# Dynamic Conditional Branch Balancing during the High-Level Synthesis of Control-Intensive Designs*

Sumit Gupta† Nikil Dutt† Rajesh Gupta§ Alex Nicolau†

†Dept. of Information and Computer Science
University of California at Irvine
{sumitg, dutt, nicolau}@cecs.uci.edu

§Dept. of Computer Science and Engineering
University of California at San Diego
gupta@cs.ucsd.edu

http://www.cecs.uci.edu/∼spark

## Abstract

*We present two novel strategies to increase the scope for application of speculative code motions: (1) Adding scheduling steps dynamically during scheduling to conditional branches with fewer scheduling steps. This increases the opportunities to apply code motions such as conditional speculation that duplicate operations into the branches of a conditional block. (2) Determining if an operation can be conditionally speculated into multiple basic blocks either by using existing idle resources or by creating new scheduling steps. These strategies lead to balancing of the number of steps in the conditional branches without increasing the longest path through the conditional block. Algorithms for these strategies have been implemented within the* Spark *high-level synthesis framework that accepts a behavioral description in ANSI-C as input and produces synthesizable register-transfer level VHDL. Experiments on two moderately complex industrial-strength applications, namely, MPEG-1 and the GIMP image processing tool, demonstrate that conditional speculation is ineffective without using these strategies.*

## 1. Introduction

The ordering and placement of operations in high-level behavioral descriptions is often governed by programming ease and how the overall behavior is conceptualized. Very often such control is not conducive to or optimal for downstream high-level synthesis and optimization tasks. This is particularly true of control-intensive designs due to the presence of nested conditionals and loops. An important aspect of our approach to high-level synthesis is the application of parallelizing transformations that move operations across conditionals and loops based on the time criticality of an operation and in the process, expose the parallelism available in the algorithm.

To this end, a set of speculative code motions have been proposed to alleviate the effects of programming styles and constructs on the quality of synthesis results. These code motions enable the movement of operations through, beyond, and into conditionals with the objective of maximizing performance [1, 2, 3]. However, this means that the heuristics that guide these code motions have to carefully manage the resource utilization across several basic blocks. This is especially true for expensive code motions such as *conditional speculation* that lead to operation duplication. This code motion should only be enabled when the resource utilization techniques are able to find idle resources in multiple basic blocks in the conditional branches.

In this paper, we present algorithms for a set of strategies that insert new scheduling steps, dynamically during scheduling, in the shorter of the two branches of a conditional block without increasing the longest path through the conditional. These new scheduling steps along with idle resources in the basic blocks of the other conditional branch can be used to schedule operations by conditional speculation. Another algorithm inspects the resource utilization of multiple basic blocks in conditional branches before applying conditional speculation. When an idle resource cannot be found in an already scheduled basic block, new scheduling steps are inserted if the basic block is part of a shorter conditional branch.

We have implemented these resource utilization improvement techniques, along with the speculative code motions and scheduling heuristics that employ them, in a high-level synthesis framework called *Spark*. We demonstrate the utility of these techniques by presenting results for experiments performed on two large industrial strength designs derived from the multi-media and image processing domains.

This paper builds upon our earlier presentations in [2, 3] where we introduced the individual speculative code motions that can be applied for improved resource utilization. In this paper, we introduce the notion of dynamic branch balancing and the heuristics to guide the speculative code motions for improved quality of synthesis results.

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

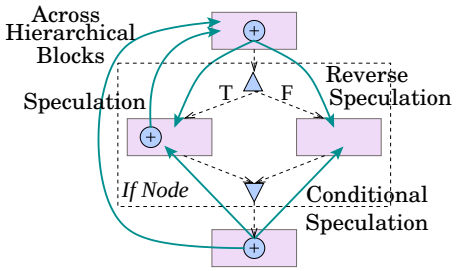


Figure 1. **Various speculative code motions**

The rest of this paper is organized as follows: the next section reviews previous work, followed by an overview of the speculative code motions. We present the scheduling step insertion strategies in Sections 4 and 5, followed by our experimental setup and results.

