Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits

S. Gupta, N. Dutt, R. Gupta and A. Nicolau

Abstract: The quality of high-level synthesis results for designs with complex and nested conditionals and loops can be improved significantly by employing speculative code motions. Two techniques are presented that add scheduling steps to the branch of a conditional construct with fewer scheduling steps. This 'balances' or equalises the number of scheduling steps in the conditional branches and increases the scope for application of speculative code motions. These branch balancing techniques have been applied 'dynamically' during scheduling. The authors have implemented algorithms for dynamic branch balancing techniques in a C-to-VHDL high-level synthesis framework called *Spark*. The utility of these techniques is demonstrated by experimental results on four designs derived from two moderately complex applications, namely, MPEG-1 and the GIMP image processing tool. These results show that the two branch balancing techniques can reduce the cycles on the longest path through the design by up to 38% and the number of states in the controller by up to 37%.

1 Introduction

The ordering and placement of operations in high-level behavioural descriptions is usually governed by programming ease and varies from designer to designer. Very often this ordering is not conducive to, or optimal for, downstream high-level synthesis and optimisation tasks [1]. This is particularly true for 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 parallelising 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, we have developed a set of speculative code motions to alleviate the effects of programming styles and constructs on the quality of synthesis results. These code motions enable the movement of operations through, across and into conditionals with the objective of maximising performance [2, 3]. However, this means that the heuristics that guide these code motions have to manage the resource utilisation across several basic blocks. This is especially true for hardware-expensive code motions, such as conditional speculation. Conditional speculation duplicates operations into the branches of a conditional block (see Section 4) [3]. Conditional speculation should only be employed when the resource utilisation techniques are able to find idle or unused resources in multiple basic blocks in the conditional branches.

In this paper, we present algorithms for two techniques 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 [4]. The new scheduling steps, together with idle resources in the basic block of the other conditional branch, can be used to schedule operations by conditional speculation. The first technique inserts scheduling steps while traversing the design during scheduling, and the second technique inserts step to enable code motions (specifically conditional speculation). We call these techniques *branch balancing during design traversal* (BBDDT) and *branch balancing during the code motions* (BBDCM), respectively. We explain these techniques in detail with example in Sections 7 and 8.

We have implemented these branch balancing algorithms, together with the speculative code motions and scheduling heuristics that employ them, in a high-level synthesis framework called Spark. This paper demonstrates the utility of these techniques by presenting results for experiments performed on four large industrial strength designs derived from multimedia and image processing domains.

The paper builds on our earlier presentations [2, 3] in which we introduced the individual speculative code motions that can be applied for improving resource utilisation and hence synthesis results. It introduces the notion of dynamic branch balancing and the heuristics to guide the speculative code motions for improving the quality of synthesis results.

2 Related work

High-level synthesis has been a subject of research for over two decades [5]. Recent work has presented speculative code motions for mixed control data flow type of designs. CVLS [6] uses condition vectors to improve resource

IEE Proceedings online no. 20030839

doi: 10.1049/ip-cdt:20030839

Paper received 27th May 2003

S. Gupta, N. Dutt and A. Nicolau are with the School of Information and Computer Science, University of California at Irvine

R. Gupta is with the Department of Computer Science and Engineering, University of California at San Diego

sharing among mutually exclusive operations. Radivojevic and Brewer [7] presented an exact symbolic formulation, which generates an ensemble schedule of valid, scheduled traces. The 'Waveschedule' approach [8] incorporates speculative execution into high-level synthesis to achieve its objective of minimising the expected number of cycles. Recent work by Rim *et al.* [9] and dos Santos and Jess [10] supports generalised speculative code motions for scheduling in high-level synthesis. Kolling et al. [11] present a scheduling heuristic based on distribution graphs (similar to force-directed scheduling) that is capable of scheduling designs with conditional branches.

A range of similar parallelising code transformation techniques has been previously developed for software compilers (especially parallelising compilers) [12, 13]. 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 minimising the compensation code overheads while applying code motion techniques that lead to operation duplication [13, 14]. On the other hand, in high-level synthesis operation duplication can be tolerated as long as it does not increase the cycles on the longest path through the design.

3 Model and terminology

We use the following terminology in this paper: A *scheduling step* is an aggregation of operations that execute concurrently. A sequence of scheduling steps with no control flow between them is encapsulated in a *basic block*. We capture the control flow between basic blocks using a hierarchical intermediate representation called *hierarchical task graphs* (HTGs) [15, 16]. HTGs model the design with three type of nodes: (*a*) *single* nodes that encapsulate basic blocks, (*b*) *compound* nodes that are hierarchical in nature and encapsulate conditional constructs such as if-then-else blocks and switch-case blocks, and (*c*) *loop* nodes that encapsulate for-loops, while-loops etc.

