UC Irvine

ICS Technical Reports

Title

Speculation techniques for high level synthesis of control intensive designs

Permalink

<https://escholarship.org/uc/item/5036p47p>

Authors

Gupta, Sumit
SavoIU, Nick
Kim, Sunwoo
et al.

Publication Date

2000

Peer reviewed

ICS

TECHNICAL REPORT

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Speculation Techniques for High Level Synthesis of Control Intensive Designs*

Sumit Gupta Nick Savoie Sunwoo Kim
Nikil Dutt Rajesh Gupta Alex Nicolau

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine.
<http://www.cecs.uci.edu/~spark>
{sumitg,savoie,sunwk,dutt,rgupta,nicolau}@cecs.uci.edu

Information and Computer Science
University of California, Irvine

*Technical Report #00-40
December 2000*

Speculation Techniques for High Level Synthesis of Control Intensive Designs*

Sumit Gupta Nick Savoiu Sunwoo Kim
Nikil Dutt Rajesh Gupta Alex Nicolau

ICS Technical Report

*Technical Report #00-40
December 2000*

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

Information and Computer Science
University of California, Irvine

*Submitted for publication

Contents

1	Introduction	2
2	A Model for Control Intensive Designs	3
3	Speculation in High Level Synthesis	4
4	Priority-based Global List Scheduling Heuristic	5
4.1	Reverse Speculation	6
4.2	Code Restructuring by Early Condition Execution	7
5	Experiments and Results	8
6	Conclusions and Future Directions	10

List of Figures

1	The Hierarchical Task Graph Intermediate Representation of a For Loop	3
2	Hierarchical Task Graph representation of the Benchmark “waka” [1]. The numbers indicate the priorities of the operations	4
3	(a) An example design along with the control-data flow graph (b) speculative execution of operations from within the branches	5
4	Reverse Speculation for the waka benchmark (a) original design (b) after reverse speculation	6
5	Code Restructuring by Early Condition Execution (a) original design (b) design after early condition execution	7

Abstract

The automated synthesis of a design from its behavioral description, known as high level synthesis, has been the subject of much research, both in the industry and academia. However, the quality of results of current high-level synthesis systems is adversely affected by the presence of conditionals and loops. Furthermore, the effect of scheduling decisions and resource sharing on the control and steering logic overhead is not well-understood. In this paper, we show how various types of code motions, such as moving operations across conditionals, out of conditionals (speculation) and into conditionals (reverse speculation), 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 describe and evaluate a new code motion technique known as *early condition execution* which executes the conditional check as soon as possible. Reductions of up to 50 % in execution cycles are obtained.

1 Introduction

High-level synthesis of digital systems from a behavioral description has received significant attention in the last 15 years [2, 3]. However, commercial synthesis tools have gained limited acceptance among designers, primarily due to poor synthesis results in the presence of conditional branches and especially loops, and lack of controllability of quality of results.

The quality of results for control intensive designs is significantly affected by the control flow in typical applications. The control flow in a design is also affected by the programming style. Some work has focussed on reducing the sensitivity of synthesis to programming style [4, 5]. For effective high-level synthesis, the system has to make the right tradeoffs among available concurrency (resource requirements), available time (latency), performance (cycle time, pipelining) and area costs (control overhead due to registers, multiplexing and steering logic). 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 along with speculative execution to extract the inherent parallelism in the design and increase resource utilization.

Speculative execution refers to the unconditional execution of instructions that were originally supposed to have executed conditionally. In the software compiler context, if these conditions evaluate 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 (see Section 3).