## 2. Related Work

High-level synthesis has been a subject of research for over two decades [4]. Recent work has presented speculative code motions for mixed control-data flow type of designs. CVLS [5] uses condition vectors to improve resource sharing among mutually exclusive operations. Radivojevic et al. [6] present an exact symbolic formulation which generates an ensemble schedule of valid, scheduled traces. The "Waveschedule" approach [7] incorporates speculative execution into high-level synthesis to achieve its objective of minimizing the expected number of cycles. Recent work by Rim [8], Santos [1] and Kountouris [9] supports generalized speculative code motions for scheduling in high-level synthesis.

A range of similar parallelizing code transformation techniques have been previously developed for software compilers (especially parallelizing compilers) [10, 11]. 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 by taking into account hardware cost models and mutual exclusivity of operations.

Compilers traditionally focus on minimizing the compensation code overheads while applying code motion techniques that lead to operation duplication [11, 12]. On the other hand, we will demonstrate that in high-level synthesis we are willing to tolerate operation duplication as long as it does not increase the cycles on the longest path through the design.

## 3. Speculative Code Motions

We have previously developed a set of code motion transformations that re-order operations to minimize the effects of syntactic variance in the input description. These beyond-basic-block code motion transformations are usually speculative in nature and attempt to extract the inherent parallelism in designs and increase resource utilization.

Generally, speculation refers to the unconditional execution of operations that were originally supposed to have executed conditionally. However, frequently there are situations when there is a need to move operations *into* conditionals [2, 3]. This may be done by *reverse speculation*, where operations before conditionals are moved into *subsequent* conditional blocks and executed conditionally, or this may be done by *conditional speculation*, wherein an operation from after the conditional block is duplicated *up* into *preceding* conditional branches and executed conditionally. Reverse speculation can be coupled with *early condition execution*; here conditional checks are evaluated as soon as possible, so that the operations in their branches do not have to be speculated for scheduling [2].

The various speculative code motions are shown in Figure 1. Also, shown is the movement of operations across entire hierarchical blocks, such as if-then-else blocks or loops. In the next few sections, we present resource utilization management algorithms that increase the scope of application of these code motions.

## 4. Inserting Scheduling Steps to Balance Conditional Branches

Design descriptions often have instances where there exist more operations in one branch of a conditional block than the other. This situation is depicted by the synthetic example shown in Figure 2(a). The example in this figure is represented using the *hierarchical task graph* (HTG) representation [13, 2]. HTGs model the design as a set of hierarchical nodes that represent if-then-else blocks, for-loops, while-loops and so on. These hierarchical nodes consist of *basic blocks*, which encapsulate a sequence of scheduling steps with no control flow between them. A *scheduling step* is an aggregation of operations that execute concurrently.

While scheduling the design in Figure 2(a), our scheduler schedules the true branch, i.e., basic block $BB_2$ first, followed by the false branch, i.e., $BB_3$. So, if the design in this example is allocated an adder and a subtracter, then the resulting design after scheduling is as shown in Figure 2(b).

This figure shows that, after scheduling, the false branch of the if-then-else HTG node has fewer scheduling steps than the true branch. This is known as an if-HTG with *unbalanced* conditional branches. In such unbalanced if-HTGs, it is possible to insert a new scheduling step in the branch with fewer scheduling steps – in this example, basic block $BB_3$ – without affecting the longest path through the if-HTG. This new step and the presence of a corresponding idle resource in the other branch ($BB_2$) of the if-HTG node, enables the conditional speculation of operation "$e$", as operations "$e_1$" and "$e_2$" in basic blocks $BB_2$ and $BB_3$ respectively, as shown in Figure 2(c).

