ICS

TECHNICAL REPORT

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

Sumit Gupta Nick Savoiu
Nikil Dutt Rajesh Gupta Alex Nicolau

Information and Computer Science

University of California, Irvine

Technical Report #01-25
June 2001

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

Sumit Gupta Nick Savoiu
Nikil Dutt Rajesh Gupta Alex Nicolau

ICS
Technical Report

Technical Report #01-25
June 2001

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

Information and Computer Science
University of California, Irvine

Contents

List of Figures

Abstract

We introduce a code transformation technique "conditional speculation" that speculates operations by moving 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 this particular code motion has positive effect on latency and controller complexity, e.g., up to 25 % 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 high-level synthesis design targets. Using an important part of the MPEG-1 example, we show total reductions in schedule lengths of 47 % while at the same time keeping the control 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 which has concentrated on reducing schedule lengths of a design by improved scheduling techniques. The presence of complex control flow significantly affects the quality of synthesis results. Scheduling algorithms 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. Conversely, in reverse speculation, which we introduced in previous work [3], operations before conditionals are moved into *subsequent* conditional blocks and hence, executed conditionally. We found that although these code motions are useful, there needs to be a judicious balance between when to speculate and when to reverse speculate. In this report, we do speculation but 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, they also lead to area and clock period overheads. 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 large interconnect and associated control logic. Interconnect refers to the multiplexors and buses that connect components together.

To address the complexity of the interconnect, this report presents a 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

An abridged version of this report has been accepted for publication at the International Symposium on System Synthesis, October 2001

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 rest of this report is organized as follows. We first discuss related work and then present the conditional speculation code motion. Section 4 evaluates this code motion along with other code motions and studies their effects on the quality of synthesis results. Finally, an interconnect minimization strategy is outlined along with a study of its effectiveness on synthesis results.

2 Related Work

Several works have presented speculative code motions and demonstrated their effect on schedule lengths. CVLS [4] uses condition vectors to improve resource sharing among mutually exclusive operations. Radivojevic et al [5] present a symbolic formulation which generates an ensemble schedule of valid and scheduled traces. The Wavescheduling approach [6] incorporates speculative execution into high level synthesis to achieve its objective of minimizing the expected number of cycles. Santos et al [7] and Rim et al [8] support generalized code motions during scheduling in synthesis systems where operations can be moved globally irrespective of their position in the input description.

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. Industry experience shows that, often, critical paths in control-intensive designs pass through the control unit. To this end, Rim et al [8] use an analytical model to estimate the cost of additional interconnect and control caused by code duplication during code motions. Bergamaschi [9] 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 [1, 2]. Tseng et al [10] use clique partitioning heuristics to find a clique cover for a module allocation graph. Paulin et al [11] 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 [12] use a network flow formulation for minimum module allocation while minimizing interconnect. Gebotys et al [13] presents an integer-programming model for simultaneous scheduling and allocation which minimizes interconnect. Mujumdar et al [14] considers operations and registers in each time-step one at a time and uses a network flow formulation to bind them.

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.

3 Conditional Speculation

The use of speculation in high-level synthesis has been studied before. In speculation, an operation is moved out of a conditional. However, the effect of the operation execution is not committed

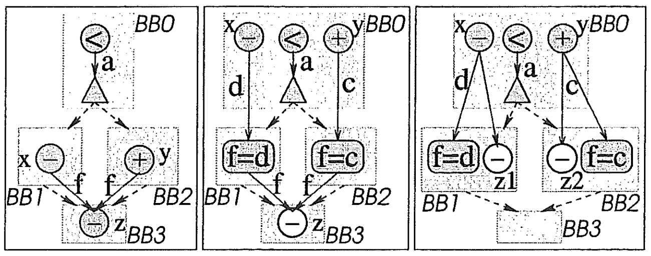

Figure 1. (a) A sample control-data flow graph (b) Operations x and y are speculated leaving idle slots in the conditional branches (c) Operation z is duplicated up into the conditionals BB_1 and BB_2. This conditional speculation leads to higher resource utilization

