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

Sumit Gupta   Nick Savoiu

Nikil Dutt   Rajesh Gupta   Alex Nicolau

CECS

Technical Report #02-29

October 2002

Center for Embedded Computer Systems

Department of Information and Computer Science

University of California, Irvine

http://www.cecs.uci.edu/∼spark

{sumitg,savoiu,dutt,rgupta,nicolau}@cecs.uci.edu

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 syntactic variance or 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 evaluate the effects of these speculative *code motions in terms of the cycles on the longest path (performance), the number of states in the finite state machine (FSM) (controller complexity), length of the critical path in the synthesized netlist (clock period) and the area of the synthesized netlist. Significant improvements in performance and reduction in controller complexity are observed. However, although critical path lengths remain fairly constant, area of the design increases due to increasing complexity of the steering logic and associated control logic. To address this, we present a methodology to reduce interconnections based on resource binding, which also leads to improvements in critical path lengths. These code transformations and controller optimizations have been implemented in a high-level synthesis research framework called* Spark*, which takes a behavioral description in ANSI-C as input and generates synthesizable register-transfer level VHDL. The experiments described in this paper have been performed on two real-life high-level synthesis design targets, namely, the MPEG-1 and ADPCM algorithms. The results demonstrate reductions in the number of states in the FSM controller and in the cycles on the longest path of between 35 % to 50 % and subsequently, the interconnect minimizing binding methodology achieves area reductions between 15 % to 32 %.*

*A version of this paper is under revision with the IEEE Transactions on CAD.

Contents

1 Introduction **4**

2 Related Work **5**

3 Code Motions in High-Level Synthesis **7**

3.1 Using Speculation in High-Level Synthesis 7

3.2 Reverse Speculation 8

3.3 Early Condition Execution 10

3.4 Conditional Speculation 11

4 The Spark High Level Synthesis Framework **12**

4.1 A Model for Control Intensive Designs 14

4.2 Code Motion Techniques in the Transformations Toolbox 16

4.3 Eliminating data dependencies by Dynamic Renaming 18

4.4 Priority-based Global List Scheduling Heuristic 19

4.5 Determining the Application of the Code Motions 21

5 Effects of Code Motions on Quality of Synthesis Results **23**

5.1 Effects on Performance 23

5.2 Effects on Area and Clock Period 26

6 Reducing Interconnect **28**

6.1 Operation to Functional Unit Binding 29

6.2 Variable to Register Binding 30

7 Results of Resource Binding **30**

8 Conclusions and Future Work **32**

List of Figures

1 **Extracting the inherent parallelism in a control-data flow graph by speculating the addition operations. This requires an additional resource, but leads to a reduction in the longest path.** 8

2 **Reverse Speculation for the waka benchmark (a) original design (b) operation c is reverse speculated into the branch of the conditional that uses its result. This leads to a reduction in schedule length by one.** 9

3 **Difference between reverse speculation and the classical notion of downward code motion in CDFGs. (a) Sample HTG representation (b) Operation a is reverse speculated by duplicating into operations $a1$ and $a2$ (c) CDFG representation of same example (d) Downward code motion of operation a.** 10

4 **Code restructuring by *early condition execution* (a) original design (b) comparison operations p and q are scheduled as soon as possible to enable early condition checking. All unscheduled operations before the conditional checks are reverse speculated into the conditional branches.** 11

5 **(a) A sample control-data flow graph (b) Operations x and y are speculated leaving idle slots in the conditional branches (c) Operation z is *conditionally speculated* into conditionals BB_1 and BB_2.** 12

6 **The Spark High Level Synthesis Framework: a complete synthesis system that provides a path from an architectural description in C to synthesizable register-transfer level VHDL.** 13

7 **The hierarchical task graph (HTG) representation of the "waka" benchmark [13]. The priorities of each operation are marked next to each operation node.** 15

8 **The hierarchical task graph representation of a For Loop.** 15

9 **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.** 17

10 **Moving one operation across another operation while eliminating (a) an anti dependency (b) an output dependency and (c) a flow dependency.** 18

11 **(a) Priority-based List Scheduling Heuristic (b) Determining the list of *Available* operations.** 20

12 **Heuristic to determine whether to conditionally speculate an operation op into multiple basic blocks given by $BBList$, while scheduling it into scheduling $step$ in basic block BB_{step}.** 22

13 **Logic synthesis results after various code motions for the MPEG $pred_case2$, $pred_case0_1$ and $calc_forward$ functions and the ADPCM Encoder function; circuit delay decreases significantly but area can increase marginally.** 27

14 **Typical critical paths in control-intensive designs pass through the steering logic and the associated control logic.** 28

15 **An example of binding leading to a large number of interconnections.** 29

16 **Reducing interconnect by improved (a) operation binding (b) variable binding.** 29

17 **Results of logic synthesis after applying a non-interconnect aware ("regular") binding and an interconnection minimizing resource binding for the MPEG $pred_case2$, $pred_case0_1$ and $calc_forward$ functions and the ADPCM Encoder function.** 31

1 Introduction

Recent years have seen the widespread acceptance and use of language-level modeling of digital designs. Increasingly, the typical design process starts with design entry in a hardware description language at the register-transfer level, 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 these factors have led to a renewed interest in high level synthesis from behavioral descriptions, both in the industry and in academia [1, 2, 3, 4, 5].

However, current synthesis efforts have several limitations: Synthesizability is guaranteed on a small, constrained sub-set of the input language and the language level optimizations are few and their effects on final circuit area and speed are not well understood. Also, for designs with moderately complex control flow, the quality of synthesis results is poor due to the presence of conditionals and loops. In general, designers are often given minimal controllability 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 syntactic variance in the input description. Several scheduling algorithms have been proposed to address this issue, that employ beyond-basic-block code motion techniques such as speculation to extract the inherent parallelism in designs and increase resource utilization.

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 [6, 7]. This may be done by *reverse speculation*, where operations before conditionals are moved 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*, moves conditions so that they are evaluated as early as possible. The motivation for this transformation comes from the fact that once a condition has been evaluated, all the operations in its branches are ready to be scheduled. However, although these code motions are shown to be useful, there needs to be a judicious balance between when to speculate, when to reverse speculate and so on.

We show how a simple priority-based global list scheduling heuristic can be used to direct these code motion transformations and obtain significant reductions in schedule lengths and controller complexity. Synthesis results, however, show significant area overheads are incurred due to aggressive code motions used during scheduling.

These area overheads are due to higher resource utilization and resource sharing caused by the code motions that in turn lead to increased steering logic and multiplexors. The moderately control-intensive nature of the benchmarks we have considered further increases the opportunities for significant resource sharing among mutually exclusive operations. Higher resource sharing and utilization implies that the synthesized netlist has increased control logic both in terms of control signal generation and interconnect. *Interconnect*, here, refers to the multiplexors and buses that connect components together.

To address the complexity of the interconnect, we have implemented a resource binding methodology that binds operations to functional units and variables to registers, such that operations with the same inputs or outputs are bound to the same functional units and then, variables that are inputs or outputs to the same functional units, are mapped to the same registers [8]. In this way, the interconnect between functional units and registers is reduced. Although this idea of binding with the aim of minimizing interconnect is not new, the formulation that we use in our approach to solve this problem is new, along with its integration in a high-level synthesis framework that includes interconnect optimizations across nested conditionals.

These code motion transformations and the interconnect minimization methodology, along with a control synthesis and optimization strategy has been implemented in a modular and extensible high-level synthesis research system called *Spark*. The system uses parallelizing compiler technology developed previously within our group [9, 10] and re-instruments and modifies it for high-level synthesis. Since one of the outputs of the system is synthesizable register-transfer level (RTL) VHDL, the system 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 and no function recursion. In this way, Spark provides an integrated flow from architectural design to logic synthesis.

The rest of this paper is organized as follows: the next section reviews previous related work followed by a presentation of a set of speculative code motions that are useful in high-level synthesis. Section 4 presents the Spark framework in which these code motions are implemented, the internal representation model used to capture the designs, several of the implemented transformations and the scheduling heuristic. We then study the effects of these code motions on performance, controller costs and synthesis results. The interconnect minimization strategy is presented in Section 6 followed by results of this methodology. Finally, we conclude the paper with a discussion.

2 Related Work

Early high-level synthesis work concentrated on data-flow designs and applied optimizations such as algebraic transformations, re-timing and code motions across multiplexors for improved synthesis results [11, 12]. Subsequent work has presented speculative code motions for mixed control-data flow type of designs and demonstrated

their effects on schedule lengths. CVLS [13] uses condition vectors to improve resource sharing among mutually exclusive operations. Radivojevic et al. [14] present an exact symbolic formulation that generates an ensemble schedule of valid, scheduled traces. Haynal [4] uses an automata-based approach for symbolic scheduling of cyclic behaviors under sequential timing and protocol constraints. This is an exact approach, but can grow exponentially in terms of internal representation size. The "Waveschedule" approach [15] incorporates speculative execution into high level synthesis to achieve its objective of minimizing the expected number of cycles. Santos et al. [16] and Rim et al. [17] support generalized code motions during scheduling in synthesis systems whereby operations can be moved globally irrespective of their position in the input description.

The chief limitation of earlier work is on the control complexity of the input description that can be synthesized. In particular, arbitrary nested loops and conditionals are not handled. For approaches that enumerate all the control paths or traces in the design, complex control flow can lead to an explosion in the number of traces that need to be scheduled and validated. Also, code motion techniques that require code motions into multiple control paths such as conditional speculation (duplication of operations into conditionals) may not be easily supported. Previous approaches also do not maintain information about hierarchical structuring of the code, which leads to expensive and inefficient code motion techniques (see Section 4.1).

Further, most previous works compare the effectiveness of their algorithms primarily in terms of schedule lengths; their impact on control generation is not considered. Industry experience shows that, often, critical paths in control-intensive designs pass through the control unit and steering logic. To this end, Rim et al. [17] use an analytical model to estimate the cost of additional interconnect and control caused by code duplication during code motions. Bergamaschi [18] proposes the behavioral network graph to bridge the gap between high-level and logic-level synthesis and aid in estimating the effects of one on the other.

