can often be resolved by dynamic renaming and combining

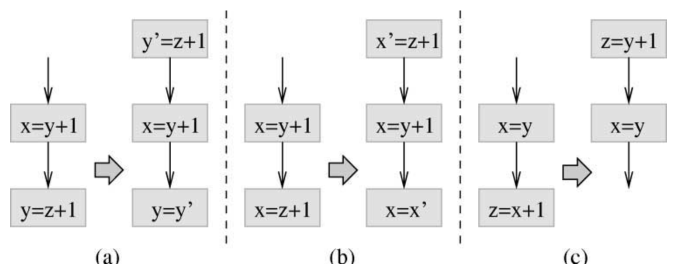


Fig. 10. Moving an operation across another operation while eliminating (a) an antidependency, (b) an output dependency, and (c) a flow dependency.

[34]. Fig. 10(a)–(c) demonstrate how one operation can be moved past another one while dynamically eliminating data dependencies. In Fig. 10(a), the operation that writes to variable y is scheduled at an earlier time step by moving only the right-hand side of the operation. The result is written to a new destination variable y' and the original operation is replaced by a copy operation from y' to y. Similarly, in Fig. 10(b), an output dependency between two operations that write to the same variable x can be resolved by creating a new destination variable x', while moving the operation and replacing the original operation with a copy operation. These copy operations can also be circumvented by a technique known as *combining*. Combining replaces the copy in the operation being moved by the variable being copied. This is demonstrated in Fig. 10(c), where the operation $z = x + 1$ is moved past the copy operation $x = y$. The variable x is replaced with the variable y on the right-hand side of the moving operation.

VI. Priority-Based Global List Scheduling Heuristic

Scheduling is the task of assigning operations to control steps or time intervals, while respecting data dependencies and resource constraints (if any) [15]. For the purpose of evaluating the various code motion transformations, we have chosen a *priority-based* global list scheduling heuristic, although the transformations presented here can be employed by other scheduling heuristics as well. Priority-list scheduling works by ordering and scheduling operations based on a *priority* or cost associated with them.

Our objective is to minimize the *longest delay* through the design. Hence, the *priority* of an operation is determined by the length of the data-dependency chain from the operation to the outputs of the design. The priority of an operation is calculated as the delay of the resource that the operation can be mapped to summed with the maximum of the priorities of all the operations that use its result. A priority of zero is assigned to operations that produce outputs. A priority of one (or two, if the output operations are multiplications) is assigned to operations whose results the output operations depend on and so on. The priority of an operation that creates a conditional check is assigned the maximum of the priorities of all the operations in the branches of the If-HTG.

Our scheduling heuristic is presented in Fig. 11. The inputs to this heuristic are the unscheduled design graph $\mathcal{DG}$ and the list of resources $\mathcal{R}$. Additionally, we can specify a list of allowed code motions, CMs, (i.e., speculation, reverse speculation, conditional speculation, etc.). This gives us control over the code motions employed, while scheduling the design by selecting and deselecting code motions from CMs. We can, thus, analyze the performance-area tradeoffs of individual code motions.

The heuristic starts by assigning a priority to each operation in the input description as explained above. Scheduling is done one scheduling step at a time while traversing the basic blocks in the design CFG G_{CFG}, starting with the first step in the first basic block of G_{CFG} (line 2 of the algorithm). For each scheduling step (step_k) in the basic block, the scheduler iterates over each resource res in the resource list $\mathcal{R}$ and calls the *GetAvailableOps* function. This function returns a list of *available* operations $\mathcal{A}$ that may be scheduled on res in step_k. (lines 4 and 5 in the algorithm in Fig. 11).

```
/* Schedules the Design Graph $\mathcal{DG}$ */
$PriorityListScheduling(\mathcal{DG}, \mathcal{R}, CMs)$
1: $Pr \leftarrow$ CalculatePriority( $G_{DFG}$ )
2: $step_k \leftarrow$ First scheduling step in FirstBB($G_{CFG}$)
3: while ($step_k \neq \phi$) {
4:     foreach (resource $res \in \mathcal{R}$) {
5:         $\mathcal{A} \leftarrow$ GetAvailableOps($\mathcal{DG}$, $step_k$, $res$, $CMs$)
6:         if ($\mathcal{A} \neq \phi$) {
7:             Pick Operation $op \in \mathcal{A}$ with lowest cost
8:             TrailblazeOp($op$, $res$, $step_k$, $\mathcal{DG}$, $CMs$)
9:         }
10:    }     /* end foreach */
11:    $step_k \leftarrow$ GetNextSchedulingStep($G_{HTG}$, $G_{CFG}$, $step_k$)
12: }     /* end while */
```