If we had not inserted this new scheduling step, we would not have been able to conditionally speculate operation $e$. The state assignments, $S0$, $S1$ and $S2$, as also shown by dashed lines in the designs in Figure 2(a), (b) and (c). Clearly, inserting the new scheduling step and the subsequent conditional speculation of

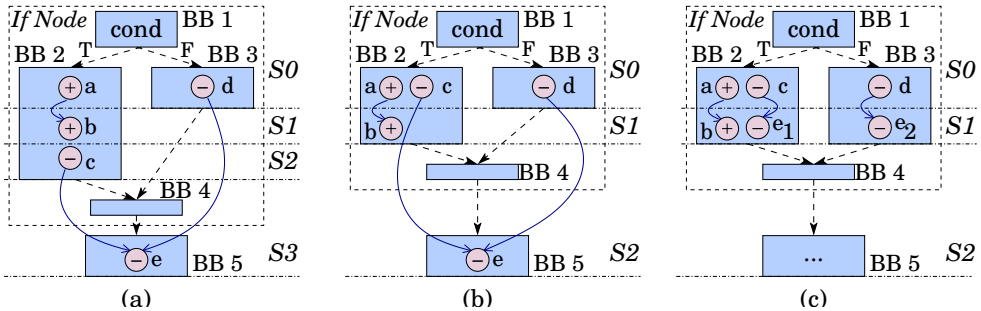


Figure 2. **(a) HTG representation of an example, (b) After scheduling basic block $BB_2$, (c) Insertion of a new scheduling step in basic block $BB_3$ enables conditional speculation of operation $e$.**

operation $e$ reduces the states required to schedule the design (and in case the schedule length). Also, since the longest path through the if-HTG is unaltered, this technique can never lead to an increase in longest path length through the design.

The next section presents an algorithm that employs this concept to dynamically insert new scheduling steps into conditional branches during scheduling.

### 4.1. Incorporating Conditional Branch Balancing into High-Level Synthesis

The scheduling heuristic in our system schedules the design by traversing the HTG of the design in a top-down manner, starting from the first basic block in the design and terminating at the last basic block. The scheduler calls the algorithm outlined in Figure 3 to get the steps to schedule in the design. The scheduling step insertion technique is incorporated into this algorithm as shown by the boxed section.

The algorithm in this figure starts by determining the current basic block, *currentBB*, that the current scheduling step, *step*, is in. If this is the first call to the algorithm (i.e. *step* is $\phi$), then the *currentBB* is assigned as the first basic block in the top level *HTG* of the design. The next scheduling step is the scheduling step after the current *step* in *currentBB*, starting with the first step in the basic block (line 6 in the algorithm).

Next, the algorithm checks if *nextStep* is empty, i.e., the last scheduling step in *currentBB* has been reached. The algorithm should then get the next basic block to schedule, however, it is at this point that the algorithm inserts a new scheduling step if the current basic block is in the shorter branch of an unbalanced conditional block (lines 7 to 12 in algorithm).

Hence, as shown in line 8, the algorithm first determines if the *currentBB* has a complementary basic block, *complementBB*. A *complementBB* exists if *currentBB* is in a if-HTG node; if the *currentBB* is in the true branch, then its *complementBB* is the false branch and vice versa. If a *complementBB* exists *and* if it has already been scheduled *and* it has more scheduling steps than *currentBB*, then the algorithm creates a new scheduling step in *currentBB* (lines 9 through 11). Note that, if profiling information is available, we can instead insert scheduling steps to basic blocks in branches that are less likely to be taken.

*Algorithm 1: Get Next Scheduling Step*
Inputs: *HTG* of design, Current Scheduling "*step*"
Output: Next Scheduling Step "*nextStep*"

```
1: if (Scheduling step step = φ) then
2:    currentBB = getFirstBasicBlock(HTG)
3: else
4:    currentBB = getBasicBlockOf(step)
5: endif
6: nextStep = Scheduling step after step in currentBB
7: if (nextStep = φ) then
8:    complementBB = getComplement(currentBB)
9:    if (complementBB ≠ φ and is scheduled) then
10:      if (numOfStepsInBB(currentBB) <
             numOfStepsInBB(complementBB)) then
11:       nextStep = createNewStepInBB(currentBB)
12: endif        /* Balance Conditional Branches */
13: if (nextStep = φ) then
14:    nextBB = getNextBasicBlock(currentBB)
15:    if (nextBB ≠ φ) then
16:       nextStep = First schduling step in nextBB
17: endif
18: return nextStep
```

