Conditional Speculation and its Effects on Performance and Area for High-Level Synthesis*

Sumit Gupta Nick Savoiu Nikil Dutt Rajesh Gupta Alex Nicolau

Center for Embedded Computer Systems
Dept. of Information and Computer Science
University of California at Irvine
http://www.cecs.uci.edu/~spark
{sumitg,savoiu,dutt,rgupta,nicolau}@cecs.uci.edu

ABSTRACT

We introduce a code transformation technique, "conditional speculation", that speculates operations by duplicating them into preceding conditional blocks. This form of speculation belongs to a class of aggressive code motion techniques that enable movement of operations through and beyond conditionals and loops. We show that, when used during scheduling in a high-level synthesis system, this particular code motion has positive effect on latency and controller complexity, e.g., up to 35 % reduction in longest path cycles and the number of states in the finite state machine (FSM) of the controller. However, it is not enough to determine complexity by the number of states in the control FSM. Indeed, the greater resource sharing opportunities afforded by speculation actually increase the total control cost (in terms of multiplexing and steering logic). This also adversely affects the clock period. We examine the effect of the various code motions on the total synthesis cost and propose techniques to reduce costs to make the transformations useful in real-life behavioral design descriptions. Using the MPEG-1 and ADPCM benchmarks, we show total reductions in schedule lengths of up to 50 % while keeping control and area costs down.

1. INTRODUCTION

High-level synthesis is the automated synthesis of a digital design from its behavioral description [1, 2]. There has been a large body of research on high-level synthesis (HLS) which has concentrated on reducing schedule lengths of a design by improved scheduling techniques. However, the presence of complex control flow significantly affects the quality of synthesis results. Hence, several beyond-basic-block code motion techniques such as speculation have been used 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. Conversely, in reverse speculation, which we introduced in previous work [3], operations before conditionals are moved into *subsequent* conditional blocks and hence, executed conditionally. In this paper, we present a variant of speculation where an operation from after the conditional block is duplicated *up* into preceding conditional branches and executed conditionally. Hence, this code motion is called *conditional speculation*.

These kind of aggressive generalized code motions lead to significant reductions in schedule lengths and controller complexity. However, the effects of code transformations on synthesis costs, particularly control and multiplexing costs are poorly understood. In order to better understand these effects, we have taken the VHDL generated after scheduling by our high-level synthesis system and synthesized it using logic synthesis tools. We find that if left unchecked, these code motions can lead to substantial area and clock period overheads.

This leads us to believe that although these type of speculative code motions are useful, there needs to be a judicious balance between when to speculate, reverse speculate or conditionally speculate. Based on our experiments, we have developed simple heuristic cost models for the code motions and use them to guide a priority-list scheduling algorithm. This heuristic uses information on which code motions are allowed, and the cost of each code motion, to determine the most favorable operation to schedule on a given resource at a given time step.

However, the higher resource utilization and sharing caused by these code motions still lead to higher area and control costs. The control-intensive nature of the "real-life" benchmarks we have considered, further adds to these costs due to increased resource sharing among mutually exclusive operations, which leads to larger interconnect and associated control logic. Interconnect, here, refers to the multiplexors and buses that connect components together.

To address the complexity of the interconnect, this paper also presents an interconnect minimization approach based on a resource binding methodology. This methodology attempts to first bind operations with the same inputs or outputs to the same functional unit. The variable to register binding then takes advantage of this by trying to map variables which 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. Our results demonstrate the effectiveness of this approach in reducing area costs without adversely effecting performance.

The rest of this paper is organized as follows. We first discuss related work and then present the conditional speculation code motion. Section 4 outlines the scheduling heuristic which guides the various code motions within a synthesis framework. Section 5 studies the effects of the various code motions on the quality of synthesis results. Finally, an interconnect minimization strategy is outlined along with a study of its effectiveness on synthesis results.