Fig. 11. Priority-based list scheduling Heuristic.

```
/* Gets ops for scheduling on $res$ in $step_k$ */
$GetAvailableOps(\mathcal{DG}, step_k, res, CMs)$
Returns: Available Operations List $\mathcal{A}$
1: $\mathcal{A} \leftarrow$ Unscheduled operations in $G_{CFG}$
       that can be scheduled on $res$
2: foreach ($op_i \in \mathcal{A}$) {
3:     if (data dependencies of $op_i$ cannot be satisfied)
4:         $\mathcal{A} \leftarrow \mathcal{A}$ - $op_i$
5:     if (MoveIsNotPossible($op_i$, $step_k$, $CMs$)
6:         $\mathcal{A} \leftarrow \mathcal{A}$ - $op_i$
7: }
8: Calculate cost of operations in $\mathcal{A}$
9: return $\mathcal{A}$
```

Fig. 12. Algorithm to determine the list of *Available* operations.

Available operations is a list of operations whose data dependencies are satisfied and that can be moved in the design graph and scheduled on the given resource at the current scheduling step. Pseudocode for collecting the list of available operations is given in Fig. 12. Initially, all unscheduled operations in the design that can be executed on the resource type of res are added to the available operations list. The unscheduled operations are collected by traversing the basic blocks on the control flow paths from the current basic block being scheduled. However, this basic block traversal algorithm skips over the loop body of any loop nodes it encounters. This is because operations from within loop nodes can only be moved outside the loop body by transformations such as loop-invariant code motion and loop pipelining. Operations whose data dependencies are not satisfied and cannot be satisfied by variable renaming are also removed from this list (lines 3 and 4 in Fig. 12).

The available operations algorithm then calls the *MoveIsNotPossible* function (presented in [33]). This function employs *Trailblazing* to determine all the code motions required to move the operation op_i from its current position to the scheduling step step_k under consideration. Operations that require code motions that have not been enabled in the list of allowed code motions CMs are removed from the available list (lines 5 and 6 of the algorithm).

Finally, the available operations algorithm assigns a cost for each remaining operation in $\mathcal{A}$. Currently, this cost is the negative of the operation's priority, i.e.,

$$\mathrm{cost}(\mathrm{op}_i) = -\,\mathrm{Pr}(\mathrm{op}_i) \quad \forall \mathrm{op}_i \in G_{\mathrm{DFG}}.$$

Since the scheduler picks the operation op with the *lowest* cost from the available operations list (line 7 of Fig. 11), this effectively means that the operation with the highest priority in the available list is chosen. Hence, operations that are on the longest path through the design are favored for scheduling. In this way, the cost function attempts to minimize the longest delay through the design.

It is important to note that minimizing a different cost function such as average delay can be done by incorporating control-flow information into the cost function. Also, if we have profiling information about which control paths are more likely to be taken, then we can give operations on those paths a higher priority than operations on less taken paths. Future work entails enhancing the cost function to include hardware (control, interconnect, and area) cost models of the code transformations.

Next, the scheduling heuristic calls *Trailblazing* to move the chosen operation op to the current scheduling step step_k (line 8 of algorithm in Fig. 11). This scheduling procedure is repeated for all resources in each scheduling step as the basic blocks in the design are traversed from top to bottom by calling function *GetNextSchedulingStep* (presented in [35]). This function traverses the design using topological ordering. Thus, when a fork node of an If-HTG is encountered, the branches of the If-HTG are traversed (scheduled) first and then, the basic blocks past the join node of the If-HTG are traversed [33].

Note that, since operations with higher priorities may be speculated into a basic block, the (lower priority) operations that were originally placed in that basic block by the designer may be left unscheduled. Either new scheduling steps are added to the current basic block to schedule them or if reverse speculation has been enabled, then these operations are reverse speculated into the branches of the subsequent If-HTG.