An example of hierarchical task graph representation of an if-then-else conditional construct is shown in Fig. 1*a*. As shown in this figure, an if-then-else or If-HTG consists of a condition basic block, compound HTG nodes for the true and false branches and an empty basic block for the merge or join of the conditional branches. Similarly, the HTG representation of a For-loop is shown in Fig. 1*b*. A For-HTG consists of an optional initialisation basic block ($i = 0$), a condition check basic block ($i < N$), a compound HTG node for the loop body, an optional increment basic block ($i = i + 1$), and an empty basic block for the loop exit. In this Figure, basic block BB_1 is the conditional basic block of the if-then-else (or If-HTG), BB_2 and BB_3 are the true and false branches respectively and BB_4 is the join basic block of the If-HTG (i.e. where the control flow in the If-HTG merges). This Figure also shows the operations and the data flow between them.

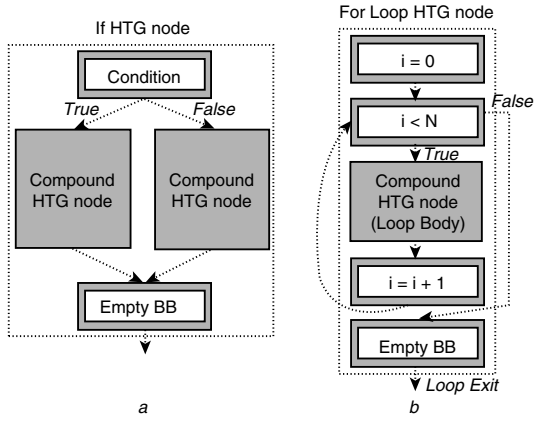

Fig. 1 *Example HTG representation*

a If-then-else conditional block and
b a For loop

Also note that we say a resource is *idle* in a scheduling step when there is no operation scheduled on the resource in that scheduling step.

4 Speculative code motions

We have previously developed a set of code motion transformations that reorder operations to improve the synthesis results in designs with complex control flow. These beyond-basic-block code motion transformations are usually speculative in nature and attempt to extract the inherent parallelism in designs and increase resource utilisation.

Generally, speculation refers to the unconditional execution of operations that were originally supposed to have executed conditionally. However, frequently there are situations in which 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 it may be done by *conditional speculation*, in which 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* in which conditional checks are evaluated as soon as possible, so that the operations in their branches do not have to be speculated for scheduling.

The various speculative code motions are shown in Fig. 2. Also shown is the movement of operations across entire hierarchical blocks, such as if-then-else blocks or loops.

5 Enabling new code motions by branch balancing

Often design descriptions are structured so that one conditional branch in an if-then-else HTG node has fewer scheduling steps than the other. We call this an If-HTG with *unbalanced* conditional branches. Consider the input description shown in Fig. 3*a*. One possible scheduled design (with a resource allocation of an adder and a subtracter) is as shown in Fig. 3*b*: operations *a* and *c* execute concurrently in state *S*0 in basic block BB_2. The state assignments (*S*0, *S*1, and so on) are demarcated by broken lines in these Figures. We can see from Fig. 3*b* that, after scheduling this example, the false branch (BB_3)

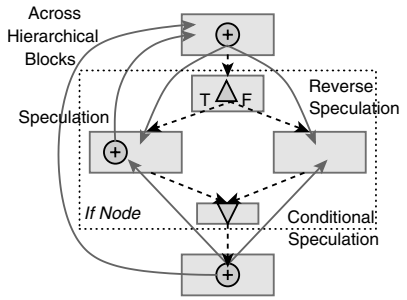

Fig. 2 *Various speculative code motions*

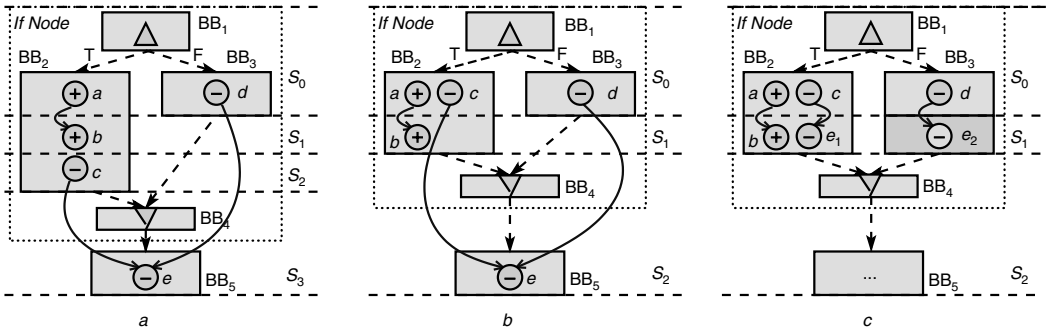

Fig. 3

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*

of the If-HTG node has fewer scheduling steps than the true branch (BB_2). Thus, *If Node* is an If-HTG in Fig. 3*b* 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, without increasing the length of the longest path through the If-HTG. Hence, in the scheduled design in Fig. 3*b*, we can insert a new scheduling step in basic block BB_3 since BB_2 has more scheduling steps than BB_3. This new step and the presence of a scheduling step in BB_2 with an idle subtracter enables the conditional speculation of operation *e*, as operations e_1 and e_2 in basic blocks BB_2 and BB_3, respectively. The resulting design is shown in Fig. 3*c*.