Binding techniques for reducing interconnect have also been studied before [8, 19, 20]. Tseng et al. [21] use clique partitioning heuristics to find a clique cover for a module allocation graph. Paulin et al. [22] perform exhaustive weight-directed clique partitioning of a register compatibility graph to find the solution with the lowest combined register and interconnect costs. Stok et al. [23] use a network flow formulation for minimum module allocation while minimizing interconnect. Gebotys et al. [24] present an integer-programming model for simultaneous scheduling and allocation that minimizes interconnect. Mujumdar et al. [25] consider operations and registers in each time-step one at a time and use a network flow formulation to bind them.

A range of code motion techniques similar to those presented in our work have also been previously developed for high-level language software compilers (especially parallelizing compilers) [26, 27, 28]. Although the basic transformations (e.g. dead code elimination, copy propagation) can be used in synthesis as well, other transforma-

tions need to be re-instrumented for synthesis. This is usually because the cost models in compilers and synthesis tools are different. For example, in compilers there is generally a uniform push towards executing operations as soon as possible by speculative code motions. Indeed, the optimality claims in percolation and trace scheduling are based entirely upon maximum movement of operations out of conditional branches. In the context of high-level synthesis, such notions of optimality have little relevance. The additional hardware costs associated with code motions must be taken into account while making scheduling decisions.

The contributions of this work include: (a) three code motion transformations derived from speculative execution techniques that are specifically targeted for high-level synthesis, (b) a heuristic approach to drive the application of these transformations and (c) a framework that provides a toolbox of code transformations and supporting compiler transformations. The toolbox approach 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 is a complete high-level synthesis system that provides a path from an unrestricted input behavioral description down to register-transfer level code, that can then be synthesized by commercial logic synthesis tools.

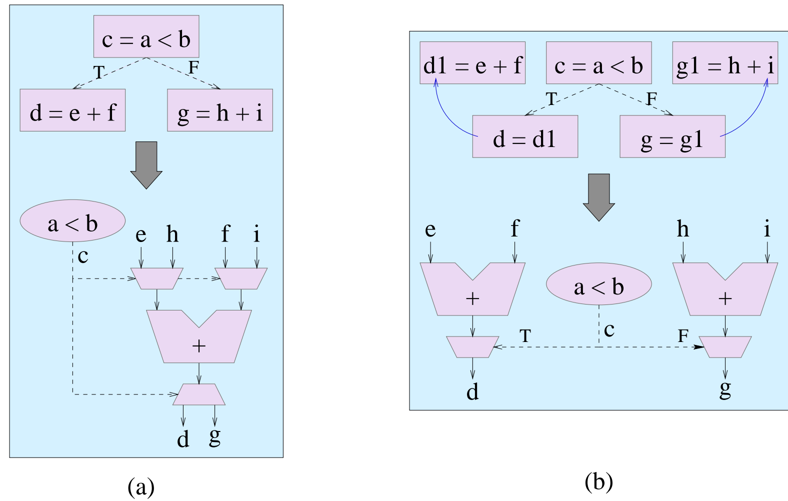
3 Code Motions in High-Level Synthesis

Code motions refer to source level transformations with the goal of improving resource utilization and extracting maximal parallelism in designs with complex control flows. In the presence of control structures, maximal parallelism can be extracted by exposing concurrency using code motions that move operations beyond control boundaries. One of the key enabling transformations for such type of code motions is speculation. *Speculative execution* refers to the execution of an operation that was to execute under a condition, before the value of this condition has been evaluated. In the compiler context, if the condition evaluates to a value that was not the predicted value, then compensation code has to be executed. However, in the hardware synthesis context, we can simply choose to either commit the results or discard them based on the evaluation of the conditions.

Although speculation has been used earlier in compilers, its use in high-level synthesis has been limited. This is because unconstrained speculative execution can actually worsen synthesis results. Synthesis techniques must apply such transformations within the context of the structure of control flow in which the target code is placed. In the next few sections, we present several types of speculative code motions that are useful for high-level synthesis, starting off with an overview of speculation.

3.1 Using Speculation in High-Level Synthesis

Speculation is demonstrated by an example in Figure 1. In Figure 1(a), variables d and g are calculated based on the result of the calculation of the conditional c. Since d and g are executed on different branches of a conditional



(a) (b)

Figure 1. **Extracting the inherent parallelism in a control-data fbw graph by speculating the addition operations. This requires an additional resource, but leads to a reduction in the longest path.**

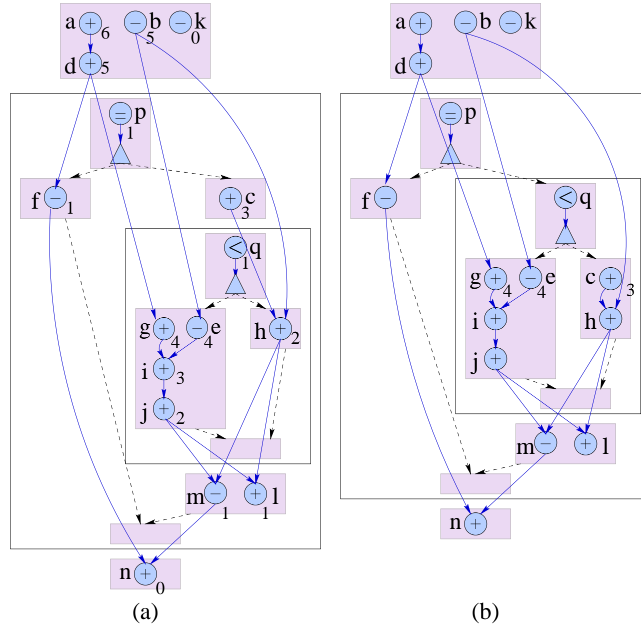
block, these two operations are *mutually exclusive*. They can, hence, be scheduled on the same hardware resource with appropriate multiplexing of the inputs and outputs as shown in the circuit in Figure 1(a).

Now, consider that enough resources (an additional adder) are available; then the operations within the conditional branches can be calculated *speculatively* and concurrently with the calculation of the conditional c as shown in Figure 1(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 Figures 1(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 more storage for the intermediate results. So, uncontrolled aggressive speculation can lead to worse results due to the extra resources and complex control required. On the other hand, idle resources can be better utilized by executing operations speculatively on them. Hence, speculation along with other code motions needs to be directed by a global scheduling heuristic.

3.2 Reverse Speculation

Reverse speculation refers to downward movement of operations into conditional branches. This may be useful in instances where an operation inside the conditional branch is on the longest path through the design, whereas an operation before the conditional is not. The operation outside the conditional branch can then be moved down



(a) (b)

Figure 2. **Reverse Speculation for the waka benchmark (a) original design (b) operation** *c* **is reverse speculated into the branch of the conditional that uses its result. This leads to a reduction in schedule length by one.**

or *reverse speculated* into both the conditional branches, so that the resource freed can be better utilized by the operation on the longest path. This code motion can reduce the variance in scheduling results caused by the choice of placement of operations in the input description by the designer. Reverse speculation has been variously referred to as *lazy* code motion or execution and *duplicating down* in past literature [17, 29].

Reverse speculation is demonstrated in Figure 2 for the *waka* benchmark [13] [1]. The operations *g* and *e* lie on the longest data dependency path of the design that starts at operation *a* and ends at operation *n* as shown in Figure 2(a) (solid lines in this graph denote data dependencies). Also, operation *c* is on a shorter dependency path that starts at operation *c* and ends at operation *n*. Therefore, operations *g* and *e* are determined to have a higher priority than operation *c*. Hence, operation *c* can be *reverse speculated* or moved into the conditional branches as shown in Figure 2(b); this leads to a reduction in the number of cycles in the schedule by one.

Note that, as shown in Figure 2(b), the reverse speculation algorithm detects that the result of operation *c* is used only in one of the branches of the conditionals and hence, moves it only into that branch. In the general case, reverse speculation may lead to duplication of the operation into both the true and the false branch of a conditional. Hence, implications on the hardware generated must be taken into account while applying this code motion.

An important difference between reverse speculation and the classical notion of downward code motion as presented in previous work using CDFGs (control-data flow graphs), is demonstrated by an example in Figure

[1] For a description of the graphical representation used in this figure see Section 4.1.

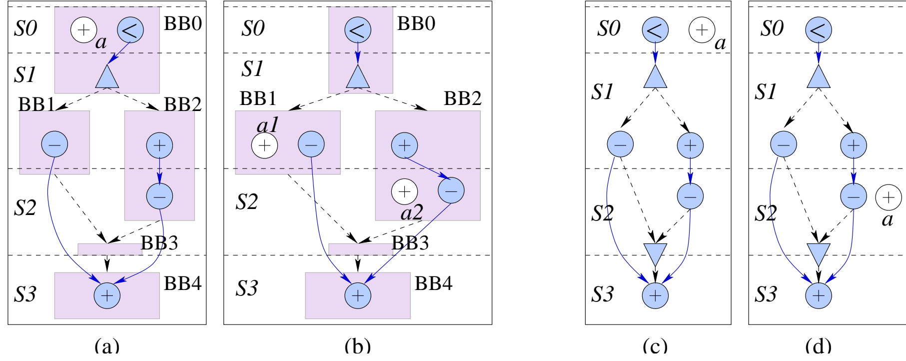


Figure 3. **Difference between reverse speculation and the classical notion of downward code motion in CDFGs. (a) Sample HTG representation (b) Operation a is reverse speculated by duplicating into operations $a1$ and $a2$ (c) CDFG representation of same example (d) Downward code motion of operation a.**

3. In this example, when operation a is reverse speculated, it is duplicated into operations $a1$ and $a2$ in basic blocks BB_1 and BB_2 respectively, as shown in Figure 3(b). These operations may now be scheduled in different time steps or states independent of each other. However, in the classical notion of downward code motion, an operation a in an equivalent CDFG representation (shown in Figure 3(c)), is usually moved down into another time step as shown in Figure 3(d). The ability to duplicate operations across fork (or branch) nodes and across join nodes (as explained in Section 3.4), gives the scheduler greater flexibility in scheduling the duplicated operations in mutually exclusive basic blocks. This flexibility is scheduling a duplicated operation in different time steps also differentiates code motions in high-level synthesis from their counterparts in compilers.

3.3 Early Condition Execution

Reverse speculation can be coupled with another novel transformation, namely, *early condition execution*. This transformation involves restructuring the original code, so as to evaluate conditional checks as soon as possible. This in effect means that the conditional check is “moved up” in the design, and hence, all operations before the conditional are reverse speculated into the conditional. This transformation is motivated by the fact that evaluating a conditional check early, resolves the control dependency for operations within conditional branches. This allows these operations to be available for scheduling sooner.