Figure 3. **The algorithm to get the next step to schedule. The boxed section adds new scheduling steps in shorter conditional branches**

If a new scheduling step is not created in the *currentBB* and the *nextStep* is still empty (line 13), then the algorithm proceeds to get the next basic block in the HTG by calling the *getNextBasicBlock* function. This function (not shown here) traverses first the "true" control path from the current basic block, then the "false" control path. False control paths exist only for basic blocks that contain a conditional check. If a new basic block is found by this function, its first scheduling step is assigned to *nextStep* (line 16).

The function *getNextBasicBlock* visits each basic block in the design once. Hence, the algorithm in Figure 3 visits each step, including the newly added steps, in each basic block in the HTG of the design once. Since while capturing the initial description by HTGs, we create a scheduling step for each operation in the design, hence, the complexity of this algorithm is in the order of O($N_{Ops}$), where $N_{Ops}$ is the number of operations in the HTG.

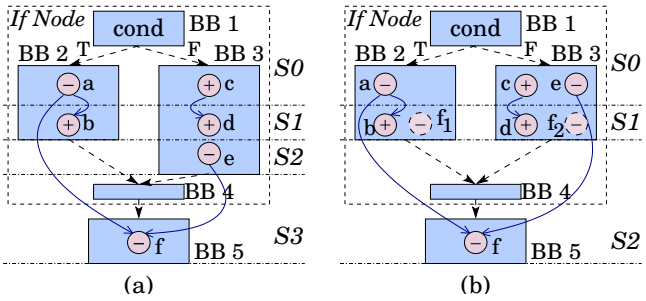


Figure 4. **(a) An example HTG (b) After scheduling basic blocks $BB_2$ and $BB_3$, it becomes possible to conditionally speculate operation "$e$"**

To understand the reason *complementBB* is checked for being scheduled, consider the example in Figure 4(a). In this example, if this check is not done, then a new scheduling step will be added while scheduling basic block $BB_2$, since it has one less scheduling step than its complementary basic block, $BB_3$. However, after scheduling both basic blocks, the number of scheduling steps is the same in both branches of the if-HTG, as shown in Figure 4(b). However, in effect, this check means that new scheduling steps are only added to the false branch by this technique. In the next section, we present a technique that inserts scheduling steps in both conditional branches.

## 5. Utilizing Idle Resources in Already Scheduled Basic Blocks

An operation can be conditionally speculated only if there are idle resources in the basic blocks that comprise the conditional branches of a if-HTG. However, if there are no idle resources, it may be possible to insert new scheduling steps if the if-HTG has unbalanced conditional branches. Also, the checks required to determine if conditional speculation is possible can be performed accurately only while scheduling the last (false) branch of the conditional block. It is only then that the actual resource utilization and the empty resource slots of all the basic blocks in both the conditional branches are known.

This can be demonstrated by the example in Figure 4(a). In this example, when the basic block $BB_2$ is being scheduled, the scheduling heuristic can conditionally speculate operation "$f$" into scheduling step $S1$ in $BB_2$. However, in basic block $BB_3$, operation "$f$" depends on operation "$e$" and in the initial description in Figure 4(a), operation "$e$" is placed in the scheduling step $S2$. Hence, the heuristic will determine that it is not possible to accommodate the copy of operation "$f$" in basic block $BB_3$, since it will lead to the addition of a new scheduling step in the already larger conditional branch ($BB_3$).

However, the above decision is made too early. After the heuristic has finished scheduling basic block $BB_3$, it actually turns out that the operation "$e$" can be scheduled earlier in scheduling step $S0$ in basic block $BB_3$, as shown in Figure 4(b). This means that operation $f$ can actually be conditionally speculated into the idle slots in scheduling step $S1$ in both basic blocks, $BB_2$ and $BB_3$. This is shown by the shaded operations, $f_1$ and $f_2$, in Figure 4(b).