The design in Fig. 3*c* requires one state less to execute than the scheduled design in Fig. 3*b*. Thus, branch balancing can introduce new opportunities for applying conditional speculation and thus, further compact the design schedule. Also, since the longest path through the If-HTG is unaltered, this technique does not lead to an increase in longest path length through the design. 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.

6 Incorporating conditional branch balancing into a high-level synthesis scheduler

To enable code motions, branch balancing has to be employed *dynamically* during scheduling. If branch balancing is applied after scheduling, it is too late to affect scheduling decisions. Conversely, branch balancing cannot be applied before scheduling since the number of scheduling steps in the branches of the conditional block is known only after scheduling them.

Figure 4 shows the overall architecture of the scheduler in our synthesis framework. The components of this scheduler framework are:

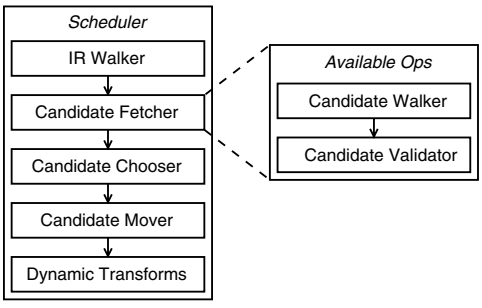

Fig. 4 *Architecture of the scheduling heuristics in the Spark framework*

(*a*) An *IR* (*intermediate representation*) *Walker* that traverses the design and returns the next step and basic block to schedule.
(*b*) A *Candidate Fetcher* that itself consists of two components:
(i) A *Candidate Walker* that traverses the design and finds the unscheduled operations that are candidates for scheduling on the current step being scheduled. These candidate operations are called *Available Operations*.
(ii) A *Candidate Validater* that removes those unscheduled available operations whose data dependencies are not satisfied or that cannot be moved to the current step being scheduled.
(*c*) A *Cost Function* that calculates the cost of each candidate in the available operations list. The scheduler then picks the operation with the lowest cost.
(*d*) A *Candidate Mover* that moves the chosen operation from its current basic block to the current step being scheduled.
(*e*) A *Dynamic Transformations* pass that applies low level compiler optimizations such as common subexpression elimination (CSE) and copy propagation dynamically during scheduling, based on the new position and possible duplication of the scheduled operation [17].

We perform dynamic branch balancing during two tasks of the scheduler:

1 *Branch Balancing during Design Traversal* (BBDDT): The IR walker traverses the design in a top-down manner starting from the first basic block in the design. It traverses the control-flow graph of the design in a topologically manner until all the basic blocks have been visited (i.e. scheduled). During this design traversal, we balance the branches of unbalanced conditional blocks as they are encountered.
2 *Branch Balancing during Code Motions* (BBDCM): The candidate mover can call the branch balancing algorithm to insert new scheduling steps in unbalanced conditional blocks if this enables a code motion required to move the candidate operation. This means that during the candidate validater task, we validate operations that can be moved if branch balancing is employed.

7 Branch balancing during design traversal

Our high-level synthesis scheduler calls the function *GetNextSchedulingStep* to get the steps to schedule in the design. The algorithm for this function is outlined in Fig. 5. This algorithm takes as input the current scheduling

```
GetNextSchedulingStep()
Inputs: G_HTG of design, current scheduling step currStep
Output: next scheduling step nextStep
1:  if (currStep = φ) then
2:    currentBB = GetNextBasicBlock(G_HTG, φ)
3:    return FirstStep(currentBB)
4:  else
5:    currentBB = ParentBB(currStep)
6:  endif
7:  nextStep = NextStep(currentBB, currStep)
8:  if (nextStep = φ) then
9:    nextStep = BalanceBranchesDuringTrav(G_HTG, currentBB)
10: if (nextStep = φ) then
11:   nextBB = GetNextBasicBlock(currentBB)
12:   if (nextBB ≠ φ) then
13:     nextStep = FirstStep(nextBB)
14: endif
15: return nextStep
```

Fig. 5 *Algorithm to get the next step to schedule. When all the scheduling steps in the current basic block have been scheduled, the algorithm calls the branch balancing algorithm (lines 8 and 9)*

step *currStep* and returns the next step (*nextStep*) in the design to schedule. On the first call to the algorithm (i.e. *currStep* is ϕ), the algorithm calls the *GetNextBasicBlock* function to get a basic block to schedule. Since it is also the first call to the *GetNextBasicBlock* function (not given here), it returns the first basic block in the design graph G_{HTG}. The *GetNextSchedulingStep* algorithm then returns the first step in the basic block (lines 1 to 3 in the algorithm in Fig. 5).

