Speculation Techniques for High Level Synthesis of Control Intensive Designs*

Sumit Gupta Nick Savoiu Sunwoo Kim
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,sunwk,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. In this paper, we explore the effectiveness of various types of code motions, such as moving operations across conditionals, out of conditionals (speculation) and into conditionals (reverse speculation), and how they can be effectively directed by heuristics so as to lead to improved synthesis results in terms of fewer execution cycles and fewer number of states in the finite state machine controller. We also study the effects of the code motions on the area and latency of the final synthesized netlist. Based on speculative code motions, we present a novel way to perform early condition execution that leads to significant improvements in highly control-intensive designs. Overall, reductions of up to 38 % in execution cycles are obtained with all the code motions enabled.

1. INTRODUCTION

High-level synthesis of digital systems from a behavioral description has received significant attention in the last 15 years [1]. However, commercial synthesis tools have gained limited acceptance among designers, primarily due to poor synthesis results in the presence of conditionals and especially loops, and lack of controllability of quality of results.

For effectiveness, a high-level synthesis (HLS) system has to make the right tradeoffs among available time, performance, and area costs. Furthermore, the presence of complex control flow significantly effects the quality of synthesis results. In this paper, we propose techniques to move operations across control structures (conditionals and loops) that enable HLS algorithms to make these tradeoffs effectively. Scheduling algorithms can use these beyond-basic-block code motion techniques like speculative execution to extract the inherent parallelism in the design and increase resource utilization.

Speculation is not a new concept; indeed, code motions and speculation are supported either partially or fully in several previously presented synthesis systems [2, 3, 4, 5]. CVLS [2] uses condition vectors to improve resource sharing among mutually exclusive operations. Radivojevic et al [3] present a symbolic formulation which generates an ensemble schedule of valid and scheduled traces. The Wavescheduling approach [5] incorporates speculative execution into high level synthesis to achieve its objective of minimizing the expected number of cycles. Santos et al [4] support generalized code motions in a synthesis system where operations can be moved and scheduled irrespective of their position in the input description. However, several of these approaches either do not handle loops or place restrictions on the nesting of loops within conditionals or vice versa. Furthermore, these approaches do not maintain information about hierarchical structuring of the code, which leads to expensive and inefficient code motion techniques (see Section 2).

Most previous works present and compare results in terms of the schedule lengths. This has prevented a clear analysis of the effects of scheduling and code motions on the area and latency of the hardware generated. Because of this, the control logic overheads are usually ignored despite the fact that industry experience has often shown that critical paths in a high performance design lie in the control unit.

We are developing a modular and extensible high-level synthesis research system called *Spark*. We have used parallelizing compiler technology developed previously in our group [6, 7] and instrumented and modified 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. Spark provides an integrated flow from architectural design to logic synthesis.

This paper presents some basic code motion transformations and a simple heuristic that judiciously chooses which code motion should be applied, while making performance, resource and area cost trade-offs. These judicious choices of the various code motions to control the quality of synthesis results is the chief contribution of this paper. The results section shows the effects of the code motions on the quality of synthesis results and demonstrates which code motions are most effective for synthesis.

*This work is supported by the Semiconductor Research Corporation

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

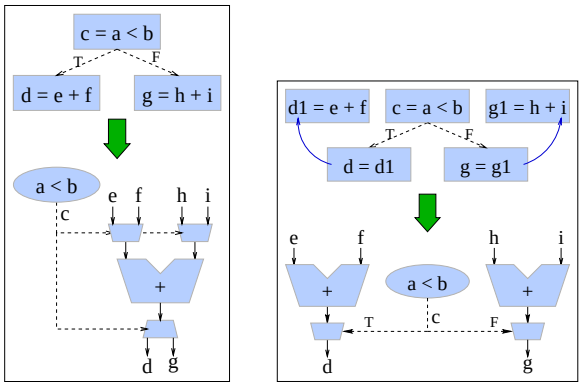

Figure 1: Effect of speculative execution on longest path and resource requirements

2. A MODEL FOR CONTROL INTENSIVE DESIGNS

The *Spark* system stores the behavioral description in an intermediate representation (IR) which retains all the information given in the input description. This is critical for enabling coarse-level transformations and making global decisions about code motion. The IR consists of basic blocks of operations encapsulated in *Hierarchical Task Graphs* (HTGs) [6]. Basic blocks making up an if-then-else conditional block or a for loop are aggregated into a compound HTG node.