*Algorithm 2: Allow Conditional Speculation of op ?*
Inputs: Operation *op*, Scheduling *step* in $BB_{step}$, Basic block List $BBList$ to which *op* will be duplicated if it is scheduled at *step*
Output: Whether to conditionally speculated *op*
1: **foreach** (Basic block *bb* in $BBList$) **do**
2: **if** (isThereIdleResourceInBB(*bb*, *op*) == false)
3: needNewSchedingStep = true
4: **if** (stepsIn(*bb*) $\geq$ stepsIn($BB_{step}$) ) then
5: **if** (needNewSchedingStep == true) **or**
6: (isBBScheduled(*bb*) == false)
7: return from function with **false** result
8: **endforeach**
9: return from function with **true** result

Figure 5. **Algorithm to determine whether to conditionally speculate an operation *op* into scheduling *step* in basic block $BB_{step}$**

### 5.1. Determining whether to allow Conditional Speculation

An algorithm that determines if operation *op* should be conditionally speculated based on available idle resources, is outlined in Figure 5. This algorithm starts with the list of basic blocks ($BBList$) into which an operation *op* will have to be duplicated, if it were to be scheduled on the scheduling step, *step*, in basic block $BB_{step}$. $BBList$ may be greater than one when the branches of the conditional block under consideration have other nested if-HTGs in them. Nested $ifs$ are fairly common in the type of control-intensive designs targeted by this work.

For each basic block *bb* in the $BBList$, the algorithm checks if there is an idle resource to schedule operation *op* on, as shown in line 2 in Figure 5. The algorithm for this check is presented in the next section. If there is no idle resource in *bb*, then a flag is set that signifies that a new scheduling step will have to be created in the basic block *bb* to accommodate operation *op*.

Next, as shown in lines 4 to 7 in Figure 5, the algorithm does not allow conditional speculation, if, (a) the current basic block *bb* already has as many or more steps than $BB_{step}$ *and*, (b) a new step will have to be created in *bb* to accommodate *op or* (c) if *bb* is unscheduled. The restriction on the new scheduling step is to prevent *bb* from becoming the longer of the two branches in the conditional block and the last restriction prevents incorrect decisions made without scheduling both the branches of the conditional block.

This algorithm is used by the scheduler to determine only whether conditional speculation is possible or not. Once the decision to conditionally speculate an operation has been made, a similar algorithm is used by the scheduler to identify idle resources. If no idle resource is found, the algorithm inserts a new scheduling step.

*Algorithm 3: Is There Idle Resource in Basic Block*
Inputs: Operation *op*, Basic Block *bb*
Output: Whether there is idle resource for *op* in *bb*
1: *matchingResList* = findResourcesForOp(*op*)
2: *currStep* = stepInBBAfterDataDependencies(*bb*, *op*)
3: **while** (*currStep* $\neq \phi$) **do**
4: **foreach** (*res* in *matchingResList*) **do**
5: **if** (isResourceIdleInStep(*step*, *res*) == true)
6: *numSteps* = numOfCyclesTakenByRes(*res*) - 1
7: *prevSteps* = getPrevSteps(*step*, *numSteps*)
8: *succSteps* = getSuccSteps(*step*, *numSteps*)
9: **if** (*res* not used in *prevSteps* **and** *succSteps*)
10: return *true*
11: **endforeach**
12: *currStep* = next step after *currStep* in *bb*
13: **endwhile**
14: return *false*

Figure 6. **Determining if there is an idle resource in basic block *bb* for scheduling operation *op***

Also, this algorithm is capable of adding new scheduling steps in both branches of a conditional block. For example, in the design in Figure 4, if basic block $BB_2$ had only one scheduling step, this algorithm would have added a new step and still conditionally speculated operation "$f$".

### 5.2. Finding an Idle Resource in a Basic Block

The algorithm to find an idle slot for an operation *op* in a basic block *bb* is outlined in Figure 6. This algorithm starts by determining the list of resources (*matchingResList*) on which the operation *op* can be scheduled. It then finds the first scheduling step in *bb* that does not have an operation with a data dependency with *op* by calling the function *stepInBBAfterDataDependencies* (not shown here).