For subsequent calls, the *GetNextSchedulingStep* function first determines the current basic block *currentBB* that *currStep* is in. This is obtained by function *ParentBB*. *nextStep* is then the scheduling step after *currStep* in *currentBB* (line 5 in the algorithm). The algorithm then checks if *nextStep* is null; this happens when *currStep* is the last scheduling step in *currentBB*. In this case, the algorithm should traverse the design graph and get the next basic block in the design to schedule. However, it is at this point that we employ the branch balancing algorithm by making a call to the function *BalanceBranchesDuringTrav* (lines 7 and 8). This function is discussed in the next Section.

The *BalanceBranchesDuringTrav* function returns the newly created scheduling step if branch balancing is successful. This new step is then returned by the *GetNextSchedulingStep* algorithm to the scheduler. However, if the *BalanceBranchesDuringTrav* function returns a null step, *nextStep* is still null (line 10). The *GetNextSchedulingStep* algorithm proceeds to get the next basic block, *nextBB*, in the design by calling the *GetNextBasicBlock* function. The first scheduling step in the basic block returned by this function is then the *nextStep* (lines 11 to 13 in Fig. 5). The *GetNextSchedulingStep* algorithm returns this *nextStep*. If *GetNextBasicBlock* returns an empty basic block (or if *nextStep* in *nextBB* is null), this indicates to the scheduler that all the basic blocks in the design (and the scheduling steps in them) have been scheduled. The scheduler then terminates.

7.1 Algorithm for the BalanceBranchesDuringTrav function

The algorithm for the *BalanceBranchesDuringTrav* function is outlined in Fig. 6. This algorithm takes the HTG of

```
BalanceBranchesDuringTrav()
Inputs: G_HTG of design, current basic block currentBB
Output: newly created scheduling step newStep
1: newStep = φ
2: complementBB = GetComplement(G_HTG, currentBB)
3: if (complementBB ≠ φ and IsScheduled (complementBB)) then
4:   if (NumSteps(currentBB) < NumSteps(complementBB)) then
5:     newStep = CreateNewStepInBB(currentBB)
6: return newStep
```

Fig. 6 *Branch balancing during design traversal algorithm. This algorithm adds new scheduling steps in the shorter branch of a conditional block*

the design, G_{HTG} and the current basic block *currentBB* as input. The algorithm starts by determining the complementary basic block *complementBB* of *currentBB*.

The *complementary* basic block of *currentBB* exists if *currentBB* is in an If-HTG node and is the basic block in the mutually exclusive conditional branch of *currentBB*. Hence, if the *currentBB* is in the true branch, then its *complementBB* is the false branch of vice versa.

If a *complementBB* exists and if it has already been scheduled, then we check if *complementBB* has more scheduling steps than *currentBB* (lines 3 and 4). If so, then the If-HTG has unbalanced conditional branches and the *BalanceBranchesDuringTrav* algorithm calls the function *CreateNewStepInBB* to create a new scheduling step in *currentBB* (lines 4 and 5). This new scheduling step is returned by the *BalanceBranchesDuringTrav* algorithm.

To understand why we insert scheduling steps in *complementBB* only if it is scheduled (line 3 in the algorithm in Fig. 6), consider the example in Fig. 7*a*. Suppose that we schedule the true branch of the If-HTG first. Hence, while scheduling BB_2 the algorithm detects that its complementary basic block BB_3 has more scheduling steps. However, it would be erroneous to insert a new scheduling step in BB_2 without scheduling BB_3. This is because after scheduling, BB_3 has the same number of scheduling steps as BB_2, as shown by the scheduled design in Fig. 7*b*.

The *BalanceBranchesDuringTrav* algorithm thus inserts new scheduling steps only after scheduling both the branches of a conditional. However, we can miss some scheduling opportunities due to this restriction. To overcome this limitation, we have developed a technique that inserts scheduling steps if they enable a code motion. This technique is presented in the next Section.

8 Branch balancing during code motions

Branch balancing can also be performed when moving an operation in the design. This is demonstrated by the example in Fig. 8*a*. In this example, consider that basic block BB_5 is the last conditional branch to be scheduled among the branches BB_2, BB_4 and BB_5. While scheduling BB_5, the scheduler finds that it is possible to schedule operation *f* into the second scheduling step of BB_5 by conditional speculation. However, this requires *f* to be duplicated into basic blocks BB_2 and BB_4. Although BB_2 has a resource that is unused in its second scheduling step, BB_4 does not have an idle resource.

It is at this point that we can take advantage of the fact that BB_4 is part of an unbalanced conditional branch. We can insert a new scheduling step in BB_4 since BB_4 has fewer scheduling steps than BB_2 and BB_5, as shown in Fig. 8*b*.