Early condition execution is demonstrated for the *waka* benchmark in Figure 4. In Figure 4(b), the comparison operations p and q, that calculate the conditions, are scheduled as soon as possible and hence, the conditionals based on them can be checked early (Boolean conditional checks are denoted by triangles in these figures). Unscheduled operations from basic blocks preceding the conditional (d, k and c) are *reverse speculated* into the conditional branches as shown in Figure 4(b). Note that operations d and c are reverse speculated into only those branches that use their results. Clearly, the design after applying the transformation as shown in Figure 4(b) has a

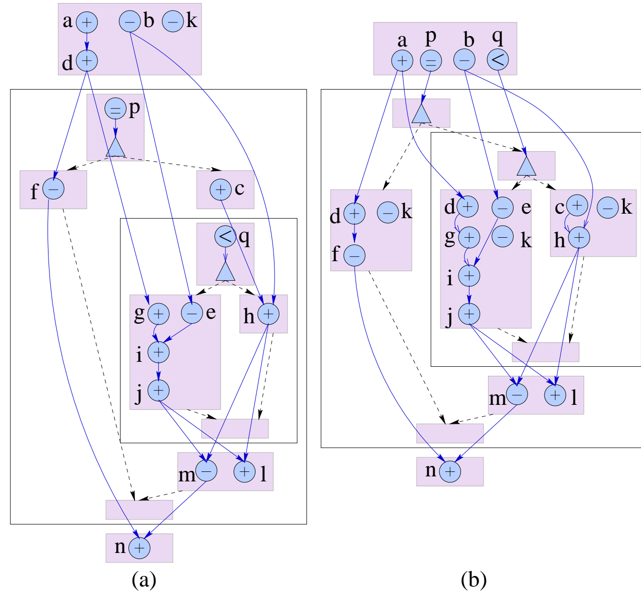


Figure 4. **Code restructuring by *early condition execution* (a) original design (b) comparison operations p and q are scheduled as soon as possible to enable early condition checking. All unscheduled operations before the conditional checks are reverse speculated into the conditional branches.**

shorter schedule length than the original design in Figure 4(a).

Although techniques developed previously can also execute conditions as early as their conditional check has been evaluated, the notion of operation duplication by reverse speculation or *downward* code is new here. As explained in the previous section, the flexibility of scheduling the duplicated operations in different time steps afforded by reverse speculation is the main difference with similar techniques developed for CDFGs. Using a more efficient hierarchical representation of the input description (see Section 4.1), along with these code motions, enables the Spark system to more efficiently and implicitly extract and use information about mutual exclusivity of operations and hence, increase resource sharing.

3.4 Conditional Speculation

Control intensive designs often have instances where the basic blocks that comprise the branches of a conditional do not have enough operations to fully utilize the resources allocated to the design. Speculation also creates such "idle slots" on resources by moving operations out of conditionals. These idle slots can be filled or utilized by scheduling operations that lie in basic blocks after the conditional branches. These operations can be *duplicated up* into both branches of the conditional and executed speculatively. We call this code motion, *conditional speculation* (CS). This is similar to the duplication-up code motion used in compilers and the node duplication transformation discussed by Wakabayashi et al. [13].

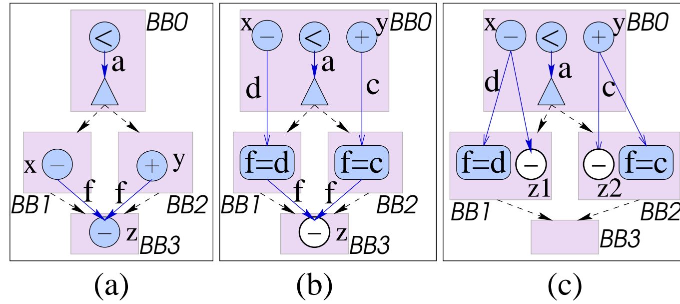


Figure 5. **(a) A sample control-data fbw graph (b) Operations x and y are speculated leaving idle slots in the conditional branches (c) Operation z is *conditionally speculated* into conditionals BB_1 and BB_2.**

Figure 5 demonstrates how such idle slots are created by speculation and how conditional speculation can be used to fill them. In Figure 5(a), consider that the operations x and y both write to the variable f in their respective conditional branches BB_1 and BB_2. Now, 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 Figure 5(b). The results of the speculated operations are written into new destination variables, d and c, that are not committed until the corresponding condition is evaluated, i.e., the results of the speculated operations are *written back* to the variable f only within the conditional branches.

Figure 5(b) demonstrates that the speculation of these operations leaves "idle" slots in which no operations have been scheduled on the resources. Furthermore, operation z is dependent on either the result of operation x or operation y depending on how the condition evaluates (i.e. operation z is dependent on the variable f). Operations such as z, that lie in basic blocks after the conditional branches, can be duplicated up or *conditionally speculated* into both branches of the conditional to fill idle slots as illustrated in Figure 5(c).

Note that condition speculation does not necessarily need speculation to be done first to activate it as shown in the example above. As stated earlier, there are often empty slots within conditional branches, that go unused unless operations are conditionally speculated from after the conditional branches.

Code motion transformations such as those presented above allow flexible motion of operations so that the manner in which the input description was written has little or no effect on the synthesis results. This syntactic invariance is an essential requirement of high-level synthesis systems because the behavioral nature of the input specifications allows designers significant freedom in the choice of programming style.

4 The Spark High Level Synthesis Framework

Our synthesis framework, *Spark* is a modular and extensible high-level synthesis system that provides a number of code transformation techniques. Spark has been designed to aid in experimenting with new transformations and

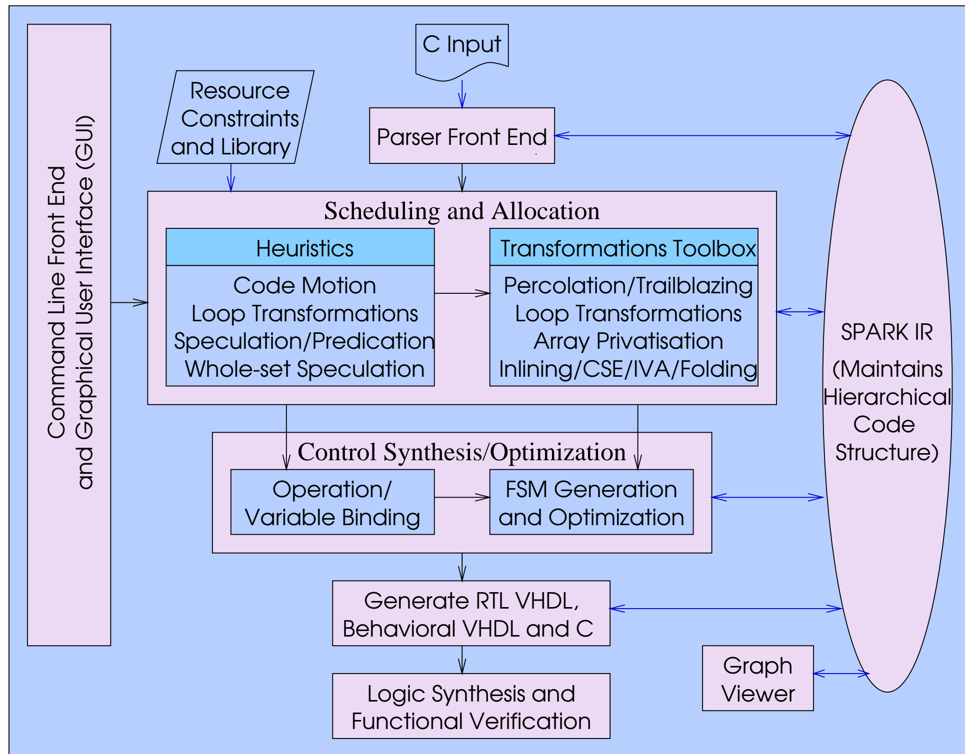


Figure 6. **The Spark High Level Synthesis Framework: a complete synthesis system that provides a path from an architectural description in C to synthesizable register-transfer level VHDL.**

heuristics that optimize the quality of synthesis results. Figure 6 provides an overview of the Spark system. The input language for design descriptions is ANSI-C, currently with the restrictions of no pointers and no function recursion. This input description is parsed into a hierarchical intermediate representation described in Section 4.1.

The core of the synthesis system has a *transformations toolbox* that consists of a set of information gathering passes, basic code motion techniques and several compiler transformations. 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 completely independent, writing new heuristics can be as simple as making calls to the toolbox.

The transformations toolbox contains a data dependency extraction pass, parallelizing code motion techniques [30, 9], dynamic renaming of variables, the basic operations of loop pipelining (or software pipelining) and some supporting compiler passes such as constant propagation and dead code elimination [31]. The code motion techniques and dynamic renaming are detailed in Sections 4.2 and 4.3.

A typical design flow through the Spark system takes as input a behavioral description of a design in ANSI-C, creates the intermediate format, schedules the design, performs control synthesis, and finally generates an output

in register-transfer level VHDL. The passes and transformations that are used can be controlled by the designer using scripts, hence, allowing experimentation with various transformations and heuristics.

For instance, the designer may decide to use the trailblazing code motion technique in conjunction with variable renaming and schedule operations in the design. In this way, once a scheduling heuristic has scheduled the design based on a given resource allocation, the next stage of the system performs control synthesis and optimization. *Control synthesis* consists of generating control circuits to implement the schedule, binding operations to functional units, tying the functional units together (interconnect binding), allocating and binding storage (registers) and generating the steering logic. The control unit is generated using the finite state machine controller style. Resource binding is done as a part of control generation since it affects the generation of the steering control logic.

The back-end of the Spark system consists of a register-transfer level (RTL) VHDL generator. This VHDL is synthesizable by commercial logic synthesis tools [32] and hence, the Spark system integrates into the standard synthesis design flow. This completes the direct 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 netlist.

In the next few sections, we examine, in more detail, some of the transformations and heuristics of the Spark synthesis framework that aid in implementing the code motion transformations presented earlier. We start with a description of the internal intermediate model used for capturing the input description.

4.1 A Model for Control Intensive Designs

The *Spark* system stores the behavioral description in an intermediate representation (IR) that retains all the information given in the input description. Hence, for example, the IR maintains information about variables used in the source code and it does not reduce array accesses to pointer arithmetic and an associated pointer access. This is critical for enabling source-level transformations and making global decisions about code motion. Retaining information about variables used in the input description is also very important from the point of view of user-interaction, since it allows a user to track the intermediate results as each transformation is applied to the input code. This is explained in more detail in Section 4.3.