*This work is supported by the Semiconductor Research Corporation: Task I.D. 781.001

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.
Copyright 2001 ACM 1-58113-418-5/01/0010 ...$5.00.

2. RELATED WORK

Initial high-level synthesis work concentrated on data-flow designs and applied optimizations such as algebraic transformations, retiming and code motions across multiplexors for improved synthesis results [4, 5]. Subsequent work has presented speculative code motions for mixed control-data flow type of designs and demonstrated their effects on schedule lengths. CVLS [6] uses condition vectors to improve resource sharing among mutually exclusive operations. Radivojevic et al [7] present an exact symbolic formulation which generates an ensemble schedule of valid, scheduled traces. The "Waveschedule" approach [8] incorporates speculative execution into high-level synthesis (HLS) to achieve its objective of minimizing the expected number of cycles. Santos et al [9] and Rim et al [10] support generalized code motions, taken from software compilers, for scheduling in HLS.

However, most previous works compare the effectiveness of their algorithms in terms of only schedule lengths. This has prevented a clear analysis of the effects of scheduling and code motions on the area and latency of the final hardware generated, since control logic overheads are usually ignored. To this end, Rim et al [10] use an analytical cost model for interconnect and control while applying code motions and Bergamaschi [11] proposes the behavioral network graph to bridge the gap between high-level and logic-level synthesis.

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) [12, 13, 14]. Although the basic transformations (e.g. dead code elimination, copy propagation) can be used in synthesis as well, other transformations 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. However, in synthesis, the additional hardware costs associated with code motions must be taken into account while making scheduling decisions.

Reducing interconnect using binding techniques has also been studied before [1, 2]. Stok et al [15] use a network flow formulation for minimum module allocation while minimizing interconnect. Paulin et al [16] perform exhaustive weight-directed clique partitioning to find the lowest combined register and interconnect costs. Mujumdar et al [17] use a network flow formulation to bind operations and registers in each time-step one at a time.

However, several of these approaches have been tested using data dominated designs with little or no control flow. Many moderately complex benchmarks extracted from industrial design descriptions such as ADPCM and parts of MPEG are control-intensive designs. This adds a new dimension to the complexity of the problem due to the presence of mutually exclusive operations on different branches of conditionals which share resources. Code motions during scheduling also lead to higher resource utilization and code duplication. This leads to added control logic both in terms of control signal generation and interconnect complexity.

The contributions of this paper include a presentation of conditional speculation which is another code motion in a class of code motions presented earlier [3] followed by a comparative study of these code motions. Then, a simple scheduling heuristic which directs these code motions is outlined. Finally, an interconnect reduction methodology is presented to reduce the higher control costs due to these aggressive code motions. This work has been implemented within a high-level synthesis framework called *Spark* which provides a path from a behavioral input description down to synthesizable register-transfer level code.

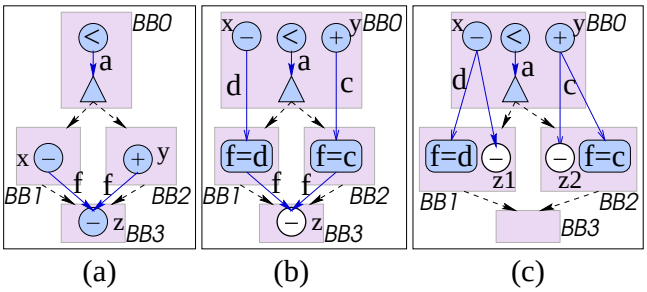


Figure 1: (a) A sample control-data flow graph (b) Operations *x* and *y* are speculated leaving idle slots in the conditional branches (c) Operation *z* is *conditionally speculated* into conditionals BB_1 and BB_2

3. CONDITIONAL SPECULATION

There are often instances in the input description 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 which lie in basic blocks *after* the conditional blocks. 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 [6].

Figure 1 demonstrates how such idle slots are created by speculation and how conditional speculation can be used to fill them. In the example in Figure 1(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 1(b). As shown in this figure, the results of the speculated operations are written into new destination variables, *d* and *c*, which 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 blocks.