A. Scheduling Loops

Scheduling of loops is done by the same procedure outlined above. However, user-specified loop transformations such as loop unrolling are applied first. Also, the scheduler cannot move operations into or out of the loop body. This can only be done by transformations such as loop-invariant code motion or loop pipelining. Hence, the available operations algorithm does not collect unscheduled operations from inside a loop body to schedule them outside the loop body. Also, while scheduling the loop body of a loop node, available operations are collected only from within the loop body.

The *Spark* framework can schedule all types of loops, including those with unknown loop iteration bounds. Our FSM is generated such that at the end of a loop body, the next state is either the first state in the loop body or the state after the loop body, depending on whether the loop condition is satisfied or not. Hence, loop bounds are not required for generating correct, synthesizable VHDL. However, when the loop bounds are unknown, several loop transformations cannot be applied to the design and we cannot establish the number of cycles that the loop takes to execute.

B. Heuristic to Determine Whether to Apply Conditional Speculation

Experimental results have shown us that conditional speculation, when applied unchecked, can lead to increased schedule lengths and interconnect complexity, due to operation duplication. Hence, we have developed a heuristic that determines if an operation op should be CS. This heuristic *AllowConditionalSpec*, is outlined in Fig. 13. The heuristic starts with the list of basic blocks (BBList) into which an operation op will be duplicated, if it is scheduled in basic block $\mathrm{bb}_{\mathrm{curr}}$. The heuristic returns a true result if it is possible to conditionally speculate op in each basic block in BBList and a false result otherwise.

```
AllowConditionalSpec(op, BBList, $bb_{curr}$)
Returns: True if op should be conditionally speculated
1: foreach ($bb_i$ in $BBList$) {
2:     if ($bb_i$ is not scheduled)
3:         return $false$
4:     if (FindIdleResInBB($bb_i$, op) = $\phi$) {
       /* Check if we can insert a new step in $bb_i$ */
5:         if (NumSteps($bb_i$) $\geq$ NumSteps($bb_{curr}$))
6:             return $false$    /* Cannot insert op in $bb_i$ */
7:     }
8: }
9: return $true$
```

Fig. 13. Heuristic to determine whether to schedule operation op into basic block $\mathrm{bb}_{\mathrm{curr}}$ by conditionally speculating (duplicating) into the basic blocks in BBList.

The heuristic iterates over each basic block bb_i in the list BBList and returns a false result if any $\mathrm{bb}_i \in \mathrm{BBList}$ is unscheduled. This is because, without scheduling a basic block bb_i first, it is not possible to accurately determine if there is an idle resource in bb_i on which to schedule operation op. If bb_i is scheduled, the *AllowConditionalSpec* heuristic calls the *FindIdleResInBB* (presented in [36]) to find an idle resource on which operation op can be scheduled (line 4 in Fig. 13).

If there is no idle resource in bb_i, then the heuristic checks if it is possible to create a new scheduling step in bb_i in which op can be scheduled. We can add a new scheduling step in bb_i when the number of scheduling steps in bb_i is less than the number of scheduling steps in $\mathrm{bb}_{\mathrm{curr}}$. This check prevents basic block bb_i from becoming the critical path in the design with the most number of cycles (or scheduling steps) among its mutually exclusive control paths. Thus, this check ensures that the length of the longest path (path with the maximum number of scheduling steps or cycles) through the If-HTG does not increase by applying conditional speculation. The *AllowConditionalSpec* heuristic returns a true result if it either finds an idle resource or if it is possible to add a new scheduling in each $\mathrm{bb}_i \in BBList$ (line 9 of the algorithm).

VII. Experimental Setup and Results

We used four designs as case studies to study the impact of the proposed code motions. First on the scheduling and controller size results, and then on the logic synthesis results. These four designs are derived from three moderately complex real-life applications representative of the multimedia and image processing domains. The designs are: the $pred1$ and $pred2$ functions from the *Prediction* block of the MPEG-1 algorithm, the *dpframe_estimate* function from the *Motion Estimation* block of the MPEG-2 encoder algorithm [37], and the tile function[3] (with the scale function inlined) from the "tiler" transform of the GIMP image processing tool [38]. Results for more designs are presented in [33].