The intermediate representation used in Spark consists of basic blocks encapsulated in *Hierarchical Task Graphs* (HTGs) [9, 33]. An HTG is a directed acyclic graph that has three types of nodes: simple nodes, compound nodes and loop nodes.

1) *Simple nodes* represent nodes that have no sub-nodes. There are two such nodes, namely, statement nodes and single nodes. *Statement* nodes, or statements for short, represent an aggregation of operations that execute concurrently. Statements that have no control flow between them are aggregated together into basic

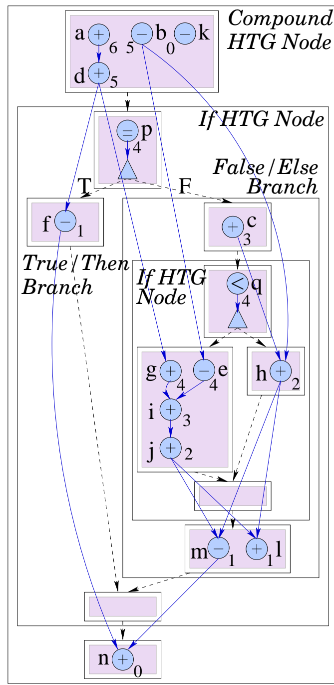


Figure 7. **The hierarchical task graph (HTG) representation of the 'waka" benchmark [13]. The priorities of each operation are marked next to each operation node.**

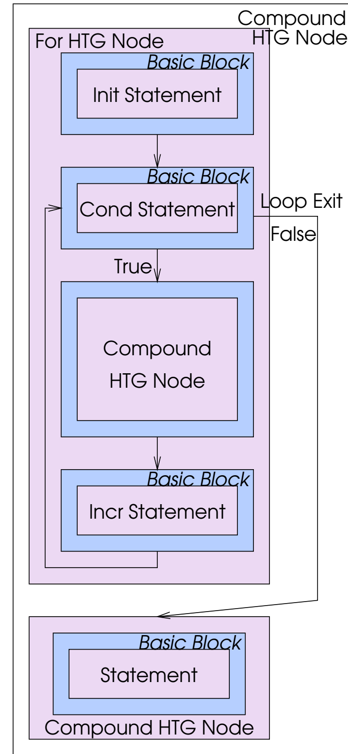


Figure 8. **The hierarchical task graph representation of a For Loop.**

blocks, which are encapsulated into *single nodes*. These single nodes (or basic blocks) then form parts of compound HTG nodes or loop nodes.

2) *Compound HTG nodes* are hierarchical in nature, i.e., they can 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) and have a loop head and a loop tail that are simple nodes and a loop body that is a compound HTG node.

HTGs are constructed from the input code by creating HTG nodes for each if-then-else, for-loop, while-loop et cetera in the code. Expressions in the code are stored as abstract syntax trees [31] and each expression is initially encapsulated in a statement HTG node of its own. Since HTGs maintain a hierarchy of nodes, they are able to retain coarse, high level information about program structure, in addition to operation level and basic block level information. For example, an if-then-else HTG contains a compound HTG with a single basic block for the conditional check, a compound HTG for the true branch, a compound HTG for the false branch and a compound HTG with a single basic block for the join node. Figure 7 illustrates the HTG for the synthetic benchmark "waka"

[13]. In this figure, the dashed arrows indicate control flow and the solid lines indicate data flow. Operations are denoted by circular nodes with the operator sign within and the triangle indicates a Boolean conditional check. In this figure, there is an if-HTG node, whose false branch contains another if-HTG node. Similarly, the conceptual HTG representation of a *For-loop HTG* is shown in Figure 8; only control flow dependencies are shown in the figure (with solid lines). A for-loop HTG consists of an initialization basic block, the conditional check basic block, a compound HTG node that represents the body of the loop and an optional basic block for the loop index increment.

An important feature of HTGs is that they are *strongly connected components* (SCC) [33]. An SCC region (in this case a HTG node) has the property of having a single entry and a single exit point. This property has several advantages: it can be used to encapsulate complex loops and also irregular regions of code, to simplify and regularize code motion techniques and possibly reduce the amount of patch-up code that needs to be inserted. These features are exploited by code motion techniques such as *Trailblazing* [9] and *Resource-Directed Loop Pipelining* [10], to make hierarchical moves when moving operations in the graph as explained in the next section.

4.2 Code Motion Techniques in the Transformations Toolbox

The code motion techniques implemented in the toolbox of the Spark system are percolation scheduling and trailblazing. *Percolation Scheduling* (PS) was developed as a technique to target code to parallel architectures such as VLIWs (Very Long Instruction Word) processors and vector processors [27, 30]. Percolation scheduling compiles programs into parallel code, by systematically applying semantic preserving transformations. There is a set of four core percolation transformations, *move-op*, *move-cj*, *delete* and *unify*, that are defined in terms of adjacent nodes in a program graph [30]. The move-op and move-cj form the heart of the percolation transformations; they move an operation or a conditional jump up one instruction in the CDFG while preserving the semantics of control and data flow.

The core transformations of percolation have been proven to be complete with respect to the set of all possible local, dependency-preserving transformations on program trees. Once the code motion heuristics have decided which operation to move and the slot to schedule it on, the appropriate percolation transformations are applied so that the operation "percolates" to its assigned resource slot.

However, percolation-based compilers typically suffer from two main efficiency problems: code explosion and linear operation moves. Code explosion occurs when trying to parallelize descriptions with conditional branches, due to duplication of operations into both branches of conditionals and insertion of copy operations. A lot of this unnecessary code explosion is due to the strictly incremental nature of percolation transformations. The problem of *linear operation moves* refers to the fact that in percolation, moving an operation from a node *A* to node *B*

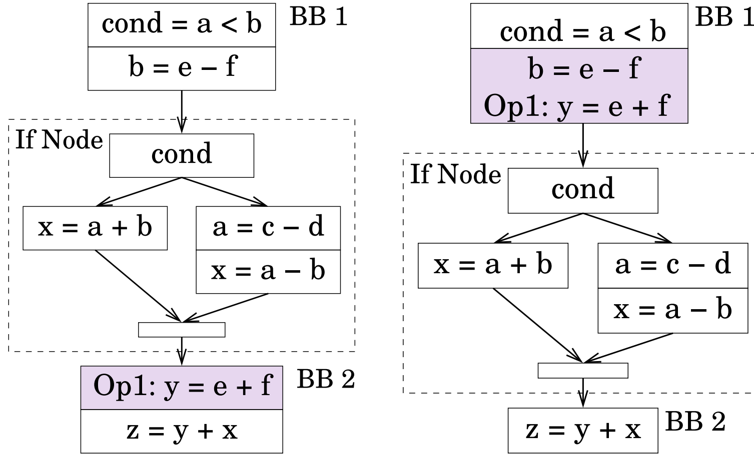


Figure 9. **Trailblazing: Operation $op1$ is moved from basic block BB_2 to basic block BB_1 across the if-then-else HTG node without visiting each basic block inside the node.**

requires a visit to each node on every control path from A to B. To circumvent these problems, the trailblazing code motion technique was proposed.

Trailblazing is a code motion technique that moves operations in a hierarchically structured control-data flow graph [9]. These graphs, known as Hierarchical Task Graphs (HTGs) [33] (see Section 4.1), structure the input description's operations and global information so that non-incremental moves can be made without visiting every operation that is bypassed. At the lowest level, trailblazing is able to perform the same fine-grained transformations as percolation. However, at a higher level, trailblazing is able to move operations across large blocks of code.

While an operation is being moved using trailblazing, if the algorithm comes across a HTG node, it checks to determine if the moving operation has any dependencies with the HTG node. If there are no dependencies, then the operation is moved across the node without visiting each of its sub-nodes. This is demonstrated in Figure 9 with an example. In this figure, the operation $Op : y = e + f$ is moved from basic block BB_2, across the if-then-else HTG node, since it has no data dependencies with any of the operations in this node. Hence, the Op operation can be scheduled in the earlier basic block BB_1 as shown in Figure 9(b). To perform the same code motion, percolation would have duplicated Op into both the branches of the if-block, then moved it up each branch, and finally unified the copies back into Op at the conditional check, hence, in the process visiting each node in the if-then-else block.

The algorithm for implementing trailblazing is discussed in detail in [9]. For implementing these code motion techniques, a data dependency collection pass has to be executed after the input description has been parsed into the intermediate representation. However, some of the data dependencies in the input description can be eliminated by dynamic renaming, as explained in the next section.

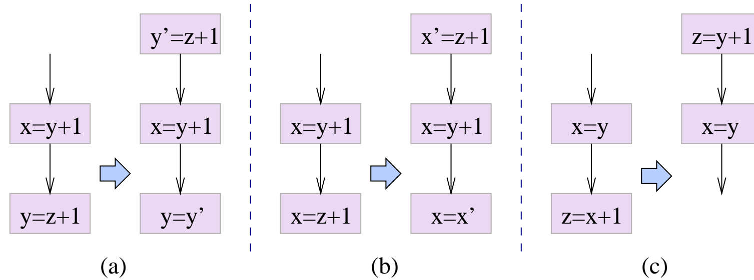


Figure 10. **Moving one operation across another operation while eliminating (a) an anti dependency (b) an output dependency and (c) a flow dependency.**

4.3 Eliminating data dependencies by Dynamic Renaming

A data dependency is said to exist between two operations if the result of one of them interferes with execution of the other. There are four types of data dependencies [29]: a *flow* dependency is said to exist when an operation that writes to a variable is followed by an operation that reads the same variable, an *anti* dependency is when one operation that reads a variable is followed by an operation that writes to the same variable, an *output* dependency exists when two operations write to the same variable one after the other and an *input* dependency when two operations read from the same variable. Of these, input dependencies do not affect scheduling.