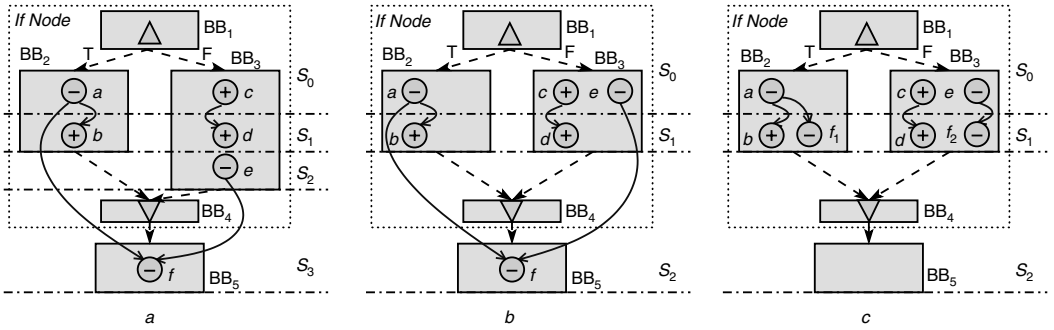

Fig. 7

a Unscheduled example; HTG representation with data flow dependencies between operations also shown
b After scheduling basic blocks BB_2 and BB_3
c It is now possible to conditionally speculate operation *f* into basic blocks BB_2 and BB_3

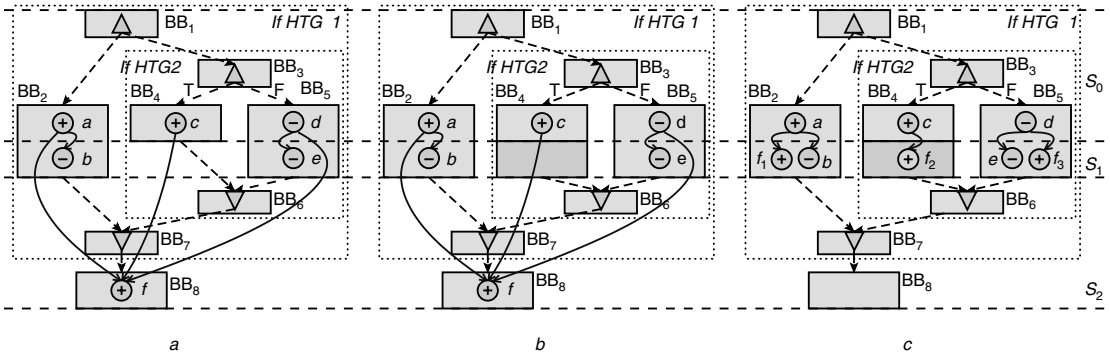

Fig. 8

a Unscheduled example
b New scheduling step is inserted in basic block BB_4
c This enables us to conditionally speculate operation *f* as operations f_1, f_2 and f_3 in basic blocks BB_2, BB_4 and BB_5

```
CanOperationBeMoved()
  Inputs: operation op, scheduling step currStep, basic block currentBB,
          basic block list BBList in which op will be duplicated to schedule at currStep
  Output: whether op can be duplicated into BBList and scheduled on currStep
1:  foreach (Basic block bb in BBList) do
2:    if (IsScheduled(bb) = = false)
3:      return false
4:    if (FindIdleResInBB(bb, op) == ϕ)
        /*No idle resource. Hence, check if we can insert a new step by branch balancing * /
5:      if (NumSteps(bb) ≥ NumSteps(currentBB)) then
6:        return false /* Not possible to insert op in bb */
7:  endforeach
8:  return true /* It is possible to insert op in all basic blocks in BBList */
```

Fig. 9 *Algorithm called by candidate validater to determine if it is possible to speculate operation op conditionally into scheduling step currStep in basic block currStep by duplicating into the basic blocks in BBList. The BBDCM technique is employed to insert new scheduling steps in basic blocks in BBList as and when required*

This enables us to speculate operation *f* conditionally as the operations f_1, f_2 and f_3 in basic blocks BB_2, BB_4 and BB_5. This resultant design is shown in Fig. 8*c*.

Thus it is possible to employ branch balancing to enable code motions. We integrate this BBDCM technique into our scheduler at two places:

(*a*) *Candidate validater*: We validate operations that can be conditionally speculated if branch balancing is employed on the conditional branches that the operation will be duplicated into.

(*b*) *Candidate mover*: This is where we perform the branch balancing if the scheduler decides to schedule an operation validated earlier on the premise of branch balancing.

The algorithm that the candidate validater calls to validate operations that require duplication for scheduling is listed in Fig. 9. This algorithm, called *CanOperationBeMoved*, takes as input 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 *CurrStep* in basic block *currentBB*. The algorithm returns a true result if it is

possible to duplicate *op* into the basic blocks in *BBList* and false otherwise.

If any basic block *bb* in the *BBList* is unscheduled, then this algorithm returns a false result (line 3 in Fig. 9). This is because we do not know the resource utilisation in an unscheduled basic block. Only after scheduling do we know the number of scheduling steps in a basic block and which resources are idle in each scheduling step.