Using this scheduling step (*currStep*) as a starting point, the algorithm determines if there is an idle resource in this step or any of its successor steps in basic block *bb* (shown by while loop in Figure 6). Each resource *res* in *matchingResList* in *currStep* is checked to see if it is idle, i.e., there is no operation scheduled on it and hence, it is potentially available for scheduling the operation *op* (line 5 in the algorithm).

However, the current resource (*res*) being checked may be a multi-cycle resource. Hence, the scheduling steps before and after the current step have to be checked to make sure that the resource is idle in them for the duration of its execution time starting in *currStep*. First, the number of steps that need to be checked is calculated (*numSteps*); this is one less than the execution cycles of the resource.

The algorithm then gets *numSteps* predecessor steps and *numSteps* successor steps (lines 6 to 8 in Figure 6). Note that these predecessor and successor steps can, and frequently are, in other basic blocks and hence, the resource utilization of the resource *res* has to be checked beyond the current basic block.

If the resource *res* is not used in any of these predecessor and successor steps, then an idle resource slot has been found in the current step and the algorithm terminates with a true result. However, if it is used in any of these steps, then the next resource in the *matchingResList* is checked and so on. This is done for all the steps following *currStep* in the given basic block *bb*, until either an idle resource slot is found or no more steps are left in *bb*.

In this way, in the worst case, the two algorithms presented in Figures 5 and 6 visit each matching resource in each step in each of the basic blocks that the operation will be duplicated into. However, some of the information calculated by these functions can be cached to be used by later calls to the same function.

## 6. Experimental Setup and Results

The dynamic conditional branch balancing algorithms presented in this paper have been implemented in a high-level synthesis research framework called *Spark* [2]. This synthesis framework takes a behavioral description in ANSI-C as input and generates synthesizable register-transfer level VHDL. Besides the speculative code motions, several standard compiler transformations such as CSE, copy and constant propagation and dead code elimination are also implemented in the *Spark* framework.

For our experiments, we have chosen two functions from the *Prediction* block of the MPEG-1 application [14], namely, *pred*0_1 and *pred*2, and the *tile* function (with the *scale* function inlined) derived from the "tiler" transform[1] in the GIMP image processing tool [15]. The run time of our system for these designs is less than 5 user seconds on a 1.6 Ghz PC running Linux.

For all the experiments we have used a priority-based list scheduler [16]. The scheduling results for these three functions are presented in Table 1, in terms of the number of states in the finite state machine controller and the cycles on the longest path through the design. The longest path through a conditional is the longer of the two branches and for loops, its the length of the loop body multiplied by loop iterations. The tables also give the number of operations, non-empty basic blocks, If blocks, and the resources used for scheduling; $+-$ does add and subtract, == is a comparator, $*$ a multiplier, / a divider, [ ] an array address decoder and << is a shifter. All resources are single cycle except the multiplier (2 cycles) and the divider (4 cycles).

The first row in Table 1 lists the results for the baseline case, i.e., when all the code motions from Figure 1 are enabled *except* conditional speculation. The second row has conditional speculation (CS) enabled along with the rest of the code motions. The percentage reduction over baseline case is given in parentheses. In the third row, all the code motions including CS are

[1]Note that this floating point function has been arbitrarily converted to an integer function here. This does not affect the nature of the control flow, but only the way the data is handled.