Several approaches exist to eliminate some data dependencies. In one popular technique used in high-level synthesis, a control-data flow graph (CDFG) is constructed from the input description in such a way that the variable names from the original source are not maintained. Hence, the CDFG just consists of nodes, which are the operations, and edges that constitute the data flow from one node to another. In this way, only flow dependencies are maintained. However, one of the main reasons for the poor acceptance of high-level synthesis tools among designers has been the inability to control the various transformations applied to the input description and more importantly, to *visualize* the intermediate results. This inability to visualize the intermediate results arises from the fact that once the input description has been captured by a flow-dependency only CDFG, there is no way to correlate the variables and operations used in the input description with the design as represented by the CDFG. In the Spark system, we maintain the complete information about variables used in the input description and hence, construct the hierarchical task graphs (HTGs) along with data dependency graphs that maintain all the data dependency types. Hence, users can correlate the variables and operations from the input description to the intermediate representation used by the synthesis tool.

However, non-flow data dependencies that prevent code motions can often be resolved by dynamic renaming and combining [34]. Figures 10(a) to (c) demonstrate how one operation can be moved past another one while dynamically eliminating data dependencies. In Figure 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 of the new destination variable y' to the original variable y. Similarly, in Figure 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.

Copy operations introduced by dynamic renaming 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 Figure 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.

Dynamic renaming and combining, when performed in conjunction with code motion techniques such as trailblazing and percolation, can lead to considerable easing of the constraints imposed by data dependencies. In the next section, we present a scheduling heuristic that uses the various passes in the Spark toolbox and generates a schedule under resource constraints.

4.4 Priority-based Global List Scheduling Heuristic

Scheduling is the task of assignment of operations to control steps or time intervals so that the allocated resources can compute the operations assigned to each step [8]. 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 applied to other scheduling heuristics as well. Priority list scheduling works by ordering operations to be scheduled based on a *priority* or cost associated with them.

Our objective is to minimize the *longest delay* through the design; hence, priorities are assigned to each operation based on their distance, in terms of the data dependency chain, from the primary outputs of the design. The priority of an operation is calculated as one more than the maximum of the priorities of all the operations that use its result. The algorithm starts by assigning operations that produce outputs a priority of zero, and hence, operations whose results are inputs to outputs have a priority of one and so on. The priority assignment of operations for the waka benchmark is shown in Figure 7. The priority assignment can also be changed to minimize a different cost function, such as average delay.

The scheduling heuristic is presented in Figure 11(a). The inputs to this heuristic are the unscheduled hierarchical task graph (HTG) of the design and the list of resource constraints. Additionally, the designer may specify a list of allowed code motions (i.e. speculation, reverse speculation, conditional speculation et cetera), whether dynamic variable renaming is allowed, and the code motion technique (percolation or trailblazing) for moving the operations. The heuristic starts by assigning a priority to each operation in the input description as explained above. Then scheduling is done one control or scheduling step at a time while traversing the basic blocks in the

Algorithm 1: Priority List Scheduling Heuristic
Inputs: Unscheduled HTG of design, Resource List R
Output: Scheduled HTG of design
1: Calculate Priority Pr of all Operations in HTG
2: Scheduling step $step = 0$
3: **while** ($step \neq$ last step of HTG) **do**
4: **foreach** (resource res in Resource List R) **do**
5: Get List of Available Operations $\mathcal{A}$
6: Pick Operation op with lowest cost in $\mathcal{A}$
7: Move op and schedule on res in $step$ with user-specified $CodeMotionTechnique$
8: **endforeach**
9: $step = step + 1$
10: **endwhile**
(a)

Algorithm 2: Get List of Available Operations
Inputs: Resource res, Scheduling $step$, $AllowedCodeMotions$
Output: Available Operations List $\mathcal{A}$
1: Candidates $\mathcal{A}$ = all unscheduled ops U in HTG that can be scheduled on resource res
2: **foreach** (op in $\mathcal{A}$) **do**
3: **if** (data dependencies of op cannot be satisfied) remove op from $\mathcal{A}$
4: **if** (op cannot be moved to $step$ using $AllowedCodeMotions$) remove op from $\mathcal{A}$
5: Calculate cost of operation op
6: **endforeach**
(b)

Figure 11. **(a) Priority-based List Scheduling Heuristic (b) Determining the list of *Available* operations.**

hierarchical task graph (HTG). In our implementation, control paths are followed such that at the fork node of a conditional block, the true branch is scheduled first and then the false branch. Within a basic block, each scheduling step corresponds to a statement HTG node in the basic block (see Section 4.1). So, at each time step in the basic block, a list of *available* operations is collected, for each resource in the resource list, as shown in line 4 in the algorithm in Figure 11(a).

Available operations is a list of operations that can be scheduled on the given resource at the current scheduling step. Pseudo-code for collecting the list of available operations is given in Figure 11(b). Initially, all unscheduled operations in the HTG that can be scheduled on the current resource type are added to the available operations list. Subsequently, operations whose data dependencies are not satisfied and cannot be satisfied by variable renaming are removed from this list. Similarly, operations that cannot be moved in the HTG to schedule them onto the current scheduling step using the allowed code motions are also removed from the available list. The list of allowed code motions is provided by the user and hence, allows experimentation with various kinds of code motions. The algorithm then assigns a cost to each remaining operation in the available list. Currently, this cost is based on the operations global priority.

Once the scheduling heuristic has received the list of available operations, it picks the operation with the *lowest*

cost from the list as shown in line 6 of Figure 11(a). The code motion technique is then instructed to schedule this operation at the current scheduling step. This is repeated for all resources in each scheduling step as the basic blocks in the HTG are traversed from top to bottom. Operations left unscheduled at the end of a basic block are moved down into the next basic block or reverse speculated into the conditional branches, as the case may be.

Scheduling of loops is done in the same manner; however, loops are scheduled first in the design. The scheduler traverses the design graph till it finds the innermost loop, schedules this first (after applying any user-specified loop transformations), and then schedules the next outer loop nest and so on. Scheduling of the rest of the design, then proceeds starting at the first node in the HTG. The Spark system can schedule all types of loops, including those with unknown loop iteration bounds. This is because, in the finite state machine (FSM) generated by Spark, at the end of a loop body iteration, the FSM either goes back to the first state in the loop body or goes to the next 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 not known, we cannot estimate the cycles the loop will take to execute and hence, for these kind of designs, we cannot present the number of cycles of execution in our results.

4.5 Determining the Application of the Code Motions

There is no fixed order of application of code motions; it depends upon the individual operation and the current scheduling step. When the available list is being constructed, the scheduler calls a code motion technique such as *trailblazing* or *percolation*, which determines all the code motions required to move the operation from its current position to the current scheduling step. It then compares these *required* code motions with the list of *AllowedCodeMotions*, which are user-specified. So, if a particular code motion, say speculation, is required to move the operation to the current scheduling step, say because it has to move out of a conditional branch, and this code motion is not in the *AllowedCodeMotions* list, then the operation is not included in the available list.

The user can specify which of the code motions are enabled (*AllowedCodeMotions*) via a script file that is read by Spark. We use this scripting ability to see how the synthesis results are affected when various code motions are enabled and disabled. Hence, once a code motion is enabled in the script file, the scheduler applies this code motion *automatically* without any further user intervention, as and when required.

Experimental results have shown us that, when code motions such as conditional speculation that lead to operation duplication, are applied unchecked, they can lead to high overheads and increased schedule lengths. Hence, currently we have developed some heuristics to control conditional speculation (CS). A heuristic that determines if an operation *op* should be conditionally speculated is outlined in Figure 12. This heuristic starts with the list of basic blocks (*BBList*) into which an operation *op* will have to be duplicated, if it were to be scheduled on

Algorithm 3: Test if Conditional Speculation should be applied

Inputs: List of basic blocks *BBList* to which the operation op will be duplicated if it is scheduled at scheduling $step$ in BB_{step}

Output: Whether op should be condionally speculated

1: Initialize: allowConditionalSpeculation = true

2: **foreach** (Basic block *bb* in *BBList*) **do**

3: **if** (isThereEmptyResourceInBB(*bb*, op) == false)

4: willHaveToCreateNewStatementInBBForOp = true

5: **if** (numOfStmtsInBB(*bb*) $\geq$ numOfStmtsInBB(BB_{step})) then

6: **if** (willHaveToCreateNewStatementInBBForOp == true)

7: allowConditionalSpeculation = false

8: **else if** (isBBScheduled(*bb*) == false)

9: allowConditionalSpeculation = false

10: **if** (allowConditionalSpeculation == false)

11: return from function with **false** result

12: **endforeach**

13: return from function with **true** result

Figure 12. **Heuristic to determine whether to conditionally speculate an operation op into multiple basic blocks given by *BBList*, while scheduling it into scheduling $step$ in basic block BB_{step}.**

scheduling step, $step$, in basic block BB_{step}.

The heuristic goes through every basic block *bb* in the list *BBList* and for each *bb*, it checks if there is an idle resource to schedule operation op on (line 3 in Figure 12). If there is no idle resource in *bb*, then the heuristic sets a flag saying that a new statement[2] will have to be created in the basic block *bb* to accomodate operation op. Next, the heuristic determines if the current basic block *bb* already has as many or more statements than BB_{step} *and* a new statement will have to be created to accomodate op, then CS is not allowed (lines 5 to 7 in the algorithm). This is to prevent basic block *bb* from becoming the critical path in the design with the most number of statements (or scheduling steps) among its mutually exclusive basic blocks. This implies that the heuristic do not allow CS if it leads to an increase in the maximum number of cycles through an if-then-else conditional HTG node. So in the example in Figure 5, if CS would lead to an additional cycle being required in either the "true" (BB_1) or the

[2]Each statement in a basic block translates into a scheduling step within the basic block (see Section 4.1)

“false” (BB_2) basic blocks, then the code motion would not be allowed.

The heuristic also does not allow CS when the *bb* already has as many or more statements than BB_{step} *and* the basic block *bb* has not been scheduled yet (lines 5, 8 and 9 in Figure 12). This is because without scheduling the basic block *bb* first, it is not possible to accurately determine if there is an idle resource in *bb* on which to schedule operation op. This can be demonstrated with the example in Figure 5; we would allow consideration of operations that require CS only while scheduling basic block BB_2 (after BB_1 has been scheduled). If we apply conditional speculation while scheduling basic block BB_1, then the operation duplication into basic block BB_2 may lead to an extra cycle being added, since the resource usage of unscheduled basic blocks cannot be determined.

The current cost model depends only on the global priority of the operation, but ignores the control and multiplexor costs associated with scheduling an operation on a functional unit. A more global notion of operation scheduling costs is required, so that undesirable code motions or scheduling decisions leading to increased interconnect costs are assigned a higher cost. In this work, we have used this simple cost model to gather more information about the effects and effectiveness of the various code motions. The results of these experiments are presented in the next section and can be used to develop more realistic cost models.