The structural nature of the HTGs retains information about the mutual exclusiveness of operations, and can be used for making global decisions about code motion and speculation. HTGs are key to fast and efficient code motion techniques such as *Trailblazing* [6] and *Resource-Directed Loop Pipelining* [7], both of which are implemented in Spark.

3. SPECULATION IN SYNTHESIS

In the presence of control structures, maximal parallelism can be extracted by exposing concurrency using code motions which move operations beyond control boundaries.

One of the key enabling transformations for efficient code motion is speculation. *Speculative execution* refers to the unconditional execution of instructions that were originally supposed to have executed conditionally. In the compiler context, if the condition evaluates to false, 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.

The example in Figure 1 demonstrates speculation. 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, these two operations are *mutually exclusive*. They can, hence, be scheduled on the same hardware resource as shown in the corresponding circuit in Figure 1(a).

Now, consider that enough resources (an additional adder) are available; then the operations within the conditionals can be calculated *speculatively* and concurrently with the calculation of the conditional c as shown in Figure 1(b). 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 that 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 used to execute operations speculatively. Hence, speculation along with other code motions needs to be directed by a global scheduling heuristic.

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 [1].

For the purpose of evaluating the various code motion transformations, we have chosen a *Priority-based* global list scheduling algorithm, although the transformations presented here can be applied to other scheduling heuristics as well. Since our objective is to minimize the *longest delay* through the design, we assign a priority to each operation which is one more than the maximum of the priorities of all the operations that use its result. Hence, operations that produce outputs have priority zero, and those which depend on them have priority one and so on, as shown in Figure 2(a) for the waka benchmark [2]. The priority assignment can also be changed to minimize a different cost function, such as average delay.

The Priority algorithm determines the operation with the highest priority which is ready to be scheduled (data dependencies are satisfied) and then employs techniques such as speculation and dynamic renaming to move the operation so as to schedule it on the available resource. To further aid in improving the resource utilization, we have implemented two more code motion techniques, namely, *reverse speculation* and *early condition execution*. These techniques are discussed in the next two sections.

4.1 Reverse Speculation

The Priority scheduling algorithm can determine if it is more "profitable" to speculatively execute operations which are within conditionals. Conversely, a situation may arise that an operation outside a conditional has lower priority than another operation inside the conditional. The operation outside the conditional can then be *reverse speculated* into the conditional, so that the resources freed can be better utilized by higher priority operations.

This is demonstrated in Figure 2. The operations g and e have higher priority than operation c since they lie on the longest path of the design. Hence, operation c can be reverse speculated into the conditional as shown in Figure 2(b). Also, 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.

4.2 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 execute 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. The motivation for this technique comes from the fact that a conditional

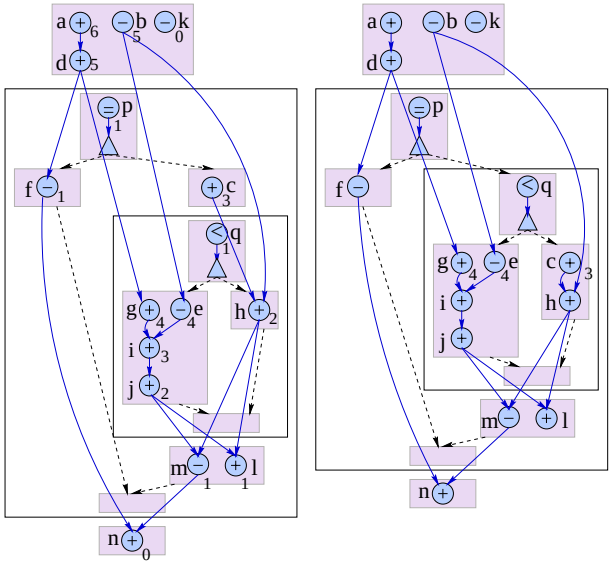

Figure 2: Reverse Speculation for the waka benchmark (a) original design (b) after reverse speculation

has high priority since all the operations in its conditional branches depend on it.

Early condition execution is demonstrated for the *waka* benchmark in Figure 3. In this figure, the operations p and q which calculate the conditions are executed as soon as possible and hence the conditionals based on them can be checked early. Unscheduled operations from basic blocks preceding the conditional (d, k and c) are *reverse speculated* into the conditional branches as shown in Figure 3(b). Note that operations d and c are reverse speculated into only those branches which use their results.

This also leads to more efficient resource sharing since the operations on either side of the conditional are mutually exclusive. By this technique and the use of HTGs, we are able to implicitly extract and use information about mutual exclusivity of operations without using computationally expensive Binary decision diagram (BDD) packages [3, 4].