Figure 1(b) demonstrates that the speculation of these operations leaves "idle" slots in which no operations have been scheduled on the resources. Furthermore, in this example, 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*, which lie in basic blocks after the conditional blocks, can be duplicated up or *conditionally speculated* into both branches of the conditional to fill idle slots as illustrated in Figure 1(c).

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

A scheduling heuristic which guides conditional speculation along with other code motions in the *Spark* high-level synthesis system is presented next.

4. PRIORITY-BASED GLOBAL LIST SCHEDULING HEURISTIC

For the purpose of evaluating the various code motion transformations, we have chosen a *priority-based* global list scheduling heuristic. Since we have chosen the objective of minimizing the *longest delay* through the design, hence, the priorities are assigned

MPEG Prediction Block; Resources = $3ALU, 2[\,], 3<<, 2==$,1*(2-cycle); BBs = non-empty Basic Blocks						
Type of	*calc_forw* (73 Ops, 31 BBs)		*pred*2 (217 Ops, 45 BBs)		*pred*0_1 (101 Ops, 26 BBs)	
Code Motion	# 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: Comparison of various types of code motion for the MPEG Pred benchmark

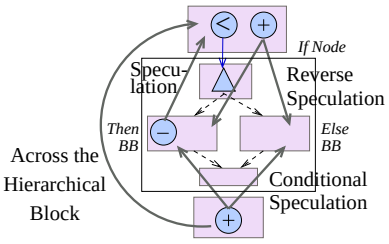


Figure 2: Various types of code motions

ADPCM Encoder(65 Ops, 38 non-empty BBs)		
Type of	$1ALU, 2==, 2[\,], 1<<$	
Code Motion	# 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: Comparison of various types of code motion for the ADPCM Encoder benchmark

to each operation based on their distance 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 with output operations having a priority of one [3].

The scheduling heuristic works as follows: first, a priority is assigned to each operation as explained above. Then scheduling is done one control (or scheduling) step at a time while traversing the control-data flow graph (CDFG). At each time step, for each idle resource, a list of *available* operations is collected [14]. Available operations are operations which can be scheduled on the current resource type and whose data dependencies are either satisfied or can be eliminated by dynamic renaming [18].

For each operation in the available operations list, the code motion technique determines if there is a path from the operation's current location to the scheduling step under consideration and the code motions that will be required to move the operation. The heuristic checks the code motions required to schedule the operation against a user specified list of code motions (i.e. speculation, reverse speculation, conditional speculation et cetera), and removes the operation from the available list if it requires a code motion that is not enabled.

The heuristic then assigns a cost to each operation based on metrics such as priority of the operation, cost of code motions and their effects on control logic. It then instructs the code motion technique to schedule the operation with the *lowest* cost from the list of remaining available operations. This is repeated for all resources in each scheduling step as the basic blocks in the CDFG 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.

Costs are assigned to each operation in the available list as:

$$Cost_{op} = -Priority * K_{CodeMotion1} * \ldots * K_{CodeMotionN}$$

where $K_{CodeMotion1}$ to $K_{CodeMotionN}$ are the multiplication factors ($<= 1$) associated with each code motion required to schedule the operation at the scheduling step under consideration. Since we found that except for conditional speculation, all the code motions are equally useful, we assigned them a multiplication factor of one. However, unchecked application of conditional speculation (CS) can lead to poor scheduling results since this code motion involves code duplication.

Hence, we developed a three-fold strategy to guide this code motion. Firstly, $K_{CodeMotion}$ for CS is set to 0.5. Next, we allow CS only after the "then" conditional branch has already been scheduled, i.e., in Figure 1, we would allow consideration of operations which require CS only while scheduling basic block BB_2 (after BB_1 has been scheduled). This gives us a better picture of available idle slots. Lastly, we do not allow CS if it leads to an increase in the maximum number of cycles through the conditional node. This means, in Figure 1, if CS would lead to an additional cycle being required in either the "then" (BB_1) or the "else" (BB_2) basic blocks, then the code motion would not be allowed.