5 Effects of Code Motions on Quality of Synthesis Results

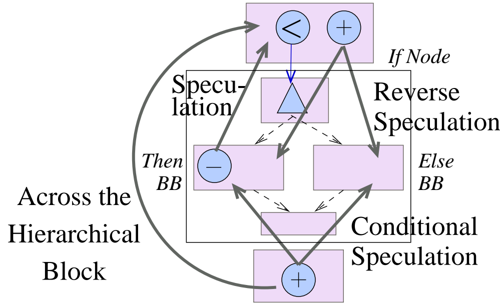
We now study the effects of the proposed code motions first on scheduling and controller synthesis results and then on logic synthesis results. We use two large realistic benchmarks for all the experimental results presented in this paper: the *Encoder* block from the ADPCM benchmark [35] and the *Prediction* block (a moderately control intensive block) from the MPEG-1 algorithm [36]. We present results for three out of five of the functions from the MPEG Prediction block, namely, $calc_forward_motion$, $pred0_1$ and $pred2$, which have 31, 45, and 26 non-empty basic blocks respectively. Of the remaining two functions, the $calcid$ function consists of a single basic block with no control flow and the $calc_backward_motion$ function is similar to the $calc_forward_motion$. The ADPCM Encoder has 38 non-empty basic blocks. The number of basic blocks gives an idea of the control complexity and size of the design.

5.1 Effects on Performance

We first study the effects of these code motions on the number of states in the finite state machine (FSM) and the cycles on the longest path in the design. These are presented in Tables 1 and 2 for the various functions in the two benchmarks. The number of states denotes the controller complexity and the longest path length is equivalent to the execution cycles of the design. For loops, the longest path length of the loop body is multiplied by the number of loop iterations, if this is known. The resources used are indicated in the two tables; *ALU* does add and subtract, $*$ is a multiplier, $==$ is a comparator, [ ] is an array address decoder and $<<$ is a shifter. The multiplier

MPEG Prediction Block; Resources = $3ALU, 2[\,], 3 <<, 2 ==, 1*$(2-cycle); BBs = non-empty Basic Blocks						
Type of Code Motion	*calc_forw* (73 Ops, 31 BBs)		*pred*2 (217 Ops, 45 BBs)		*pred*0_1 (101 Ops, 26 BBs)	
	# States	Long Path	# States	Long Path	# States	Long Path
Within basic blocks	37	37	182	6359	187	3072
+across hier blocks	28(-24%)	28(-24%)	157(-14%)	5956(-6%)	162(-13%)	2871(-7%)
+speculation	26(-7%)	26(-7%)	102(-35%)	4263(-28%)	137(-15%)	2177(-24%)
+early cond exec	24(-8%)	24(-8%)	100(-2%)	4261(-0%)	134(-2%)	2174(0%)
+cond speculation	22(-8%)	22(-8%)	92(-8%)	3945(-7%)	122(-9%)	1910(-12%)
Total Reduction	**40.5 %**	**43.2 %**	**49.5 %**	**38.0 %**	**34.8 %**	**37.8 %**

Table 1. **Scheduling and controller size results for the various code motions for the MPEG Pred block.**



Summary of various types of code motions

ADPCM Encoder: 65 Ops, 38 non-empty BBs		
Type of Code Motion	$1ALU, 2 ==, 2[\,], 1 <<$	
	# States	Long Path
Within basic blocks	33	313
+across hier blocks	28(-15%)	273(-13%)
+speculation	26(-7%)	253(-7%)
+early cond exec	24(-8%)	233(-12%)
+cond speculation	16(-33%)	152(-35%)
Total Reduction	**51.5 %**	**51.4 %**

Table 2. **Scheduling and controller size results for the various code motions for the ADPCM Encoder.**

is a 2-cycle resource and all other resources have single cycle execution time. These tables also show the number of non-empty basic blocks and operations in the design.

The Spark system treats function calls as resources and creates a functional unit corresponding to them in hardware. For example, the function *calc_forward_motion* is called from both the functions *pred*0_1 and *pred*2, and hence, is a component or functional unit that is embedded in these hardware blocks. Hence, called functions contribute towards the schedule length and number of states in the controller of the calling function.

The rows in Tables 1 and 2 present results with each code motion enabled incrementally, i.e., these signify the allowed code motions while determining the available operations (see Section 4.4) and do *not* represent an ordering of code motions. We first allow code motions only within basic blocks (first row) and then, in the second row, we also allow code motions across hierarchical blocks, i.e., across entire if-then-else conditionals and loops.

The third row further allows speculation, the fourth row has early condition execution enabled as well and the final row has the conditional speculation code motion also enabled. The percentage reductions of each row over the previous row are given in parentheses. Typical run times of Spark to produce these results is in the range of 5 user seconds (0.2 kernel seconds) on a Sun Ultra 250 running at 400 Mhz.

As each code motion is enabled, we see significant reductions in both the number of FSM states and the cycles on the longest path. Code motions across hierarchical blocks, speculation out of conditionals and conditional speculation by far are the most effective code motions. The results demonstrate that early condition execution, which uses reverse speculation to move operations down into conditional branches, can lead to improvements in some cases. However, these improvements vary with each function in the benchmarks. Similarly, the results for the various code motions also demonstrate that the nature of the benchmark functions dictates what code motions are most effective on them. For example, the fifth row in Table 2 demonstrates that enabling conditional speculation leads to reductions of over 36 % both in the number of states and the longest path cycles for the ADPCM encoder. However, the corresponding row in Table 1 shows reductions of only between 5-15 % for the MPEG algorithm. This is because the ADPCM benchmark is highly control intensive with nearly as many conditional checks as operations. Hence, moving operations into its conditional branches significantly improves resource utilization. The functions in the MPEG Prediction block on the other hand have a more mixed distribution of data and control operations. Hence, the improvements due to the various code motions are more uniform for the MPEG functions, as evident from Table 1.

These observations, and the results in Tables 1 and 2, demonstrate that the effectiveness of a particular code motion is heavily dependent on the characteristics of the behavioral description being synthesized. Control-intensive designs (such as $calc_forw$ and ADPCM) benefit more from code motions that move operations into conditional branches (such as early condition execution and condition speculation), whereas designs that have more data operations than conditionals (such as $pred0_1$ and $pred2$) benefit more from code motions such as speculation. We also note that opportunities for conditional speculation increase with increasing resources, leading to up to 30 % reductions for the MPEG benchmark. This indicates that resources are most idle within conditional branches, especially as more resources are allocated.

Tables 1 and 2 show that these kind of speculative code motions lead to substantial improvements in the latency of the design and complexity of the controller. The total reduction in execution cycles and number of states achieved with all the transformations enabled over code motion only within basic blocks ranges between 35 % to 51 % (last row in the tables). Note that, when code motions only within basic blocks are enabled, the priority-list scheduling heuristic we have presented reduces to the classical list scheduling approaches presented in previous

Benchmark	Number of Basic Blocks	Resources	Schedule Length				
			CVLS [13]	HRA [37]	Radivojevic [14]	Santos [3]	Spark
kim [37]	7	2+,1-,2==	6	7	6	6	6
parker [38, 39]	20	2+,3-,5==	4	NA	4	4	4
waka [13]	9	1+,1-,2==	7	7	7	7	7
rotor [14]	11	2+-,2*,1[ ]	NA	NA	8	8	8

Table 3. **Comparison of schedule lengths with other methods using classical high-level synthesis benchmarks. NA represents results that are not available. The scheduling results are optimal for these benchmarks.**

works [8, 20].

As stated earlier, most of the benchmarks used to present results in previous high-level synthesis literature are small and many of them are synthetic. However, to provide a comparison of Spark with these works, we present scheduling results of several previous approaches along with our system, Spark, in Table 3. The comparisons are made with the CVLS approach [13], the HRA approach [37], the *exact* approach presented by Radivojevic [14] and the approach presented by Santos et al. [3]. These comparisons have been made using classical high-level synthesis benchmarks. The benchmarks are: *kim* from [37], *parker* from [38, 39], *waka* from [13] and *rotor* from [14]. 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. The results in this table show that for these benchmarks, the Spark system produces scheduling results that are as good as those produced by the other systems. These scheduling results are optimal for these benchmarks. Whereas the runtime for the exact approach by Radivojevic on the "rotor" benchmark is in the order of 13.7 seconds on a SUN Sparc Station 10 (probably running at 33 or 66 Mhz), the runtime for the Spark system for the same benchmark is in about 0.13 seconds on a 170 Mhz SUN Sparc Station 5.

5.2 Effects on Area and Clock Period

Although aggressive code motions lead to significant reductions in the execution cycles of a design, their overall effects on synthesis results should take into account the associated costs of control logic. These are not obvious until the design is synthesized. Hence, to further evaluate the effects of the various types of code motions, we synthesized the register-transfer level (RTL) VHDL generated after scheduling by the Spark synthesis system, using the Synopsys *Design Compiler* logic synthesis tool. The LSI-10K synthesis library was used for technology mapping.