These transformations are implemented in the Spark system and the Priority list scheduling algorithm determines when to apply the transformations based on the priorities of the operations. That is, at each cycle, the highest priority "available" operation is scheduled on each resource by employing the necessary code motions. 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.

5. EXPERIMENTS AND RESULTS

This section studies the effects of the various code motions directed by the Priority list scheduling algorithm on the synthesis results. The results are presented in terms of the number of states in the finite state machine in the controller of the generated RTL VHDL and the cycles on the longest path in the design. For loops, the longest path length of the loop body is multiplied by the number of loop iterations. Longest path length is equivalent to the execution cycles of the design. The results of logic synthesis are also presented to evaluate the effects of the transformations on area and latency of the synthesized design.

Table 1 demonstrates the effectiveness of the various types

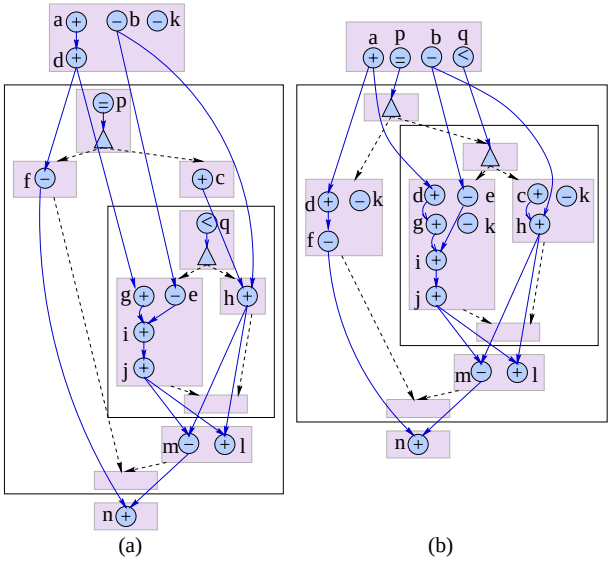

Figure 3: Code Restructuring by Early Condition Execution (a) before (b) and after

of code motion described in this paper. The benchmarks used are the *Encoder* from the ADPCM benchmark and the *Prediction* block from the MPEG-1 algorithm (available for download from Spark's web page). The ADPCM benchmark has 37 Basic Blocks and MPEG Prediction has four functions, *calc_forward_motion*, *calc_backward_motion*, *calcid* and *pred* with 36, 36, 1 and 80 basic blocks respectively (only non-empty basic blocks are counted). The *pred* function has a 3-case switch statement which we partitioned into two functions *pred*0_1 and *pred*2 since logic synthesis tools cannot handle the complete function. Since the first two functions are extremely similar and the *calcid* function has only 1 basic block, we only present results for the functions *calc_forward_motion*, *pred*0_1 and *pred*2. The resources used are indicated in the table; *ALU* does add and subtract, = is a comparator, [] is an array address decoder and << is a shifter. All components used have single cycle execution time. The number of resources has been chosen so as to get shortest schedule length possible.

Table 1 compares the results for code motions only within basic blocks (first row), across Hierarchical Task Graphs (HTGs), i.e., across entire if-then-else structures and loops but without speculation (second row), then with speculation enabled (third row), with reverse speculation enabled as well (fourth row) and the fifth row also performs the early condition execution code transformation[1]. The number of states in the FSM and the longest path length (execution cycles) are presented along with the percentage reductions of each row *over the previous row* in parentheses.

Significant reductions are achieved when code motions across HTGs are enabled and then again when speculation is enabled. The table demonstrates that reverse speculation on its own does not lead to improvements, but when performed as a part of early condition execution, additional reductions of up to 13 % can be achieved. The maximum benefits by this transformation are seen in the ADPCM benchmark since this benchmark is highly control intensive with nearly as many conditional checks as operations. The total reduc-

[1]Although both benchmarks have loops, we have not applied any loop transformations, across loop boundary code motion or loop unrolling for these experiments