Table I lists the characteristics of the four designs in terms of the number of if-then-else conditional blocks (If HTGs), loops (Loop HTGs), nonempty basic blocks, and the total number of operations in the input description. The number of If HTGs, Loop HTGs, and basic blocks is indicative of the control complexity of the design. All these designs have doubly nested loops. Typical runtimes of *Spark* to produce the results for these designs are in the range of five user seconds on a 1.6-GHz Linux PC.

Table I also lists the type and quantity of the resources used for scheduling these designs: $+-$ does add and subtract, $==$ is a comparator, $*$ a multiplier, / a divider, [] an array address decoder, and $\ll$

[3]Note that this floating point function has been arbitrarily converted to an integer function for the purpose of our experiments. This does not affect the nature of the data and control flow, but only the data that is processed.

TABLE I
CHARACTERISTICS OF THE FOUR DESIGNS USED IN OUR EXPERIMENTS, ALONG WITH THE RESOURCES ALLOCATED FOR SCHEDULING THEM

Benchmark	# Ifs	# Lps	# BBs	# Ops	# Resources $+-$	$*$	$/$	$<<$	$==$	$[\,]$
$Mpeg$-1 $pred1$	4	2	17	123	2	1	-	2	2	2
$Mpeg$-1 $pred2$	11	6	45	287	2	1	-	2	2	2
$Mpeg$-2 $dpframe$	18	4	61	260	4	1	-	2	2	2
$GIMP$ $tiler$	11	2	35	150	3	1	1	2	2	2

TABLE II
SCHEDULING RESULTS AFTER APPLYING THE SPECULATIVE CODE THE MPEG-1 *PRED*1 AND *PRED*2 FUNCTIONS

Allowed	$MPEG$-1 $pred1$		$MPEG$-1 $pred2$	
Code Motions	# States	Long Path	# States	Long Path
Within BBs	71	2009	125	4555
+across HTGs	60(-14.3 %)	1937(-3.6 %)	111(-11 %)	4409(-3.2 %)
+early cond exec	61(+1.7 %)	1937(0 %)	113(-1.8 %)	4409(0 %)
+speculation	56(-8.2 %)	1862(-3.9 %)	104(-8.0 %)	4178(-5.2 %)
+cond spec	40(-28.6 %)	1091(-41 %)	74(-28.8 %)	2575(-38.4%)
Total Reduction	**42.9 %**	**45.7 %**	**40.8 %**	**43.5 %**

TABLE III
SCHEDULING RESULTS AFTER APPLYING THE SPECULATIVE CODE THE *DPFRAME_ESTIMATE* AND *TILER* DESIGNS

Allowed	$MPEG$-2 $dpframe$		$GIMP$ $tiler$	
Code Motions	# States	Long Path	# States	Long Path
Within BBs	77	911	69	6144
+across HTGs	72(-6.5 %)	863(-5.3 %)	66(-4.3 %)	5944(-3.3 %)
+early cond exec	71(-1.4 %)	859(-0.5 %)	63(-4.5 %)	5544(-6.7 %)
+speculation	58(-18.3 %)	607(-29.3 %)	54(-14.3 %)	4734(-14.6 %)
+cond spec	49(-15.5 %)	571(-5.9 %)	31(-42.6 %)	2534(-46.5 %)
Total Reduction	**36.4 %**	**37.3 %**	**55.1 %**	**58.8 %**

is a shifter. The multiplier $(*)$ executes in two cycles and the divider $(/)$ in five cycles. All other resources are single cycle.

A. Effects on Performance and Controller Size

Tables II and III list the scheduling results for the four designs as each code motion is enabled. These scheduling results are in terms of the number of states in the FSM controller (denotes controller complexity) and the cycles on the longest path in the design (i.e., execution cycles). The longest path through an if-then-else construct is the number of scheduling steps through the longest branch. For loops, the longest path length of the loop body is multiplied by the number of loop iterations (loop bounds are known for all four designs).

The rows in Tables II and III present results with each code motion enabled incrementally, i.e., these signify the "allowed code motions," while determining the available operations (see Section VI) and do *not* represent an ordering of code motions. We first allow code motions only within basic blocks (first row) and then, then across hierarchical blocks as well, i.e., across entire if-then-else conditionals and loops (second row), then with reverse speculation and early condition execution also enabled (third row), then with speculation (fourth row), and, finally, with conditional speculation enabled as well (fifth row). The percentage reductions of each row over the previous row are given in parentheses.