The results after logic synthesis are summarized in Figure 13 for the $pred2$, $pred0_1$ and the $calc_forw$ func-

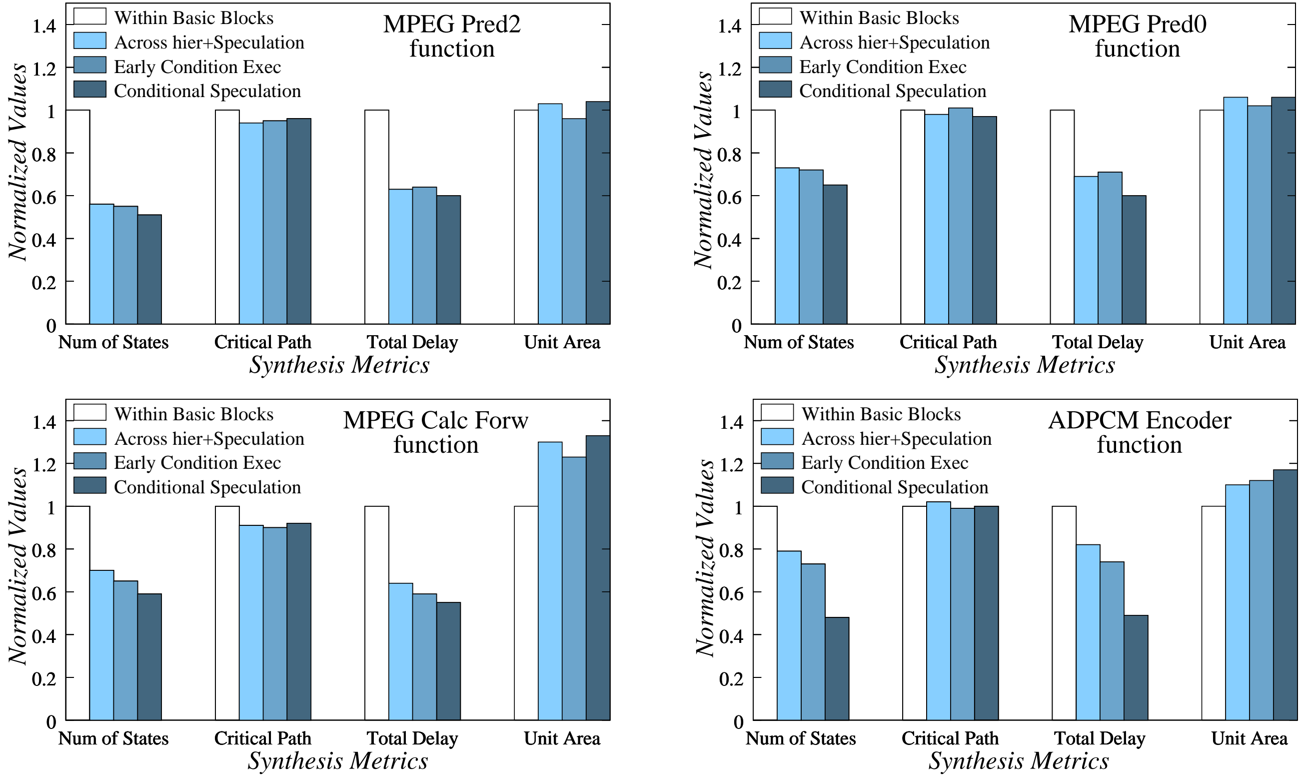


Figure 13. **Logic synthesis results after various code motions for the MPEG** $pred_case2$**,** $pred_case0_1$ **and** $calc_forward$ **functions and the ADPCM Encoder function; circuit delay decreases significantly but area can increase marginally.**

tions of the MPEG Prediction block and for the Encoder function of the ADPCM algorithm. In these graphs, four metrics are mapped: the number of states in the FSM, the critical path length (in nanoseconds), the unit area and the maximum delay through the design. The critical path length is the length of the longest combinational path in the netlist as determined by static timing analysis. The critical path length dictates the clock period of the final design. The unit area is in terms of the synthesis library used (the LSI-10K library). The maximum delay is the product of the longest path length (in cycles) and the critical path length (in ns) and signifies the maximum input to output latency of the design.

These four metrics are mapped for, code motions allowed only within basic blocks, then with across hierarchical block code motions and speculation also allowed, with early condition execution as well and finally with conditional speculation allowed too. The values of each metric are normalized by the metric's value when code motions are allowed only within basic blocks. We synthesized these designs with a arbitrary binding of operations to functional units so as to ensure that the number of resources synthesized are as per the resources allocated during scheduling by the high-level synthesis tool.

These graphs demonstrate that as we apply more and more aggressive code motions, the size of the controller (number of states) decreases and the performance of the design increases, i.e. total delay decreases. These values are almost halved when all the code motions are enabled over when code motions only within basic blocks are allowed. For the early condition execution transformation, even though the scheduling results presented in the previous section showed no improvement in some cases, we find that the synthesis results are usually better, especially in terms of the area of the circuit. This is because this transformation moves operations down into conditional branches and at times only into those branches that need the results of the operation.

These graphs also demonstrate that the critical path length in the design remains fairly constant, while the area increases steadily. This area increase is due to increasing complexity of the steering logic and associated control logic caused by resource sharing. Critical paths also typically pass through this steering logic. A typical critical path in the synthesized designs is shown in Figure 14. It starts in the control logic that generates the select signals for the multiplexors connected to the functional units. The path continues through the multiplexors, through the functional unit and then through another multiplexor, that finally writes to the output register. As the resource utilization and sharing increases as a result of aggressive speculative code motions, the size of these interconnects (multiplexors and demultiplexors) gets increasingly large, leading to increased area.

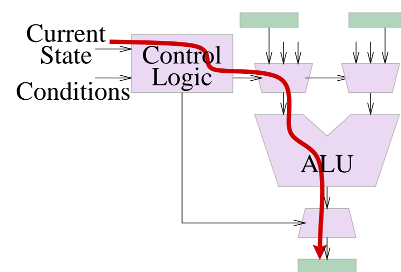


Figure 14. **Typical critical paths in control-intensive designs pass through the steering logic and the associated control logic.**

As noted above, the critical path length does not change significantly as more and more code motions are enabled. This is because although aggressive code motions affect critical path lengths adversely due to higher resource utilization and sharing, they also lead to reduced number of states in the FSM and shorter schedule lengths. This leads to smaller controllers that moderate the effects of the increased interconnect and effectively leads to negligible effect of the code motions on critical path lengths.

6 Reducing Interconnect

The very resource sharing that is leading to increases in circuit complexity, also provides an opportunity to minimize interconnect. Since the resources have several operations and variables mapped to them, there exist opportunities to reduce the number of inputs to, and hence, the complexity of, the (de)multiplexors between these resources by resource binding techniques. Fewer inputs not only mean smaller interconnects but also simpler associated control logic. The interconnect minimizing resource methodology implemented in the Spark framework attempts to first bind operations with the same inputs or outputs to the same functional unit. The variable to register

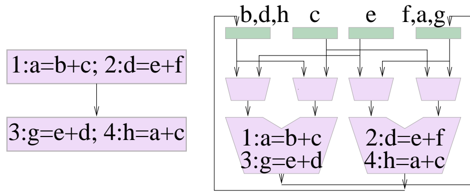


Figure 15. **An example of binding leading to a large number of interconnections.**

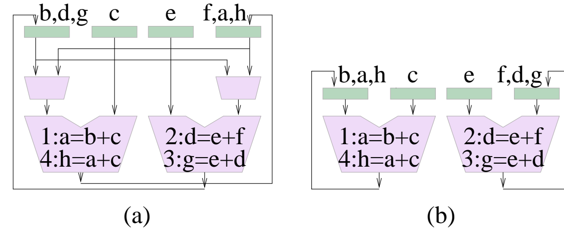


Figure 16. **Reducing interconnect by improved (a) operation binding (b) variable binding.**

binding then takes advantage of this by trying to map variables that are inputs or outputs to the same functional units to the same register. In this way, the number of registers feeding the inputs and storing the outputs of functional units is reduced, in effect, reducing the size of the multiplexors and demultiplexors connected to the functional units. The following sections describe an operation and variable binding methodology to minimize these interconnect and control costs.

6.1 Operation to Functional Unit Binding

The number of interconnections required to connect units to each other and to registers can be reduced by combining operations that have the same inputs and/or same outputs. This can be intuitively understood by considering the classical example of binding and resultant hardware shown in Figure 15 [8]. The interconnect shown in this circuit can be simplified by exchanging the functional units that operations 3 and 4 are bound to, as shown in Figure 16(a). This is because operations 1 and 4 have the input variable *c* in common and operations 2 and 3 have variable *e* in common.

So, the operation binding problem can be defined as follows: given a scheduled design and a set of resource constraints, map each operation to a functional unit from among the given resources, such that the interconnect is minimized. We formulate this problem by creating an operation compatibility graph for each type of resource in the resource list. Each operation in the design that can be mapped to the resource type under consideration has a node in the graph. *Compatibility edges* are created between nodes corresponding to operations that are scheduled in either different control steps or execute under a different set of conditions. Note that, mutually exclusive operations (and their variables) scheduled in the same time step are compatible with each other.

For reducing interconnect, we add additional edge weights between operations for each instance of common inputs or outputs between them. Then, a maximally weighted clique cover of this graph will lead to binding that reduces interconnect. However, the constraint on the number of resources means that the number of cliques cannot exceed the number of resources of each type. To solve this problem, we formulate it as a multi-commodity network flow problem. A max-cost flow through this multi-commodity network represents a valid maximally

weighted clique cover [40, 41]. We note that Chang et al. [40] use the same formulation for module allocation but their objective is to minimize power consumption.

6.2 Variable to Register Binding

Variable to register binding can take advantage of the improved operation binding by mapping variables that are inputs or outputs to the same port of the same functional unit to the same registers. For example, the result obtained after operation binding shown in Figure 16(a) can be further improved by changing the variable binding as shown in Figure 16(b). Here, the binding of the variables d and a has been switched, so that variables b and a which feed the same input of the first adder are mapped to the same register and f and d which feed the second adder are mapped to the same register. Similarly, the binding of the output variables g and h has been switched too.

The formulation of this problem is similar to the operation binding problem, except that we do not place a constraint on the number of registers. A compatibility graph is created with a node corresponding to a write to a variable in the design. Compatibility edges are added between nodes corresponding to variables that do not have overlapping lifetimes or are created under a different set of conditions.

Additional edge weights are added between variables for each instance of them being inputs or outputs to the same port of the same functional unit. A maximally weighted clique cover of this graph represents a valid variable to register binding with minimal interconnect. This is solved by formulating it as a min-cost max-flow network problem. Similar approaches to solve this clique problem have been used in [23] and [42]. Details of the implementation of the binding methodology can be found in [43].