For each scheduled basic block *bb* in the *BBList*, the algorithm calls the function *FindIdleResInBB* to find an idle resource on which operation *op* can be scheduled. This function is presented in the next section. If the *FindIdleResInBB* does not find an idle resource in *bb* to schedule *op*, then the *CanOperationBeMoved* algorithm checks if it is possible to schedule *op* in *bb* by performing branch balancing first. It thus checks if *bb* has more scheduling steps than *currentBB* (line 5 in the algorithm in Fig. 9). If this is true, then it is not possible to insert a new scheduling step into *bb* and hence we cannot schedule *op* in *bb*. The *CanOperationBeMoved* function thus returns a false result (line 6).

If the basic block *bb* either has an idle resource for *op* for *bb* has fewer scheduling steps than *currentBB*, it is possible to insert to schedule *op* in *bb*. The *CanOperationBeMoved* algorithm checks all the basic blocks in *BBList* in the same manner and returns a true result if it is possible to schedule a copy *op* in each *bb* in *BBList* either on an idle resource or by inserting a scheduling step (by branch balancing).

This algorithm is used by the scheduler during candidate validation. If the scheduler decides to schedule an operation that requires conditional speculation, a similar algorithm is used by the candidate mover to schedule *op* in each basic block *bb* in *BBList* by inserting scheduling steps if required.

8.1 Algorithm for finding an idle resource in a basic block

The algorithm to find an idle resource for an operation *op* in a basic block *bb* is outlined in Fig. 10. This algorithm starts by calling the function *FindMatchingResForOp* (not given here) to determine the list of resources, *matchingResList*, on which the operation *op* can be executed. There may be multiple resources in *matchingResList* as there may be several instances of the resource type on which *op* may execute.

The *FindIdleResInBB* function then calls the function *GetStepInBBAfterDataDeps* to find the first scheduling step in *bb* that does not have an operation with a data dependency with *op*. This function (not given here) looks for operations whose result *op* reads and that are in basic block *bb*. It then finds the last scheduling step in *bb* with any of these operations that *op* depends on and returns the next scheduling step. This returned step, *currStep*, signifies the first scheduling step in *bb* that *op* can be potentially scheduled on. Note that the ordering of scheduling steps in a basic block denotes their execution sequence.

Using this scheduling step (*currStep*) as a starting point, the *FindIdleResInBB* algorithm determines if there is an idle resource for *op* in *currStep* or any of its successor steps in basic block *bb* (shown by the while loop Fig. 10). 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* (lines 4 and 5 in the algorithm).

If *res* is idle in *currStep*, and if *res* is a multi-cycle resource, we must make sure that *res* is idle in scheduling steps before and after *currStep* for the duration of its execution. We first determine the number of steps *numSteps* that need to be checked. *numSteps* is one less than the execution cycles of the resource (line 6 in the algorithm in Fig. 10). The algorithm then calls the *GetPrevSteps* and *GetSuccSteps* functions to get *numSteps* predecessor steps and *numSteps* successor steps (lines 7 and 8 in Fig. 10). Since the predecessor and successor steps can, and frequently are, in the predecessor and successor basic blocks of *bb*, these two functions (not described here) look for steps not only in the current basic block *bb* but also may traverse to the predecessor and successor basic blocks of *bb*. Hence the resource utilisation 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, an idle resource has been found in the current step *currStep* and the algorithm terminates by returning *currStep* (lines 9 and 10 in the algorithm). However, if *res* is used in any of these steps, the procedure is repeated for the next resource in the *matchingResList* and so on. This is done for all the steps following *currStep* in the given basic block *bb*, until either a step with an idle resource is found or all the steps in *bb* have been visited.

The *FindIdleResInBB* function is called by the candidate validater and by the candidate mover. Whereas the validater only checks for idle resources in basic blocks, the candidate

```
FindIdleResInBB()
  Inputs: operation op, basic block bb
  Output: scheduling step in bb with idle resource for op
1:    matchingResList = FindMatchingRestForOp(op)
2:    currStep = GetStepInBBAfterDataDeps(bb, op)
3:    while (currStep ≠ ϕ) do
4:       foreach (res ∈ matchingResList) do
5:          if (IsResourceIdleInStep(step, res) = = true)
6:             numSteps = ExecCycles(res) – 1
7:             prevSteps = GetPrevSteps(G_HTG, currStep, numSteps)
8:             succSteps = GetSuccSteps(G_HTG, currStep, numSteps)
9:             if (res is idle in prevSteps and succSteps)
10:               return currStep
11:      endforeach
12:      currStep = NextStep(bb, currStep)
13:   endwhile
14:   return ϕ
```

Fig. 10 *Determining if there is an idle resource in basic block bb for scheduling operation op*

Table 1: Characteristics of the four designs used in our experiments along with the resources allocated for scheduling them