The results in Tables II and III demonstrate that enabling just the non-speculative code motions across hierarchical blocks of code (second row) leads to modest improvements: ranging from 4% to 14% in the number of FSM states and 3 to 5% in the cycles on the longest path. Early condition execution (that employs reverse speculation) also leads to little or no improvements. (see third row in Tables II and III).

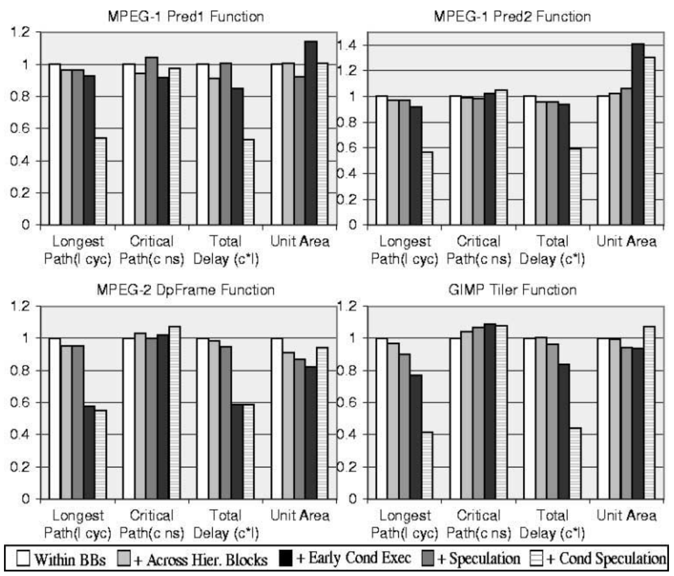


Fig. 14. Effects of speculative code motions on logic synthesis results for *PRED*1, *PRED*2, *DPFRAME_ESTIMATE*, and *TILER* designs.

The largest improvements in performance (cycles) are obtained by employing speculation and conditional speculation. Speculation reduces the number of states and the number of cycles by 8 to 18% and 3 to 29% respectively. Conditional speculation leads to even more impressive improvements of 15 to 42% in the number of states and 5 to 46% in the number of cycles, as shown in the fifth row of Tables II and III.

The total reductions for the two metrics with all the code motions enabled over code motions only within basic blocks are given in the last row of Tables II and III. These improvements are in the range of 36 to 55% and 37 to 58% in the number of states and cycles respectively. These substantial gains demonstrate that speculative code motions are essential in designs with moderate control-flow. We also find that to be truly effective all the code motions must be applied together, thus, giving the scheduler the maximum freedom to choose from among the candidate operations. Note that, when code motions only within basic blocks are enabled, our global list scheduling heuristic reduces to the classical list scheduling approaches presented for data flow only designs [15], [16].

B. Effects on Area and Clock Period

In order to gauge the control and interconnect costs of the code motions, we synthesized the RTL VHDL generated by *Spark* after scheduling and resource binding using the Synopsys *Design Compiler* logic synthesis tool. We used the LSI-10K synthesis library for technology mapping.

The logic synthesis results are summarized in the graphs in Fig. 14 for the four designs. We map three metrics in these graphs: 1) the critical path length (in nanoseconds); 2) the unit area (in terms of synthesis library used); and 3) the maximum delay through the design. The *critical path length* is the length of the longest combinational path in the netlist as reported by the static timing analysis tool. The critical path length dictates the clock period of the final design. The maximum delay is the product of the longest path length (in cycles) and the critical path length (in nanoseconds) and signifies the maximum input to output latency of the design.

The bars in the graphs in Fig. 14 present results for code motions within basic blocks only (first bar), for code motions across hierarchical blocks enabled as well (second bar), early condition execution enabled also (third bar), speculation enabled as well (fourth bar), and finally

TABLE IV
COMPARISONS WITH PREVIOUS WORK USING CLASSICAL HIGH-LEVEL SYNTHESIS BENCHMARKS. NA{ }=RESULTS ARE NOT AVAILABLE

Benchmark	# BBs	Resources	Schedule Length				
			CVLS [1]	HRA [2]	Radiv [3]	Santos [4]	*Spark* Ours
kim [2]	7	2+,1-,2==	6	7	6	6	6
parker[5]	20	2+,3-,5==	4	NA	4	4	4
waka [1]	9	1+,1-,2==	7	7	7	7	7
rotor [3]	11	2+-,2*,1[ ]	NA	NA	8	8	8

with conditional speculation enabled as well (fifth bar). Thus, the fifth bar represents the results with all the speculative and nonspeculative code motions enabled and the second bar represents the results when only the nonspeculative code motions are enabled.