7 Results of Resource Binding

To validate the interconnect minimizing methodology presented above, we synthesized the MPEG and ADPCM designs, using a non-interconnect aware resource binding technique and using the interconnect minimizing binding technique. For the rest of this section, we refer to the non-interconnect aware resource binding methodology as the "regular" binding. We have been unable to compare our results to any previous work because of a number of reasons; past work on reducing interconnect by resource binding has used purely data-oriented DSP benchmarks (such as elliptic wave filter, et cetera) for validating their techniques [23, 25]. Our resource binding methodology specifically targets designs with a moderate mix of data and control operations, with the objective of reducing the large multiplexors that characterize these type of designs. Also, recent work in binding for high-level synthesis has concentrated on reducing power or improving testability by resource binding [25, 42]. Comparison with previous work is further complicated by the fact that binding results are often presented in terms of absolute area numbers or registers required by the design. Area of the netlist depends on the synthesis library and the logic synthesis

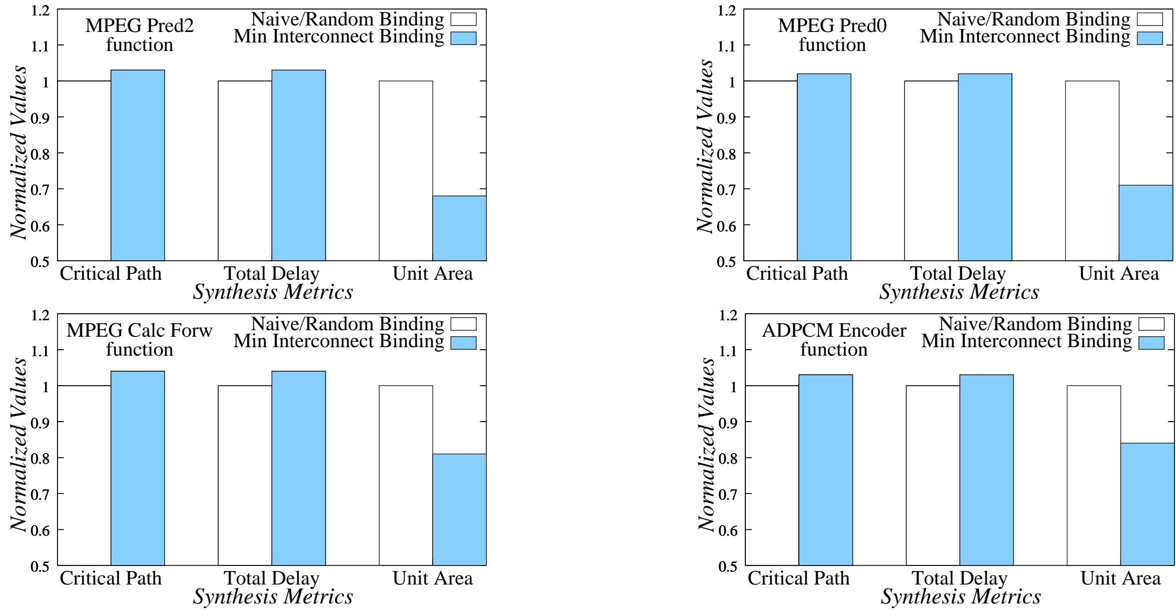


Figure 17. **Results of logic synthesis after applying a non-interconnect aware ('regular') binding and an interconnection minimizing resource binding for the MPEG** $pred_case2$**,** $pred_case0_1$ **and** $calc_forward$ **functions and the ADPCM Encoder function.**

tool used and our methodology does not attempt to reduce the number of registers, but rather the interconnect requirements of the design.

Hence, we compare our results with a non-interconnect aware binding methodology. The graphs in Figure 17 present the synthesis results for the $pred2$, $pred0_1$ and the $calc_forw$ functions of the MPEG Prediction block and for the Encoder function of the ADPCM algorithm obtained with regular resource binding and the interconnect minimizing resource binding. The metrics compared in these graphs are the critical path lengths, total delay and the unit area. The values for each metric in these graphs are normalized to the value with regular binding. The results are obtained with all code motions enabled.

The reductions in area due to the interconnect minimizing binding over the regular binding are significant; the area reduces by 32 % for the $pred2$ function and atleast by 15 % for all other designs. These improvements are despite the fact that in our interconnect minimization strategy, we sometimes choose to allocate more registers if this leads to a reduction in the steering logic. Hence, the reductions in interconnect complexity dominate any increases due to higher register requirements. Furthermore, we found that although synthesis using a binding methodology, which minimizes only registers, leads to fewer registers, the total area of the design is worse, since the interconnect sizes are not reduced.

The graphs show that critical path lengths remain fairly constant in all the designs, hence, barely leading to any changes in delay through the circuit. These results demonstrate that the interconnect methodology is able to

achieve more area efficient designs without sacrificing performance.

For both, the interconnect minimizing and the non-interconnect aware binding results, we find that applying the conditional speculation transformation can lead to an increase in the critical path lengths of the design and always leads to an increase in area of the synthesized circuit. As can be seen for the graph of the MPEG $calc_forw$ function, the total delay can also increase over the previous value (after early condition execution). This goes to show that these code motion transformations have to be judiciously used by the scheduling heuristic by taking into account additional control costs that may be incurred due to higher resource utilization and sharing. Hence, more accurate control cost estimation models have to be developed and used during scheduling in high-level synthesis.

8 Conclusions and Future Work

In this paper, we have presented a set of speculative code motions that re-order, speculate, and sometimes even increase the number of operations in a behavioral description so as to achieve higher quality of synthesis results. These code motions of operations are essential to minimize the effects of syntactic variance caused by programming style in high level languages. Scheduling results after applying these code motions to moderately complex real-life benchmarks show improvements of up to 50 % in performance and reduction in controller size when compared to list scheduling techniques that allow code motions only within basic blocks. Logic synthesis results show similar reductions in the total delays through the circuits. Furthermore, we demonstrate that the control and interconnect overheads incurred due to these code motions can be reduced by resource binding targeted at interconnect minimization. This methodology leads to area reductions between 15 % to 32 %. We have also described the Spark high-level synthesis framework in which the various transformations presented in this paper are implemented. This framework provides a platform for applying a range of coarse-grain and fine-grain code optimizations aimed at improved synthesis results.

Future work entails working on developing more comprehensive cost models for the various code motions. This is necessary especially for code transformations, such as conditional speculation and loop unrolling, that change the number of operations in the design. Uncontrolled application of conditional speculation can lead to significantly poorer results, since the number of operations on the longest path may increase due to operation duplication. Furthermore, when scheduling an operation on a functional unit, the effects of this decision on interconnect (multiplexors et cetera) must be taken into account and be included in the cost of the code motion.

References

[1] K. Wakabayashi. C-based synthesis experiences with a behavior synthesizer, ”Cyber”. In *Design, Automation and Test in Europe*, 1999.

[2] Get2Chip Incorporated. Volare multi-level synthesis.

[3] L.C.V. dos Santos. *Exploiting instruction-level parallelism: a constructive approach*. PhD thesis, Eindhoven University of Technology, 1998.

[4] S. Haynal. *Automata-Based Symbolic Scheduling*. PhD thesis, University of California, Santa Barbara, 2000.

[5] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Wavesched: a novel scheduling technique for control-flow intensive designs. *IEEE Transactions on CAD*, May 1999.

[6] S. Gupta, N. Savoiu, S. Kim, N.D. Dutt, R.K. Gupta, and A. Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Design Automation Conference*, 2001.

[7] 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 *Intl. Symp. on System Synthesis*, 2001.

[8] D. D. Gajski, N. D. Dutt, A. C-H. Wu, and S. Y-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[9] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *International Conference on Parallel Processing*, 1993.

[10] S. Novack and A. Nicolau. An efficient, global resource-directed approach to exploiting instruction-level parallelism. In *Conference on Parallel Architectures and Compilation Techniques*, 1996.

[11] M. Potkonjak and J. Rabaey. Optimizing resource utlization using tranformations. *IEEE Trans. on CAD*, March 1994.

[12] R. Walker and D. Thomas. Behavioral transformation for algorithmic level IC design. *IEEE Trans. on CAD*, Oct. 1989.

[13] K. Wakabayashi and H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. In *Design Automation Conference*, 1992.

[14] I. Radivojevic and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE Transactions on CAD*, January 1996.

[15] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. In *Design Automation Conference*, 1998.

[16] L.C.V. dos Santos and J.A.G. Jess. A reordering technique for efficient code motion. In *Design Automation Conf.*, 1999.

[17] M. Rim, Y. Fann, and R. Jain. Global scheduling with code-motions for high-level synthesis applications. *IEEE Transactions on VLSI Systems*, September 1995.

[18] R.A. Bergamaschi. Behavioral network graph unifying the domains of high-level and logic synthesis. In *Design Automation Conference*, 1999.

[19] R. Camposano and W. Wolf. *High Level VLSI Synthesis*. Kluwer Academic, 1991.

[20] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[21] C.J. Tseng and D.P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, July 1986.

[22] P. G. Paulin and J. P. Knight. Scheduling and Binding Algorithms for High-Level Synthesis. In *Design Automation Conference*, 1989.

[23] L. Stok and W.J.M. Philipsen. Module allocation and comparability graphs. In *IEEE International Sympoisum on Circuits and Systems*, 1991.

[24] C.H. Gebotys and M.I. Elmasry. Optimal synthesis of high-performance architectures. *IEEE Journal of Solid-State Circuits*, March 1992.

[25] A. Mujumdar, R. Jain, and K. Saluja. Incorporating performance and testability constraints during binding in high-level synthesis. *IEEE Trans. on CAD*, 1996.

[26] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, July 1981.

[27] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *International Conf. on Parallel Processing*, 1985.

[28] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *3rd International Conference on Supercomputing*, 1989.

[29] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[30] A. Nicolau. A development environment for scientific parallel programs. Technical Report TR 86-722, Department of Computer Science, Cornell University, 1985.

[31] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles and Techniques and Tools*. Addison-Wesley, 1986.

[32] Synopsys Incorporated. Design compiler.

[33] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on Parallel & Distributed Systems*, Mar. 1992.

[34] S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *International Symposium on Microarchitecture*, 1992.

[35] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, 1997.

[36] Spark Synthesis Benchmarks FTP site. ftp://ftp.ics.uci.edu/pub/spark/benchmarks.

[37] T. Kim, N. Yonezawa, J.W.S. Liu, and C.L. Liu. A scheduling algorithm for conditional resource sharing - a hierarchical reduction approach. *IEEE Transactions on CAD*, April 1994.

[38] A.C. Parker, J. Pizarro, and M. Mlinar. MAHA: A program for datapath synthesis. In *Design Automation Conference*, 1986.

[39] 1991 High-Level synthesis design repository. ftp://ftp.cecs.uci.edu/pub/hlsynth/HLSynth91, 1991.

[40] J.-M. Chang and M. Pedram. Module assignment for low power. In *European Design Automation Conference*, 1996.

[41] L. Stok. Transfer free register allocation in cyclic data flow graphs. In *European Conf. on Design Automation*, 1992.

[42] J.-M. Chang and M. Pedram. Register allocation and binding low power. In *Design Automation Conf.*, 1995.

[43] 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. Technical Report ICS-TR-01-25, UC Irvine, 2001.