Code motions and speculation are supported either partially or fully in some previously presented synthesis systems [1, 6, 7, 8, 9]. Whereas CVLS [1] does not support all types of speculation, [6] does scheduling for small groups of basic blocks at a time. Brewer et al [7] use an exact symbolic formulation where the entire scheduling problem is modeled as Boolean functions. This, along with an expensive trace enumeration and validation step make this approach computationally very expensive. The work presented by Santos in [10] supports speculation completely but uses computationally expensive algorithms such as genetic algorithms and simulated annealing for searching the design space. Additionally, several of these approaches employ Binary Decision Diagram (BDD) packages for extracting mutually exclusive information about operators, further adding to complexity of the problem. Several of these approaches either do not handle loops or place restrictions on the nesting of loops within conditionals or vice versa.

Figure 1: The Hierarchical Task Graph Intermediate Representation of a For Loop

Also, most of the previous published works compare the performance of their algorithms by calculating only schedule lengths, and not the area and latency of the hardware generated. This has prevented a clear analysis of the effects of scheduling and code motions on area of the final design, since control logic overheads are usually ignored. Industry experience often shows 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 [11] and instrumented and modified it for high-level synthesis. The Spark system tries to maximally extract parallelism, enhance resource sharing and schedule the operations based on resource constraints, by employing heuristic-directed speculation and code motions. 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 level and fine-grain level optimizations on synthesis results. The input language for Spark is ANSI-C, currently with the restrictions of no pointers and no function recursion. The system provides an integrated flow from architectural design to logic synthesis.

The idea of code motions and speculation has been studied extensively in the software compiler community. However, the effects of code motions on synthesis results are not well understood. 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. The results section shows the effects of the code motions on the quality of synthesis results.

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

Figure 2: Hierarchical Task Graph representation of the Benchmark “waka” [1]. The numbers indicate the priorities of the operations

transformations and making global decisions about code motion. The IR consists of basic blocks encapsulated in *Hierarchical Task Graphs* (HTGs) [12, 11]. In addition to operation level and basic block level information, HTGs also maintain coarse, high level information about compound nodes. That is, the basic blocks comprising an if-then-else conditional block or a for loop are aggregated into a compound HTG node. For example, as shown in Figure 1, a *For loop HTG* consists of the member basic blocks; the initialization basic block, the conditional check block, the body of the loop and the loop index increment. Similarly, in Figure 2 the benchmark *waka* introduced by Wakabayashi et al [1] is shown as represented by HTGs. In this figure, the dashed arrows indicate control flow and the solid lines indicate data flow. Operations are denoted by circular nodes with the operator sign within and the triangle indicates a conditional check. The number next to each operation node indicates the priority of the operation (see Section 4).

The structural information maintained by HTGs can be used for making global decisions about code motion and speculation and is used by heuristics to guide code motion techniques such as *Trailblazing* [11] and *Resource-Directed Loop Pipelining* [13], both of which are implemented in the Spark system. For brevity, these are not explained further in this paper.

3 Speculation in High Level Synthesis

In the presence of control structures, maximal parallelism can be extracted with the use of code motion, i.e., code restructuring by the movement of operations within and beyond basic blocks, and across and out of control structures such as conditionals and loops. Code motion exposes concurrency, hence, increasing opportunities for resource utilization without an increase in latency. Also, code motions restructure the control flow in the design and lead to reduced control costs. These reduced control costs can be quantified in terms of the number of states in the finite state machine controller.

Figure 3: (a) An example design along with the control-data flow graph (b) speculative execution of operations from within the branches

One of the key enabling transformations for efficient code motion is *speculation*. As explained earlier (see Section 1), speculation is the execution of an operation before the condition under which it is supposed to execute has been evaluated. The example in Figure 3 demonstrates how speculation can be used to extract parallelism and reduce execution time. In Figure 3(a), variables d and g are calculated based on the result of the calculation of the conditional c . Since calculation of d and g is done on the 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 hardware circuit in Figure 3(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 3(b). Based on the evaluation of the conditional, one of the results will be discarded and the other committed. As is evident from the corresponding hardware circuits, the longest path gets shortened from being a sequential chain of the comparison followed by the addition to being a parallel computation of the comparison and the additions followed by a multiplexing of the results.