The graphs in Fig. 14 show that the total delay is almost halved when all the code motions are enabled over when code motions only within basic blocks are allowed. The results in these graphs also show that the critical path lengths remain fairly constant as the code motions are enabled. This means that the clock period does not increase by applying these code motions. The constant critical path length, coupled with large decreases in cycles on the longest path, leads to large decreases in the total delay through the circuit.

However, code motions such as speculation and conditional speculation can lead to an increase in area, as we can see from graphs in Fig. 14. This area increase is due to the increasing complexity of the interconnect (multiplexers and associated control logic) that is a product of the shorter schedule lengths produced by the speculative code motions. Shorter schedule lengths mean that resource utilization and resource sharing increases and this leads to an increase in the complexity of the multiplexers and associated control logic. This complexity increase is particularly large due to conditional speculation because it duplicates operations and, thus, more operations are mapped to the same number of resources as before.

Note that critical path lengths remain fairly constant despite the increases in interconnect complexity, since these increases are counterbalanced by decreases in the controller size. We found that critical paths in our designs typically start in the control logic that generates the select signals for the multiplexers, continue through the multiplexers, the functional unit, a demultiplexer, and finally write to an output register. Even though the code motions increase the size and complexity of the multiplexers and demultiplexers, they also lead to fewer states in the FSM controller. Thus, the size and complexity of the controller decreases.

We also keep area and critical path overheads in check by employing an interconnect minimizing resource binding methodology [9] that aids the logic synthesis tool to optimize away a lot of the complexity of the interconnect. The area may still increase sometimes since we direct the logic synthesis tool to sacrifice area in a bid to achieve shorter critical path lengths.

C. Comparison With Previous Work

In Table IV, we compare our scheduling results with the CVLS approach [4], the HRA approach [39], the *exact* approach presented by Radivojevic [17], and the approach presented by Santos *et al.* [5]. The classical high-level synthesis benchmarks used for comparisons in these works are used here as well. These are: *kim* [39], *parker* [40], *waka* [4], and *rotor* [17]. The columns present the number of basic blocks, the resources used for scheduling and the longest path length (cycles) of the schedule produced by each approach. For these small benchmarks, nearly all the approaches (including *Spark*) are able to achieve the shortest scheduling length. We are unable to compare area and control costs, since these have not been published in previous work.

VIII. CONCLUSION AND FUTURE WORK

We presented a set of speculative code motions that reorder, speculate, and sometimes even increase the number of operations in a behavioral description so as to achieve a higher quality of synthesis results. These code motions are essential for minimizing the effects of the choice of control flow or programming style in high-level languages. We presented a scheduling heuristic that guides these code motions and improves scheduling results (schedule length and FSM states) and logic synthesis results (circuit area and delay) for moderately complex real-life designs by up to 50% in performance and controller size, when compared to list scheduling techniques that allow code motions only within basic blocks. We implemented the code motions and heuristics in the *Spark* high-level synthesis framework that provides a platform for applying a range of coarse-grain and fine-grain code optimizations aimed at improving synthesis results. Future work entails developing more comprehensive cost models that incorporate the control and interconnect costs of the code motions.

ACKNOWLEDGMENT

The authors would like to thank all the reviewers for their comments and for helping significantly improve the quality of this publication.