Benchmark	Number of ifs	Number of loops	Number of basic blocks	Number of operations	Resources
MPEG-1 *pred*1	4	2	17	123	$2+-, 2\ll, 2==, 2[\,]$
MPEG-1 *pred*2	11	6	45	287	$2+-, 2\ll, 2==, 2[\,]$
GIMP *tiler*	11	2	35	150	$3+-, 1/, 1_*, 2\ll, 2==, 2[\,]$
GIMP *nlfilt*	16	0	37	127	$3+-, 1/, 1_*, 1\ll, 1==, 1[\,]$

Table 2: Scheduling results after applying conditional speculation (CS), branch balancing during design traversal (BBDDT) and during code motions (BBDCM) for MPEG-1 *pred*1 and *pred*2

Strategy applied	MPEG-1 *pred*1 number of states	long path	MPEG-1 *pred*2 number of states	long path
Baseline (all CMs)	55	2595	112	5790
+Cond Spec (CS)	52(–5.5%)	2466(–5.0%)	106(–5.4%)	5469(–5.5%)
+CS + BBDDT	41(–25.5%)	1825(–29.7%)	85(–24.1%)	4188(–27.7%)
+CS + BBDCM	45(–18.2%)	2081(–19.8%)	92(–17.9%)	4636(–19.9%)
+CS + BBDDT + BBDCM	41(–25.5%)	1825(–29.7%)	85(–24.1%)	4188(–27.7%)

Table 3: Scheduling results after applying conditional speculations (CS), branch balancing during design traversal (BBDDT) and during code motions (BBDCM) for GIMP *tiler* and *nlfilt* designs

Strategy applied	GIMP *tiler* Number of states	long path	GIMP *nlfilt* Number of states	long path
All CMs-CS	67	6431	37	37
+Allow CS	65(–3%)	6231(–3.1%)	37(0%)	37(0%)
+CS + BBDDT	52(–22.4%)	4931(–23.3%)	33(–10.8%)	33(–10.8%)
+CS + BBDCM	47(–29.9%)	4431(–31.1%)	34(–8.1%)	34(–8.1%)
+CS + BBDDT + BBDCM	42(–37.3%)	3931(–38.9%)	33(–10.8%)	33(–10.8%)

mover schedules the operation *op* on the scheduling step *currStep* returned by the *FindIdleResInBB* function.

9 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* [16]. This synthesis framework takes a behavioural description in ANSI-C as input and generates synthesisable register-transfer level VHDL. In addition to 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 the *pred*1 and *pred*2 functions from the prediction block of the MPEG-1 application [18] and the *nlfilt* and *tile* functions (with the scale function inlined) [Note 1] from the GIMP image processing tool [19]. The run time of our system for these designs is less than 5 user seconds on a 1.6 GHz PC running Linux.

Table 1 lists the characteristics of the various designs used in terms of the number of if-then-else conditional blocks (If-HTGs), loops (Loop-HTGs), basic blocks and the total number of operations in the input description. The number of If-HTGs, Loop-HTGs and basic blocks is indicative of the control complexity of the design. The Table also gives the resource allocation used for scheduling the designs in the experiments presented below: $+-$ does add and subtract, $==$ is a comparator, $*$ a multiplier, $/$ a divider, [16] an array address decoder and $\ll$ is a shifter. All resources are single cycle except the multiplier (two cycles) and the divider (five cycles).

For all the experiments presented below, we used a priority-based list scheduler that employs speculative code motions [16]. The scheduling results for these four functions are presented in Tables 2 and 3 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, the longest path is the length of the loop body multiplied by loop iterations.

The first rows in Tables 2 and 3 list the results for when all the code motions from Fig. 2 are enabled; *except* conditional speculation. We call this the *baseline case*. The second row has conditional speculation (CS) enabled along with the rest of the code motions. In the third row, all the code motions including CS are enabled along with the BBDDT. The fourth row lists the results for when all the code motions are enabled along with the BBDCM. The fifth row has both the branch balancing algorithms enabled along

Note 1: These floating point functions have been arbitrarily converted to integer functions here. This does not affect the nature of the data and control flow, but only the data values that are processed.

with all the code motions. The percentage reductions of each row over the baseline case are given in parentheses.

The results in Tables 2 and 3 demonstrate that the branch balancing algorithms presented in this paper have to be employed to make conditional speculation truly effective. When conditional speculation is enabled alone there are modest (0–5%) improvements in number of states and cycles on the longest path (second row in both Tables). However, when both the BBDDT and BBDCM algorithms are employed to increase the opportunities for conditional speculation, the improvements range from 10 to 37% in the number of states and 10 to 38% in the cycles on the longest path (last row in both Tables).

The results in the two Tables also demonstrate that the two branch balancing algorithms are complementary to some extent. Whereas the BBDDT technique is more effective for the MPEG-1 *pred*2 design, the BBDCM technique is more effective for the GIMP *tiler* design. Also, we can see from the results in the two Tables that the best result for the GIMP *nlfilt* design is obtained when both the BBDDT and BBDCM algorithms are employed. These results demonstrate that each of the branch algorithms create different and unique opportunities for employing conditional speculation.