However, this example also demonstrates the additional costs of speculation. Speculation leads to requires more resources and more storage for the intermediate results. So, uncontrolled aggressive speculation can lead to worse results due to extra resources required. On the other hand, resources that are idle 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, one of the main tasks in high level synthesis, is the task of assignment of operations to control steps or time intervals so that the resources allocated to the design can compute the operations assigned to each step [2, 3, 14].

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. The Priority algorithm assigns a priority to each operation to be scheduled which is one more than the maximum of the priorities of all the oper-

Figure 4: Reverse Speculation for the waka benchmark (a) original design (b) after reverse speculation

ations that use its result. Hence, the priority of operations whose result is not used by any other operation (like an output variable) is zero and those on which these operations depend have priority one and so on. The priorities of all the operations in the waka benchmark are denoted next to the nodes in Figure 2.

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 [15] to move the operation so as to schedule it on the available resource. It uses efficient code motion techniques such as Trailblazing [11]. 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

Synthesis results can vary widely due to syntactic variance of the input description. 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 by this reverse speculation can be better utilized by higher priority operations.

This is demonstrated in Figure 4. The operations g and e lie on the longest path of the design and the operation c does not. Hence, g and e have higher priority than operation c . Operation c is reverse speculated into the conditional as shown in Figure 4(b). Also, the reverse speculation algorithm detects that operation c is required only in one of the branches of the conditionals and hence, moves it only into that conditional. This is determined by checking which basic block(s) have operations that use the result of the operation c and moving c only to those basic block(s).

Figure 5: Code Restructuring by Early Condition Execution (a) original design (b) design after early condition execution

Also, as shown in the figure, this leads to more efficient resource sharing since the operations on either side of the conditional are mutually exclusive. By this technique, we are able to implicitly extract and use information about mutual exclusivity of operations without using computationally expensive binary decision diagram (BDD) packages [7, 8].

4.2 Code Restructuring by Early Condition Execution

Reverse speculation can be coupled with another novel transformation that we introduce, 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 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 5. 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. All the operations which have not been executed so far from the basic blocks preceding the conditional (d , k and c) are *reverse speculated* into the conditional branches as shown in Figure 5(b). Furthermore, since the result of operation d is required in only two out of the three conditional basic blocks, it is reverse speculated into only those two and operation c is reverse speculated into only one of the conditional branches.

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. Hence, at each cycle, the highest priority available operation is scheduled on the resource by employing the necessary code motions. Any operations that are 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. In this way, syntactic invariance of the input description can also be

Type of Code Motion	ADPCM Encode (37 BBs)		MPEG Pred 4ALU, 1*, 2[], 2 <<			
	1ALU, 2 ==, 2[], 1 <<		calc_forw (36 BBs)		pred (180 BBs)	
	Num of States	Longest Path Cycles	Num of States	Longest Path Cycles	Num of States	Longest Path Cycles
Only within BBs	32	1204	35	35	55	5525
+ across HTGs(no spec)	27(-15.6%)	1004(-16.6%)	26(-25.7%)	26(-25.7%)	50(-9.1%)	4885(-11.6%)
+ with speculation	21(-22.2%)	764(-23.9%)	25(- 3.8%)	25(-3.8%)	44(-12 %)	4307(-11.8%)
+ with reverse spec	21(0 %)	764 (0 %)	25(0 %)	25(0 %)	44(0 %)	4307(0 %)
+ with early cond exec	19(-9.5%)	684 (-10.5%)	24(-4 %)	24(- 4 %)	42(-4.5 %)	4305(0 %)

Table 1: Comparison of various types of code motion for the ADPCM Encode and MPEG Pred benchmarks; % reduction is given in parentheses

eliminated since operations are moved in the design independent of their placement in the original input description.