	MPEG Prediction Block $3ALU, 1*, 2[\,], 3 <<, 2 ==$					
Type of	calc_forw (36 BBs)		pred0_1 (30 BBs)		pred2 (52 BBs)	
Code Motion	# States	Long Path	# States	Long Path	# States	Long Path
Within basic blocks	35	35	44	2588	48	5391
+across hier blocks	25(-29%)	25(-29%)	41(-7%)	2396(-7%)	44(-8%)	5006(-7%)
+speculation	24(-4%)	24(-4%)	28(-32%)	1564(-35%)	31(-30%)	3278(-35%)
+early cond exec	23(-4%)	23(-4%)	27(-4%)	1563(-0%)	31(0%)	3278(0%)
+cond speculation	**21(-9%)**	**21(-9%)**	**23(-15%)**	**1307(-16%)**	**29(-7%)**	**2836(-14%)**
Total Reduction	40.0 %	40.0 %	47.7 %	49.5 %	39.5 %	47.4 %

Table 1. Comparison of various types of code motion for the MPEG Pred benchmark

until the corresponding condition is evaluated. Consider the control-data flow graph (CDFG) in Figure 1(a). The operations x and y can be speculated out of the conditional branches BB_1 and BB_2 respectively as shown in Figure 1(b). This figure demonstrates that the results of the speculated operations are *written back* to the variable f only within the conditional blocks.

Furthermore, there are often operations after the conditional blocks which depend on these variables being written back to, in the conditionals. In Figure 1(b), operation z is dependent on the variable f. Also, the conditional blocks have several "idle" slots in which no operations have been scheduled on the resources. Hence, operations such as z, which lie in basic blocks after the conditional blocks, can be *duplicated up* into both branches of the conditional and executed speculatively. We call this code motion, *conditional speculation.* This is the same as the duplication-up code motion used in compilers.

Figure 1(c) demonstrates conditional speculation of operation z into the conditional branches BB_1 and BB_2 as operations z_1 and z_2. These operations directly use the value that was being written into the variable f in their conditional branch by dynamic renaming [15].

Conditional speculation is not limited to operations which depend on variable write-backs. Any operation after a conditional block can be speculated in this manner provided its dependencies are satisfied.

We have implemented conditional speculation along with other code motions in the *Spark* high-level synthesis system. In the next section, we compare the effectiveness of these code motions by first analyzing their scheduling results and then studying the area and clock periods obtained after logic synthesis.

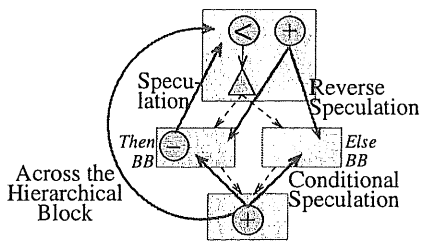

Figure 2. Various types of code motions

4 Effects of Code Motions on Quality of Synthesis Results

The various code motions presented earlier [3] along with conditional speculation are demonstrated 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.

4.1 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 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 benchmarks used are the *Encoder* block from the ADPCM benchmark and the *Prediction* block (a control intensive block) from the MPEG-1 algorithm [16]. The resources used are indicated in the tables; *ALU* does add and subtract, $==$ is a comparator, $[]$ is an array address decoder and $<<$ is a shifter. All of them have single cycle execution time.

The rows in the tables present results with each code motion enabled incrementally. We first enable code motions only within basic blocks (first row) and then across hierarchical blocks, i.e., across entire if-then-else conditionals and loops (second row). The third row allows speculation too, the fourth row has early condition execution enabled as well and the final row has the conditional

ADPCM Encoder(37 Basic Blocks)		
Type of	$1ALU, 2 ==, 2[\,], 1 <<$	
Code Motion	# States	Long Path
Within basic blocks	32	313
+across hier blocks	27(-16%)	262(-16%)
+speculation	23(-15%)	222(-15%)
+early cond exec	20(-13%)	192(-14%)
+cond speculation	**15(-25%)**	**142(-26%)**
Total Reduction	53.1 %	54.6 %

Table 2. Comparison of various types of code motion for the ADPCM Encode benchmark