REFERENCES

[1] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer cyber," in *Proc. Design, Automation Test Eur.*, 1999, pp. 390–393.
[2] DK Design Suite [Online]. Available: http://www.celoxica.com.
[3] Behavioral Design Suite [Online]. Available: http://www.forteds.com.
[4] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors," in *Proc. Design Automation Conf.*, 1992, pp. 112–115.
[5] L. C. V. dos Santos, "Exploiting instruction-level parallelism: A constructive approach," Ph.D. dissertation, Dept. Elect. Eng., Eindhoven Univ. Technol., Eindhoven, U.K., 1998.
[6] S. Haynal, "Automata-based symbolic scheduling," Ph.D. dissertation, Dept. Elect. Comput. Eng., Univ. California, Santa Barbara, 2000.
[7] G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Wavesched: A novel scheduling technique for control-FBW intensive designs," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 505–523, May 1999.
[8] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proc. Int. Conf. VLSI Design*, 2003, pp. 461–466.
[9] S. Gupta, N. Savoiu, N. D. Dutt, R. K. Gupta, and A. Nicolau, "Conditional speculation and its effects on performance and area for high-level synthesis," in *Proc. Int. Symp. Syst. Synthesis*, 2001, pp. 171–176.
[10] SPARK Parallelizing High-Level Synthesis Framework Website [Online]. Available: http://www.cecs.uci.edu/~spark.
[11] A. Nicolau and S. Novack, "Trailblazing: A hierarchical approach to percolation scheduling," in *Proc. Int. Conf. Parallel Process.*, 1993, pp. 120–124.
[12] S. Novack and A. Nicolau, "An efficient, global resource-directed approach to exploiting instruction-level parallelism," in *Proc. Conf. Parallel Architect. Compilation Tech.*, 1996, pp. 87–96.
[13] M. Potkonjak and J. Rabaey, "Optimizing resource utlization using tranformations," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 277–293, Mar. 1994.
[14] R. Walker and D. Thomas, "Behavioral transformation for algorithmic level IC design," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 1115–1128, Oct. 1989.
[15] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Norwell, MA: Kluwer, 1992.
[16] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[17] I. Radivojevic and F. Brewer, "A new symbolic technique for control-dependent scheduling," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 45–57, Jan. 1996.
[18] G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Incorporating speculative execution into scheduling of control-FBW intensive behavioral descriptions," in *Proc. Design Automation Conf.*, 1998, pp. 108–113.
[19] L. C. V. dos Santos and J. A. G. Jess, "A reordering technique for efficient code motion," in *Proc. Design Automation Conf.*, 1999, pp. 296–299.
[20] M. Rim, Y. Fann, and R. Jain, "Global scheduling with code-motions for high-level synthesis applications," *IEEE Trans. VLSI Syst.*, vol. 3, pp. 379–392, Sept. 1995.
[21] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. 30, pp. 478–490, July 1981.
[22] A. Nicolau, "Uniform parallelism exploitation in ordinary programs," in *Proc. Int. Conf. Parallel Process.*, 1985, pp. 614–618.
[23] K. Ebcioglu and A. Nicolau, "A global resource-constrained parallelization technique," in *Proc. 3rd Int. Conf. Supercomput.*, 1989, pp. 154–163.
[24] A. Orailoglu and D. D. Gajski, "Flow graph representation," in *Proc. Design Automation Conf.*, 1986, pp. 503–509.
[25] M. Girkar and C. D. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, pp. 166–178, Mar. 1992.
[26] M. C. McFarland, "The value trace: A Database for automated digital design," Design Research Center, Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. DRC-01-4-80, 1978.
[27] V. Chaiyakul, D. D. Gajski, and L. Ramachandran, "Minimizing syntactic variance with assignment decision diagrams," Dept. Inform. Comput. Sci., Univ. California, Irvine, Tech. Rep. ICS-TR-92-34, 1992.
[28] D. Ku and G. De Micheli, "Relative scheduling under timing constraints," in *Proc. Design Automation Conf.*, 1990, pp. 59–64.
[29] A. A. Kountouris and C. Wolinski, "Hierarchical conditional dependency graphs as a unifying design representation in the CODESIS high-level synthesis system," in *Proc. Int. Symp. Syst. Synthesis*, 2000, pp. 66–71.
[30] D. W. Wall, "Limits of instruction-level parallelism," in *Proc. Int. Conf. Architect. Support Program. Lang. Oper. Syst. (ASPLOS)*, 1991, pp. 176–188.
[31] M. S. Lam and R. P. Wilson, "Limits of control fbw on parallelism," in *Proc. Int. Symp. Comput. Architect.*, vol. 20, 1992, pp. 46–57.
[32] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Mateo, CA: Morgan Kaufmann, 1997.
[33] S. Gupta, "Coordinated coarse-grain and fine-grain optimizations for high-level synthesis," Ph.D. dissertation, Dept. Inform. Comput. Sci., Univ. California, Irvine, 2003.
[34] S.-M. Moon and K. Ebcioglu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *Proc. Int. Symp. Microarchitect.*, 1992, pp. 55–71.
[35] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," Center for Embedded Comput. Syst., Univ. California, Irvine, Tech. Rep. CECS-TR-02-35, 2002.
[36] ——, "Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs," in *Proc. Design, Automation Test Conf.*, 2003, pp. 270–275.
[37] UCLA Mediabench Benchmark Suite, C. Lee, M. Potkonjak, and W. H. M. Smith. [Online]. Available: http://cares.icsl.ucla.edu/MediaBench/.
[38] GNU Image Manipulation Program [Online]. Available: http://www.gimp.org.
[39] T. Kim, N. Yonezawa, J. W. S. Liu, and C. L. Liu, "A scheduling algorithm for conditional resource sharing—A hierarchical reduction approach," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 425–438, Apr. 1994.
[40] A. C. Parker, J. Pizarro, and M. Mlinar, "MAHA: A program for datapath synthesis," in *Proc. Design Automation Conf.*, 1986, pp. 461–466.