10 Conclusions

This paper presented two branch balancing techniques that dynamically insert scheduling steps in the shorter branch of a conditional block. This creates new opportunities for speculative code motions without increasing cycles on the longest path through the design. The branch balancing techniques are critical to code motions such as conditional speculation that duplicate operations into multiple basic blocks. Also, if profiling information is available, these techniques can be easily modified to add scheduling steps only in the conditional branches that are less likely to be taken. Results for four real-life multimedia and image-processing designs show improvements of up to 38% in cycles on the longest path and 37% in controller size when the dynamic conditional branch balancing techniques are enabled.

11 Acknowledgment

This work was supported by the Semiconductor Research Corporation: Task ID 781.001.

12 References

1 Chaiyakul, V., Gajski, D.D., and Ramachandran, L.: 'High-level transformations for minimizing syntactic variances'. Presented at Design Automation Conf., Dallas, TX, 14–18 June 1993
2 Gupta, S., Savoiu, N., Kim, S., Dutt, N.D., Gupta, R.K., and Nicolau, A.: 'Speculation techniques for high level synthesis of control intensive designs'. Presented at Design Automation Conf., Las Vegas, NV, 18–22 June 2001
3 Gupta, S., Savoiu, N., Dutt, N.D., Gupta, R.K., and Nicolau, A.: 'Conditional speculation and its effects on performance and area for high-level synthesis'. Presented at Int. Symp. on System Synthesis, Montreal, Canada, 30 September–3 October 2001
4 Gupta, S., Dutt, N.D., Gupta, R.K., and Nicolau, A.: 'Dynamic Conditional branch balancing during the high-level synthesis of control-intensive designs'. Presented at Design, Automation and Test Conference, Munich, Germany, 3–7 March 2003
5 Gajski, D.D., Dutt, N.D., Wu, A.C.-H., and Lin, S.Y.-L.: 'High level synthesis: introduction to chip and system design' (Kluwer Academic, Boston, MA, 1992)
6 Wakabayashi, K., and Tanaka, H.: 'Global scheduling independent of control dependencies based on condition vectors'. Presented at Design Automation Conf., Anaheim, CA, 8–12 June 1992
7 Radivojevic, I., and Brewer, F.: 'A new symbolic technique for control-dependent scheduling', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 1996, **15**, (1), pp. 45–57.
8 Lakshminarayana, G., Raghunathan, A., and Jha, N.K.: 'Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions'. Presented at Design Automation Conf., San Francisco, CA, 15–19 June 1998
9 Rim, M., Fann, Y., and Jain, R.: 'Global scheduling with code-motions for high-level synthesis applications', *IEEE Trans. Very Large Scale Integr. (VLSI) Systems*, 1995, **3** (3), pp. 379–392
10 dos Santos, L.C.V., and Jess, J.A.G.: 'A reordering technique for efficient code motion'. Presented at Design Automation Conf., New Orleans, LA, 21–25 June 1999
11 Kolling, P., Al-Hashimi, B., and Abott, K.M.: 'Efficient scheduling of behavioural descriptions in high-level synthesis'. *IEEE Proceedings-Computers and Digital Techniques*, 1995, **144**, (2), pp. 75–82
12 Fisher, J.: 'Trace scheduling: a technique for global microcode compaction', *IEEE Trans. on Comput.*, 1981, **30**, pp. 478–490
13 Muchnick, S.S.: 'Advanced compiler design and implementation' (Morgan Kaufmann, San Francisco, CA, 1997)
14 dos Santos, L.C.V.: 'A method to control compensation code during global scheduling'. Presented at Workshop on Circuits, Systems and Signal Processing, Mierlo, The Netherlands, 27–28 November 1997
15 Girkar, M., and Polychronopoulos, C.D.: 'Automatic extraction of functional parallelism from ordinary programs', *IEEE Trans. Parallel Distrib. Syst.*, 1992, **3**, (2), pp. 166–178
16 Gupta, S., Dutt, N.D., Gupta, R.K., and Nicolau, A.: 'SPARK: a high-level synthesis framework for applying parallelizing compiler transformations'. Presented at Int. Conf. on VLSI Design, New Delhi, India, 4–8 January 2003
17 Gupta, S., Reshadi, M., Savoiu, N., Dutt, N.D., Gupta, R.K., and Nicolau, A.: 'Dynamic common sub-expression elimination during scheduling in high-level synthesis'. Presented at Int. Symp. on System Synthesis, Kyoto, Japan, 2–4 October 2002
18 Spark Synthesis Benchmarks FTP site. ftp://ftp.ics.uci.edu/pub/spark/benchmarks, accessed 10 January 2002
19 GNU Image Manipulation Program. http://www.gimp.org, accessed 10 January 2002
20 Gupta, S.: 'Coordinated Coarse-Grain and Fine-Grain Optimizations for High-level Synthesis'. PhD thesis, University of California, Irvine, 2003