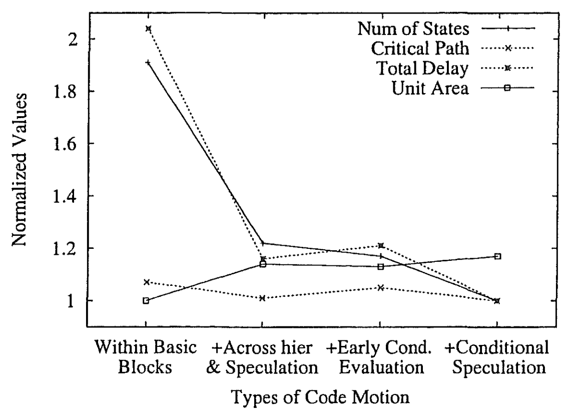

Figure 3. Effects of code motions on various metrics for the MPEG *pred_case2* function

speculation code motion also enabled[1].

The fifth row in these tables demonstrate that enabling conditional speculation alone leads to reductions of 9 to 25 % both in the number of states and the longest path cycles. This code motion is most effective for the ADPCM benchmark since this benchmark is highly control intensive with nearly as many conditional checks as operations. 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.

4.2 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. 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 Synopsys's *Design Compiler* logic synthesis tool. The results for the *pred2* function of the MPEG Prediction block are summarized in the graph in Figure 3.

In this graph, 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, which is the LSI-10K library distributed with Synopsys tools. 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 only within basic blocks, then with across hierarchical block code motions and speculation also enabled, with early condition execution as well and finally with conditional speculation enabled too.

This graph demonstrates that although the longest path cycles and the total delay are halved, the area increases by almost 20 % with all the code motions enabled. Also, although this graph shows that the critical path length remains fairly constant, the critical paths are, in fact, getting

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

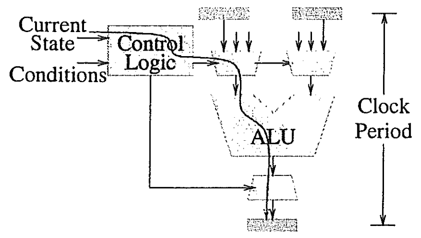

Figure 4. Typical critical paths in control-intensive designs

more complex.

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 multiplexor connected to the functional units. The path continues through the multiplexor itself, through the functional unit and then through a demultiplexor, which finally writes to the output register. The size of these interconnects (multiplexors and demultiplexors) gets increasingly large with the improved resource utilization and sharing caused by aggressive code motions.

5 Reducing Interconnect

Increased resource sharing, however, provides an opportunity to minimize interconnect. Since the resources have several operations and variables mapped to them, there exist opportunities to reduce the the number of inputs to 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 few sections describe an operation and variable binding methodology to minimize these interconnect and control costs.

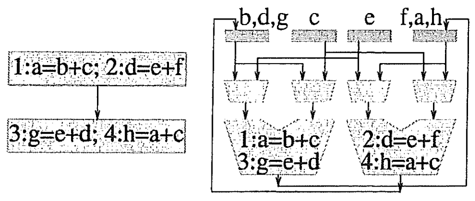

Figure 5. An example of binding leading to a large number of interconnections

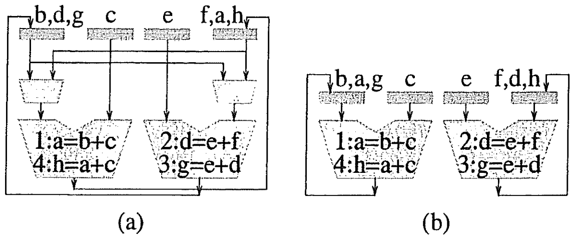

Figure 6. Reducing interconnections by improved (a) operation binding (b) variable binding