These cost models have been developed as first-cut heuristic rules to guide the scheduling heuristic. The next section presents results which compare the effectiveness of the various code motions when guided by this scheduling heuristic.

5. EFFECTS OF CODE MOTIONS ON QUALITY OF SYNTHESIS RESULTS

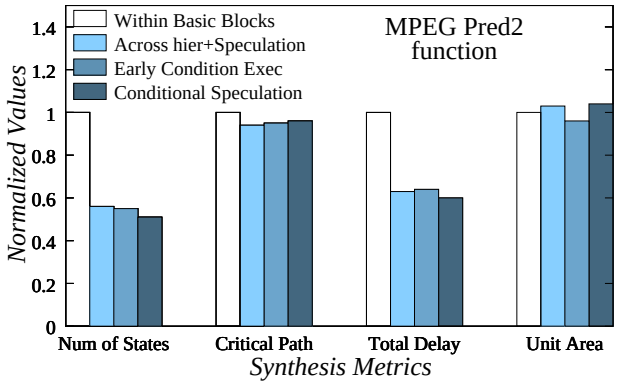
The various code motions presented earlier [3] along with conditional speculation are shown again in Figure 2. In addition to this, early condition execution is a code motion technique which executes conditional checks as soon as possible by reverse speculating unscheduled operations before the conditional, into the conditional's branches.

The two benchmarks used for the experiments in this paper are the *Encoder* block from the ADPCM algorithm and the *calc_forw*, *pred*0_1 and *pred*2 functions from the *Prediction* block of the MPEG-1 algorithm[1] [19].

5.1 Effects on Performance

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 are presented in Tables 1 and 2. The percentage reductions of each row over the previous row are in parentheses. 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. The resources used are indicated in the tables; *ALU* does add and subtract, $*$ is a multiplier, $==$ is a comparator, $[\,]$ is an array address decoder and $<<$ is a shifter. The multiplier is

[1]Although both benchmarks have loops, no loop transformations have been applied for these experiments



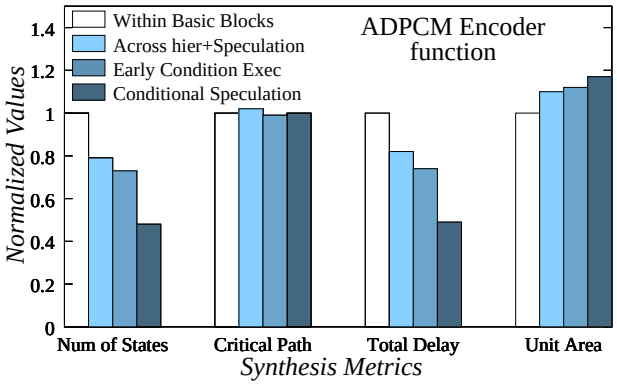


Figure 3: Effects of code motions on various metrics for the MPEG *pred2* and the ADPCM Encoder

a 2-cycle resource and all other resources have single cycle execution time. The number of non-empty basic blocks and operations are also given in these tables.

The rows in the tables present results with each code motion enabled incrementally, i.e., these signify the allowed code motions while determining the available operations (see Section 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 allows speculation too, the fourth row has early condition execution enabled as well and the final row has the conditional speculation code motion also enabled.

The fifth row in these tables demonstrate that enabling conditional speculation leads to reductions of 33-35 % in the number of states and the longest path cycles for the ADPCM encoder and between 7-12 % for the MPEG algorithm. This code motion is more effective for the ADPCM benchmark since this 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 is more uniform for the MPEG functions (see 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 Encoder) benefit more from code motions that move operations into conditional branches (such as early condition execution and condition speculation), whereas designs which have more data operations than conditionals (such as *pred*0_1 and *pred*2) benefit more from code motions such as speculation.