5 Experiments and Results

In this section, we quantify the effects of the code motion transformations using 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. If a loop is encountered while calculating the number of cycles on the longest path, 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. We go a step further and also present logic synthesis results to evaluate the effects of the transformations on area and speed of the synthesized design. Spark is being developed in C++ on both the Sun Solaris and Microsoft Windows platforms. It uses *Graphviz* [16] as its graphical user interface and for graph layout and visualization.

Table 1 demonstrates the effectiveness of the various types 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 at our project ftp site [17]). The ADPCM benchmark has 37 Basic Blocks and the MPEG Pred has four functions, *calculate_forward_motion*, *calculate_backward_motion*, *calcid* and *pred* with 36, 36, 1 and 180 basic blocks respectively. Since the first two functions are extremely similar and the *calcid* function has only 1 basic block, we only present results for the functions *calculate_forward_motion* and *pred*. ADPCM Encode has been scheduled with 1 ALU (does add and subtract), 2 comparators 2 array address decoders and 1 shifter. MPEG Pred has been scheduled with 4 ALUs (does add, subtract and comparisons), 1 single-cycle multiplier, 2 array address decoders and 2 shifters.

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 also enabled (fourth row) and the fifth and final row also performs the early condition execution code transformation¹. The comparisons are based on the number of states in the finite state machine of

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

Type of Code Motion	MPEG Pred 4ALU, 1*, 2[], 2 << calculate forward motion function(36 BBs)				
	States	Crit Path (c ns)	Longest Path(l)	Delay (c.l ns)	Unit Area
Only within BBs	35	20.03	35	701.05	11602
+ with spec	25	20.66	25	516.50(-26.3%)	13401 (+15.5 %)
+ rev spec & early cond exec	24	20.73	24	497.52(-3.7%)	12812 (-4.4 %)
Type of Code Motion	pred function (180 BBs)				
	States	Crit Path (c ns)	Longest Path(l)	Delay (c.l ns)	Unit Area
Only within BBs	55	20.94	5525	115693.50	239858
+ with spec	44	20.96	4307	90274.72 (-22.0 %)	240468 (+0.3 %)
+ rev spec & early cond exec	42	20.81	4305	89587.05(-0.1 %)	239302(-0.5%)

Table 2: Effect of code motion on the final logic synthesis results for the MPEG Pred benchmarks

the controller and the longest path length (execution cycles). The percentage reductions of each row over the previous row are given in parentheses.

The results demonstrate that reductions of up to 10-25 % in execution cycles can be obtained by enabling code motions across HTGs alone. An additional 4-24 % are enabled by speculation of operations. Also, as seen in the last two rows in the table that although performing reverse speculation alone does not lead to improved results, when done as part of early condition execution, an additional reduction of up to 10 % can be obtained. The total reduction in terms of execution cycles possible with all the transformations over no code motion is up to 50 %.

Table 2 shows the effect these transformations have on the final logic synthesis results for the two functions from the MPEG Pred benchmark. The columns in this table present the results for the number of states in the finite state machine (St), 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 synthesis results were obtained by synthesizing the register-transfer level VHDL code produced by Spark using the Synopsys logic synthesis tool, *Design Compiler* [18].

The results in this table are for scheduling with only the Priority algorithm with no beyond basic block code motion (first row), with speculation (second row) and with early condition execution (third row). Once again the percentage reductions of the delay and area over those in the previous row are given in parentheses. 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 (due to additional storage required). However, the critical path length of the design either reduces or is constant which means that the clock cycle does not increase due to these techniques.