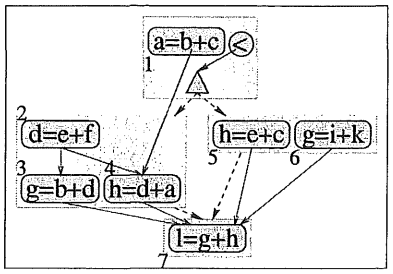

Figure 7. Example used to demonstrate operation and variable binding

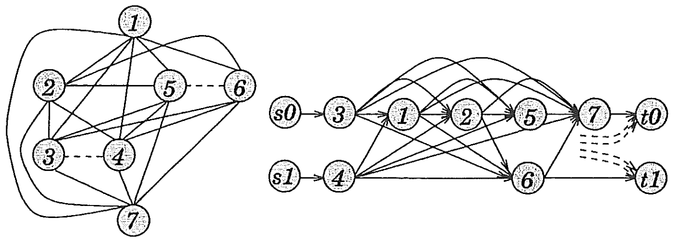

Figure 8. Operation Binding Graph (a) Compatibility Graph (b) Multi-commodity network

5.1 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 6(a).

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. This means that mutually exclusive operations (and their variables) scheduled in the same time step are compatible with each other. The operation binding graph for for the "adder" resource in the example description shown in Figure 7 is presented in Figure 8(a).

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.

A source node and a sink node is created for each of the resources of the current type. A control step in the schedule which uses all the resources of the type under consideration is picked and an edge each is added from a source node to an operation in this control step, such that there is only one edge going out from each source node to any one of the operations in the control step. The

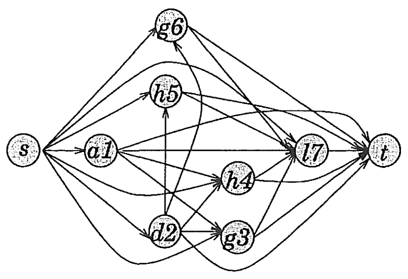

Figure 9. Variable Binding Graph

number of operations in this control step will be less than or equal to the number of resources, since the scheduling was done under resource constraints. Edges are added from each operation node to the sink nodes. All edges in the graph are made directed in such a manner that there are no cycles in the graph (this is possible since the execution times of the operations form intervals). The resulting graph is shown in Figure 8(b). This graph does not show the edge weights and the edges from all the nodes to the sink nodes (t_0 and t_1) have been omitted for clarity.

Chang et al [17] use the same formulation for module allocation but their objective is to minimize power consumption. A max-cost flow through this multi-commodity network represents a valid maximally weighted clique cover [17, 18]. We determine this flow by negating all the weights in the graph and then finding a min cost flow of value equal to the number of resources/source nodes. Nodes left uncovered are put into a compatible clique which leads to the maximum increase in total weight of the cover. This solution represents a valid operation to functional unit binding which minimizes interconnect. For the example in Figure 8, operations 1, 3, 6 and 7 are mapped to one adder and operations 4, 2 and 5 to the other.

5.2 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 6(a) can be further improved by changing the variable binding as shown in Figure 6(b).

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 each instance of a write to a variable in the CDFG. If a variable is written twice, each write gets a new node in the graph. Compatibility edges are added between nodes corresponding to variables which do not have overlapping lifetimes or are created under a different set of conditions. Variables in loops have multiple split lifetime intervals.

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. The resulting compatibility graph for the example in Figure 7 is shown in Figure 9. The weights have been omitted from this figure for clarity.

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. A source and a sink node has been added in the graph in Figure 9 along with edges from(to) the source(sink) node to(from) each node in the graph. A resultant flow of 2 is obtained for this graph

Type of code motion	Critical Path(ns) Unbd	Critical Path(ns) Bound	Unit Area/1000 Unbd	Unit Area/1000 Bound
within BBs	23.0	20.7(-10%)	175	108(-38%)
+hier+spec	21.6	20.1(-7%)	199	138(-31%)
+early cond	22.5	20.0(-11%)	197	137(-30%)
+cond spec	21.4	21.0(-2%)	204	169(-17%)

Table 3. Comparison of critical path length and unit area before and after resource binding