Type of Code Motion	ADPCM Encode (37 BBs) $1ALU, 2==, 2[\,], 1<<$		MPEG Prediction Block $3ALU, 1*, 2[\,], 3<<, 2==$ calc_forw (36 BBs)		pred0_1 (30 BBs)		pred2 (52 BBs)	
	# States	Long. Path	# States	Long. Path	# States	Long. Path	# States	Long. Path
Within BBs	32	313	35	35	44	2588	48	5391
+across HTGs	27(-15.6%)	262(-16.3%)	25(-28.6%)	25(-28.6%)	41(-6.8%)	2396(-7.4%)	44(-8.3%)	5006(-7.1%)
+speculation	23(-14.8%)	222(-15.3%)	24(-4%)	24(-4%)	28(-31.7%)	1564(-34.7%)	31(-29.5%)	3278(-34.5%)
+reverse spec	23(0 %)	222(0 %)	24(0%)	24(0%)	28(0%)	1564(0%)	31(0%)	3278(0%)
+early cond	20(-13.0%)	192(-13.5%)	23(-4.2%)	23(-4.2%)	27(-3.6%)	1563(-0%)	31(0%)	3278(0%)
Total Reduction	**37.5 %**	**38.7 %**	**34.3 %**	**34.3 %**	**32.6 %**	**32.3 %**	**37.5 %**	**36.8 %**

Table 1: Comparison of various types of code motion for the ADPCM Encode and MPEG Pred benchmarks

Type of Code Motion	*calculate_forward_motion* func(36 BBs) Crit. P. (c ns)	Long. P.(l)	Delay (c.l nanosec)	Unit Area	*pred_case_0_1* function(30 BBs) Crit. P. (c ns)	Long. P.(l)	Delay (c.l nanosec)	Unit Area
Within BBs	22.30	35	780.5	10752	22.31	2588	57738	276077
+across HTGs+spec	22.81	24	547.4(-29.8%)	14813(+27.4%)	22.46	1564	35127(-39.2%)	286019(+3.5%)
+rev spec+early cond	22.49	23	517.3(-5.5%)	14261(-3.7%)	21.86	1563	34167(-2.7%)	290882(+1.7%)

Table 2: Effect of code motion on the logic synthesis results for the functions of the MPEG Pred benchmark

Bench mark	# BBs	Resources	Jess Sch Len	Brewer Sch Len	Brewer Run Time	Spark Sch Len	Spark Run Time
waka	9	1+,1-,2=	7	7	-	7	0.1
rotor	11	2+-,2*,1[ ]	7	7	13.7	7	0.16
s2r	21	3+-,2*,1[ ]	8	9	177.9	9	0.38

Table 3: Comparison with previous work

tion in execution cycles achieved with all the transformations enabled over code motion within basic blocks is up to 38 %.

We have synthesized the RTL VHDL generated by Spark using Synopsys's *Design Compiler* and present the results for two functions from the MPEG Pred benchmark in Table 2. The columns in this table present the results the critical path length (c nanoseconds), the execution cycles of the longest path (l), the total delay through the design (c.l nanoseconds) and the unit area (total of the combinational and non-combinational areas). The unit area is based on the technology library being used; we have used the LSI-10K library that is distributed with Synopsys tools. The percentage reductions in the delay and area over those in the previous row are given in parentheses.

The first row in this table gives results for only within basic block code motions, the second row with across HTGs and speculation also enabled and the third row with all code motions enabled. This table shows that total delay of the design can be reduced significantly by speculation and early condition execution. Speculation comes with a higher area cost since we have not done any binding and rely on the logic synthesis tool for this. This leads to higher storage, interconnect and control costs. However, the critical path length of the design remains almost constant and hence, the clock cycle length does not increase due to these techniques.

Table 3 compares the results of our system, *Spark* with Brewer [3] and Jess [4] for the benchmarks *waka* [2], *rotor* and *s2r* [3] for the indicated resources. The columns present the number of basic blocks, the longest path length/cycles and the CPU run time in seconds of Brewer's system on a SUN Sparc Station 10 (probably running at 33 or 66 Mhz) and of Spark on a 170 Mhz SUN Sparc Station 5. The results show that the Spark system produces similar results when compared to the other systems for these benchmarks. Although we do not have the run times for the system "Jess", this system uses combinatorial approaches such as genetic algorithms or simulated annealing for search space exploration which usually have high run times. This table demonstrates that by making judicious choices of code motions, using even a simple priority list scheduling heuristic, the Spark system is able to produce good results without resorting to more complex scheduling heuristics.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a comparative study of the effects of various kinds of code motions on the quality of results of high-level synthesis and studied which are most effective. The results demonstrate that code motions across entire conditional blocks and speculative code motions lead to most improvements. Early condition execution can provide an additional reduction up to 13 % in execution cycles for highly control-intensive designs. We have tried to compare our approach to similar works using commonly used benchmarks. However, this is not always possible mainly because of our focus on control intensive and moderately complex designs. We are currently implementing resource binding to reduce the area increase due to code motions.

7. REFERENCES

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

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

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

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

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

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

[7] 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.