| Strategy Applied | *pred*2(217 Ops,45 BBs,11 IFs) $2+-, 1*, 2<<, 2==, 2[]$ | | *pred*0_1(101 Ops,26 BBs,4 IFs) $2+-, 1*, 2<<, 2==, 2[]$ | | *tiler*(145 Ops,35 BBs, 11 IFs) $3+-, 1*, 1/, 2<<, 2==, 2[]$ | |
|---|---|---|---|---|---|---|
| | # States | Long Path | # States | Long Path | # States | Long Path |
| Baseline | 157 | 7220 | 175 | 3254 | 74 | 6731 |
| +Allow CS | 152(-3.2 %) | 7215(-0.1 %) | 168(-4 %) | 3249(-0.1 %) | 59(20.3 %) | 5131(-23.8 %) |
| CS+Add Steps | 125(-17.8 %) | 5613(-22.2 %) | 140(-16.7 %) | 2838(-12.7 %) | 55(-6.8 %) | 4731(-7.8 %) |
| CS+CS Algo | 119(-21.7 %) | 4718(-34.6 %) | 134(-20.2 %) | 2576(-20.7 %) | 53(-10.2 %) | 4631(-9.7 %) |
| CS+Both Algo | 112(-26.3 %) | 4270(-40.8 %) | 127(-24.4 %) | 2249(-30.8 %) | 52(-11.9 %) | 4431(-11.7 %) |
| **Total Reduct.** | **-28.7 %** | **-40.9 %** | **-27.4 %** | **-30.9 %** | **-29.7 %** | **-34.2 %** |

Table 1. **Scheduling results after applying the various resource utilization strategies for** *pred*2 **and** *pred*0_1 **from the MPEG-1 Prediction block and** *tiler* **from the GIMP image processing tool**

enabled along with the algorithm that adds scheduling steps (Algorithm 1 in Figure 3). In the fourth row, all the code motions along with the algorithms that control CS are enabled (Algorithms 2 and 3 in Figures 5 and 6). The fifth row has all the algorithms enabled along with all the code motions. The percentage reductions of the 3rd, 4th and 5th row over the 2nd row (with none of the algorithms enabled) are given in parentheses. The last row gives the total reduction of the 5th row over the baseline case.

These results demonstrate that the algorithms presented in this paper have to enabled to make conditional speculation (CS) truly effective. This is especially evident for the MPEG functions; with CS alone, the improvements are less than 4 %. With these algorithms enabled, the improvements for both metrics for all three designs range from 6 % to 34 %, as seen from the results in the 3rd and 4th rows. Also, we find that the algorithms that determine whether to conditionally speculate (4th row) lead to larger improvements in both the metrics than the first algorithm (3rd row).

The improvements by applying both the algorithms (5th row) over only enabling CS (2nd row) can range between 11 to 26 % in the number of states and 11 to 40 % for the longest path cycles. Furthermore, the results in the 3rd to 5th rows indicate that to some extent these algorithms are complementary. The total reductions (last row) with both algorithms enabled over the baseline case are between 27 to 29 % for controller size and 30 to 40 % for performance.

## 7. Conclusions and Future Work

We presented a set of strategies that insert scheduling steps in the shorter of the two branches of a conditional block without increasing the cycles on the longest path through the design. These strategies are critical to effectively and judiciously use code motions such as conditional speculation that duplicate operations into multiple basic blocks. Also, if profiling information is available, these algorithms can easily be modified to add scheduling steps only in the branches that are less likely to be taken. Results for two real-life multimedia and image processing applications show improvements of up to 40 % in cycles on the longest path and 29 % in controller size when the dynamic conditional branch balancing strategies are enabled.

## References


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

[2] S. Gupta et al. Speculation techniques for high level synthesis of control intensive designs. In *Design Automation Conference*, 2001.

[3] S. Gupta et al. Conditional speculation and its effects on performance and area for high-level synthesis. In *Intl. Symp. on System Synthesis*, 2001.

[4] D.D.Gajski et al. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer, 1992.

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

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

[7] G. Lakshminarayana et al. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. *DAC*, 1998.

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

[9] A.A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviors for high-level synthesis. *TODAES*, July 2002.

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

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

[12] L.C.V. dos Santos. A method to control compensation code during global scheduling. In *Workshop on Circuits, Systems and Signal Processing*, 1997.

[13] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on PDS*, Mar. 1992.

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

[15] GNU Image Manipulation Program. http://www.gimp.org.

[16] S. Gupta et al. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. *Intl. Conf. on VLSI Design*,2003.