Pitfalls of Hierarchical Fault Simulation

Sandip Kundu

***Abstract*—Certain circuit structures, such as self-loop, asynchronous reset, and clock division, may not be visible in a hierarchical (mixed) simulation system. Since the simulator does not know about their existence, it cannot cope with them like it normally would in a flat circuit. If this leads to a logic-simulation problem, users can usually discover them easily during the validation process. However, if it only causes fault-simulation inaccuracy, it is hard to find the problem. In this paper, we show examples illustrating their existence. The examples negate an assumption that has been used in many papers on mixed-mode simulation. The examples have been abstracted from real industrial designs of microprocessors.**

***Index Terms*—Fault, fault simulation, sequential circuit fault simulation, sequential logic circuit testing, testing.**

I. Introduction

Functional testing has remained a popular test method. It offers several advantages that are not easily achievable with other test approaches. First of all, it is done atport speed, allowing delay defects to be exposed. Second, it is not done in a special mode. Therefore, any signal race that may compromise functional operation is easily uncovered. Thirdly, at speed application leads to reduced test time. At today's device and tester speed, it may be possible to apply several million functional patterns in fraction of a second. Lastly, because it is applied through chip input/outputs (I/Os), whose count does not increase at the same rate as the chip size, the test-data volume problem is largely contained. Today's system-on-a-chip may easily contain upwards of ten million transistor with hundreds of I/Os and more than 200 000 flip-flops. If each flip-flop is turned into a I/O cell through scan operation, a single scan test vector will consist of more than $200\,000 \times 2 = 400\,000$ bits of data. This leads to an unacceptably large tester data volume. Usually with functional test, test data volume is not as large an issue.

Lack of design-for-testability (DFT) standard for cores has lead to an even stronger adoption of functional test. This approach allows testing of the whole system, hiding several problems such as those of internal accessibility, assorted internal clocking schemes, varying degree of internal DFT support, and potential nonavailability all test modes through test access ports. If each core has its own test protocol, test controller design to support all of them also becomes a time-to-market issue.

However, functional test approach also suffers from several problems. The first and foremost of which is test development process, which is, more often than not, a manual problem. The test writer needs to have stopping criteria to know when to quit. The test writer also needs feedback to know which areas to target. This is usually done by single stuck-at fault simulation. Single stuck-at fault coverage ensures that every signal has been exercised and observed.

Unfortunately, it is also a large problem. It is impossible to do fault simulation of a design with five million gates and eight million faults against ten million patterns, which may be a typical representative of a microprocessor or a system on a chip today. While fault sampling technique may cut the size of the problem down, it is not adequate.

This has lead several researchers to explore hierarchy as a way to manage data volume and improve runtime. The key principle is to target

Manuscript received March 12, 2003; revised April 28, 2003 and July 8, 2003. This paper was recommended by Associate Editor S. M. Reddy.

The author is with the Intel Corporation, Austin, Texas 78746 USA (e-mail: sandip.kundu@intel.com).

Digital Object Identifier 10.1109/TCAD.2003.822099