Also, while experimenting with different resource constraints, we found that opportunities for conditional speculation increase with increasing resources, leading to up to 30 % reductions for the MPEG benchmark. This goes towards showing that resources are most idle within conditional branches, especially as more resources are allocated.

These tables 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).

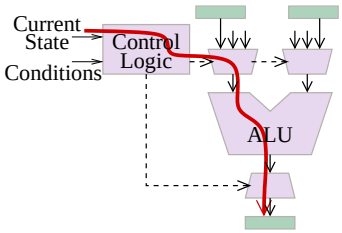


Figure 4: Typical critical paths in control-intensive designs

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 control costs. These are not obvious until the design is synthesized to the gate level. Hence, to further evaluate the effects of the various types of code motions, we synthesized the register-transfer level 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 for the MPEG *pred*2 function and ADPCM Encoder are presented in the graphs in Figure 3. 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.

The values of each metric are normalized by the lowest value for that metric. They 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 enabled too. We synthesized these designs with a simple, naive 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 system.

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.

These graphs also demonstrate that the critical path lengths in

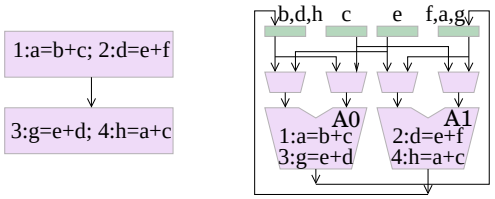


Figure 5: An example of binding leading to a large number of interconnections

the designs remain 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 4. 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, which finally writes to the output register. As the resource utilization and sharing increases, due to aggressive speculative code motions, the size of these interconnects (multiplexors and demultiplexors) gets increasingly large, leading to increased area.

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 which counter balance the effects of the increased interconnect and effectively leads to negligible affect 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 next two sections describe a resource binding methodology to minimize these interconnect and control costs.

6.1 Operation to Functional Unit Binding

The number of interconnections required to connect modules to each other and to registers can be reduced by combining operations which 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 5 [1]. The interconnect can be simplified by exchanging the functional units that operations 3 and 4 are bound to, as shown in Figure 7(a). This is because operations 1 and 4 have the input variable *c* in common and operations 2 and 3 have *e* in common.

Hence, the operation binding problem can be defined as follows: given a scheduled control data flow graph (CDFG) 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 of the resource type under consideration has a node in the graph. Compatibility edges are created between nodes corresponding to operations which are scheduled in either different control steps or execute under a different set of conditions.

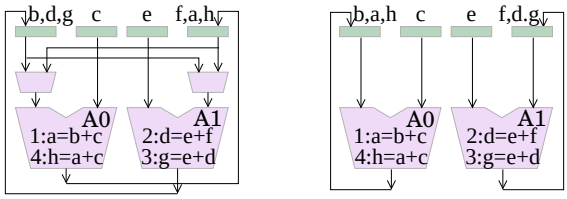


Figure 7: Reducing interconnections by improved (a) operation binding (b) variable binding

This means 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. A maximally weighted clique cover of this graph will lead to binding that reduces interconnect. 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 and find the max-cost flow [20, 21]. Chang et al [20] 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. Hence, the result obtained after operation binding shown in Figure 7(a) can be further improved by changing the variable binding as shown in Figure 7(b). In this binding, variables *b* and *a*, which are inputs to operations 1 and 4 (which are bound to the same functional unit), have been bound to the same register. Similarly, variables *f* and *d* are bound to the same register.

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 CDFG. Compatibility edges are added between nodes corresponding to variables which do not have overlapping lifetimes or are created under a different set of conditions.

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 represents a valid variable to register binding with minimal interconnect . This is solved by formulating it as a min-cost max-flow network problem. This formulation has been used earlier to solve similar problems in [15] and [22].