Table 3 compares the results of our system, *Spark* with Brewer et al's OBDD based system [7] and system described in [10, 8]. The columns present the longest path length/cycles and the CPU run time of Brewer and Spark on a Sun Sparc Station 10 and 5 respectively. The results are

Bench- mark	# Basic Blocks	Resources	Jess	Brewer		Spark	
			Longest Path Cycles	Longest Path Cycles	CPU Run Time	Longest Path Cycles	CPU Run Time
waka	9	1+,1-,2==	7	7	-	7	0.1 sec
rotor	11	2+/-,2*,1[]	7	7	13.7 sec	7	0.16 sec
s2r	21	3+/-,2*,1[]	8	9	177.9 sec	9	0.38 sec

Table 3: Comparison of our system, Spark, with previous work, Jess [8] and Brewer [7]

presented for the benchmarks *waka* [1], *rotor* and *s2r* [7] for the indicated resources (== means comparator, [] means array address decoder). All components used have single cycle execution time. The results in this table demonstrate the Spark system produces equally good results when compared to the other systems for these benchmarks. However, the CPU times demonstrate that the other approaches are more computationally expensive. Even for a small benchmark like *s2r* with 21 basic blocks, Brewer’s approach requires 177.9 seconds on a SUN Sparc Station 10 (probably running at 33 or 66 Mhz) whereas our system requires just 0.38 seconds on a 170 Mhz SUN Sparc Station 5. 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. We also measured the run time of the MPEG Pred benchmark which has a total of over 250 basic blocks and found it be only 16.69 seconds.

6 Conclusions and Future Directions

In this paper, we have presented results to show the effects of various kinds of code motion on the quality of results of high-level synthesis. Although code motions and particularly speculation have been popular concepts in software compilers, their effect on synthesis results has not been studied carefully before. We have also shown how the synthesis results can overcome syntactic variance in the input description by techniques such as early condition execution and reverse speculation.

However, currently the early condition execution transformation does not move conditional checks across other HTG nodes. This transformation can be enhanced to execute the first conditional check that is ready even if that involves interchanging the conditional check order. Also, during the course of our experiments, we have seen that the critical path in high performance processor-block type of designs typically lies in the control logic. Hence, we are currently exploring control optimizations, to minimize the area and performance overheads in the control unit.

References

- [1] K. Wakabayashi and H. Tanak. Global scheduling independent of control dependencies based on condition vectors. In *Proceedings of the Design Automation Conference*, 1996.
- [2] 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.
- [3] R. Camposano and W. Wolf. *High Level VLSI Synthesis*. Kluwer Academic, 1991.
- [4] V. Chaiyakul, D.D. Gajski, and L. Ramachandran. Minimizing syntactic variance with assignment decision diagrams. Technical Report ICS-TR-92-34, UC Irvine, 1992.
- [5] M.Janssen, F.Catthoor, and H.De Man. A specification invariant technique for operation cost minimisation in flow-graphs. In *International Symposium on High-level Synthesis*, 1994.
- [6] M. Rim, Y. Fann, and R. Jain. Global scheduling with code-motions for high-level synthesis applications. *IEEE Transactions on VLSI Systems*, September 1995.
- [7] I. Radivojevic and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE Transactions on CAD*, January 1996.
- [8] L.C.V. dos Santos and J.A.G. Jess. A reordering technique for efficient code motion. In *Design Automation Conference*, 1999.
- [9] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. In *Proceedings of the Design Automation Conference*, 1998.
- [10] L.C.V. dos Santos. *Exploiting instruction-level parallelism: a constructive approach*. PhD thesis, Eindhoven University of Technology, 1998.
- [11] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *International Conference on Parallel Processing*, 1993.
- [12] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, March 1992.
- [13] 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.
- [14] P. G. Paulin and J. P. Knight. Force-directed scheduling in automated data path synthesis. In *Design Automation Conference*, 1987.
- [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] AT&T Research Labs. Graphviz - open source graph drawing software. <http://www.research.att.com/sw/tools/graphviz/>.

- [17] UC Irvine's Spark High Level Synthesis System. Spark benchmark ftp site.
<ftp://ftp.ics.uci.edu/pub/spark/benchmarks>.
- [18] Synopsys Incorporated. Design compiler. <http://www.synopsys.com/products/logic/logic.html>.