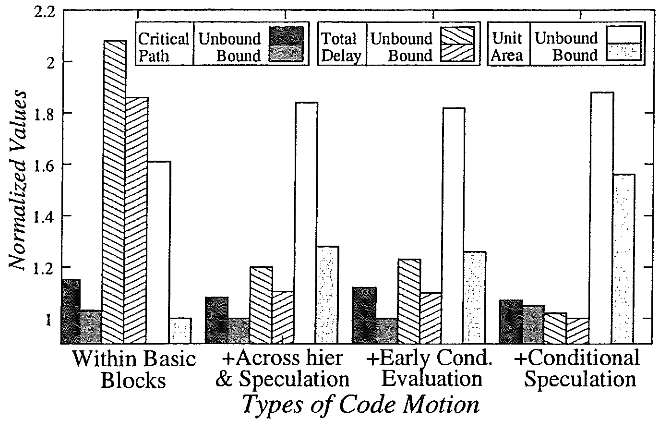

Figure 10. Graphical comparison of critical path length, delay and area before and after resource binding

which binds the variables a_1, g_3, g_6 and l_7 to one register and d_2, h_4 and h_5 to another register. Similar approaches to solve this problem have been used in [12] and [19].

6 Experimental Results and Discussion

We have implemented the various code motions and interconnect minimization resource binding methodology in the *Spark* high-level synthesis system. Spark is being developed in C++ on both the Sun Solaris and Microsoft Windows platforms. It uses *Graphviz* [20] as its graphical user interface and for graph layout and visualization. We have used the LEDA software library [21] for the storing the binding graphs and solving the network flow algorithms.

Table 3 presents the comparison of critical path lengths, and unit area obtained by synthesis of the *pred2* function before and after resource binding for the various code motions. The reductions of the "Bound" column over the "Unbound" column are given in parentheses. Area reductions are significant; between 17 to 38 %. Critical path lengths also decrease albeit to a smaller extent. The unbound area column in the last row in this table shows that the minimum area (169 units) of the design with all the code motions enabled is less than the minimum area (175 units) among all the unbound designs (first row).

The comparisons of these two metrics along with the total delay through the design before and after resource binding are better illustrated in Figure 10 for the different code motions. The values for each metric in this graph are normalized to the minimum value for that metric among all the values (before and after binding).

This graph again demonstrates that 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

Type of Binding	# Regs	Critical Path(ns)	Unit Area
None	130	21.43	204276
Simple	37	21.67(+1.1%)	175857(-13.9%)
Complex	43	21.03(-3.0%)	168960(-3.9%)

Table 4. Comparison of synthesis results for different types of binding

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 almost constant critical path lengths.

Table 4 presents the synthesis results for the *pred2* function when different types of operation and variable binding is done. The first row are the results when no explicit resource binding is done, the second row is with a "simple" binding in which no additional edge weights aimed at reducing interconnect are added between nodes in the binding graphs. The complex binding row presents the results for when these additional weights are added.

This table demonstrates that although simple binding requires the fewest number of registers, it also results in the highest critical path length. Only when binding is done with the interconnect minimization goal, do we obtain a combined lower area and critical path length. Also, despite the higher number of registers needed after complex binding the overall area is lower, signifying that the savings in interconnect and control costs are greater than the extra register area.

7 Conclusions

In this report, we have presented a new code motion for synthesis. This code motion is particularly effective for control-intensive behaviors. We have shown that such 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, clock periods and hence, latency of the final hardware generated by logic synthesis tools.

8 Acknowledgements

The authors would like to thank Prof. Sandra Irani for her help and inputs. This research is supported by the Semiconductor Research Corporation.

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 *Proceedings of the Design Automation Conference*, June 2001. To appear.

[4] K. Wakabayashi and H. Tanak. Global scheduling independent of control dependencies based on condition vectors. In *Proceedings of the Design Automation Conference*, 1996.

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

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

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

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

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

[10] 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.

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

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

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

[14] A. Mujumdar, R. Jain, and K. Saluja. Incorporating performance and testability constraints during binding in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October 1996.

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

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

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

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

[19] J.-M. Chang and M. Pedram. Register allocation and binding low power. In *Proceedings of the Design Automation Conference*, 1995.

[20] AT&T Research Labs. Graphviz - open source graph drawing software. http://www.research.att.com/sw/tools/graphviz/.

[21] Algorithmic Solutions Software GmbH. Leda product page. http://www.algorithmic-solutions.com/as_html/products/leda/products_leda.html.