7. RESULTS OF RESOURCE BINDING

We synthesized the various designs with a naive resource binding ("Unbound" case) and with the interconnect minimization methodology outlined above ("Bound" case). The results for the MPEG *pred*2 and the ADPCM Encoder designs are summarized in the graphs in Figure 6. The metrics compared in these graphs are the critical path length, total delay and the unit area. The values for each metric are normalized to the minimum value for that metric among all its values (before and after binding).

The reductions in area of the "Bound" case over the "Unbound" case are significant, especially for the *pred*2 function. 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

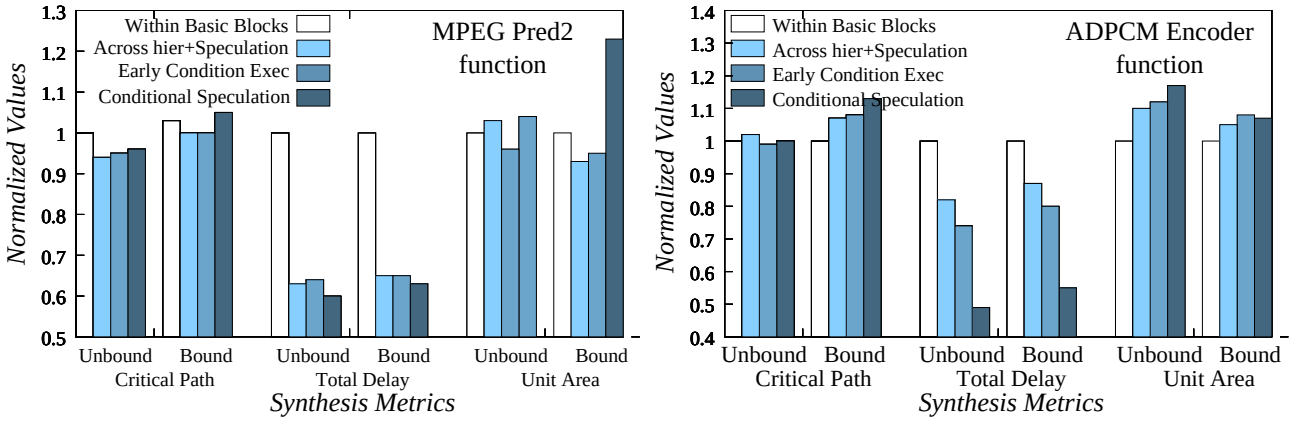


Figure 6: Synthesis results before and after resource binding for MPEG *pred2* and ADPCM Encoder

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 bound and unbound cases, we find that applying the conditional speculation transformation can lead to an increase in the critical path lengths and area of the synthesized circuit. This is due to the higher resource sharing and code duplication that this code motion leads to. Hence, there is a need to develop more accurate control cost models, which the scheduling heuristic can use while deciding which code motions to apply.

8. CONCLUSIONS

In this paper, we have presented a new code motion for synthesis, called conditional speculation. This code motion is particularly effective for control-intensive behaviors where more opportunities exist to move operations into otherwise idle slots in conditional branches. We have shown that these kind of aggressive code motions can be directed to obtain significant reductions in the execution cycles of a design and also the number of states in the controller for large control-intensive segments of industrial strength benchmarks. Furthermore, the control and interconnect overheads incurred due to these code motions can be minimized by resource binding targeted at interconnect minimization. This leads to lower area, without adversely effecting the latency of the final hardware generated by logic synthesis tools. Future work will involve developing more elaborate cost models that will help in guiding when to apply the various code motions.

9. REFERENCES

[1] 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.

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

[3] 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 Conf.*, 2001.

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

[5] R. Walker and D. Thomas. Behavioral transformation for algorithmic level ic design. *IEEE Trans. on CAD*, October 1989.

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

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

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

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

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

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

[12] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 7, July 1981.

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

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

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

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

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

[18] S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. In *Intl. Symp. on Microarchitecture*, 1992.

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

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

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

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