Improving Circuit Performance with Multispeculative Additive Trees in High-Level Synthesis

Alberto A. Del Barrio, Román Hermida, Seda Ogrenci Memik, José M. Mendías, María C. Molina

Abstract— **The recent introduction of Variable Latency Functional Units (VLFUs) has broadened the design space of High-Level Synthesis (HLS). Nevertheless their use is restricted to only few operators in the datapaths because the number of cases to control grows exponentially. In this work an instance of VLFUs is described, and based on its structure, the average latency of tree structures is improved. Multispeculative Functional Units (MSFUs) are arithmetic Functional Units that operate using several predictors for the carry signal. In spite of utilizing more than a predictor, none or only one additional very short cycle is enough for producing the correct result in the majority of the cases. In this paper our proposal takes advantage of multispeculation in order to increase the performance of tree structures with a negligible area penalty. By judiciously introducing these structures into computation trees, it will only be necessary to predict the carry signals in certain selected nodes, thus minimizing the total number of predictions and the number of operations that can potentially mispredict. Hence, the average latency will be diminished and thus performance will be increased. Our experiments show that it is possible to improve 26% execution time. Furthermore, our flow outperforms previous approaches with Speculative FUs.**

Keywords: Variable-Latency Functional Units, speculation, additive operation trees, High-Level Synthesis.

I. INTRODUCTION

There is a multitude of DSP and multimedia applications composed of one or several additive structures. In particular, structures like addition chains or trees are commonly found in signal processing applications such as ECG [1], numerical integration methods [2], or the ADPCM [3]. Similar structures are also frequently found in mobile devices, which deploy a large quantity of signal processing and multimedia functionality. Therefore, it is crucial to improve the quality of adders and addition-dominated computations without incurring significant area or power overhead. This is especially relevant in the field of design automation, because synthesis algorithms should include mechanisms for the detection and efficient implementation of such additive structures, which will require not only a redesign of the Functional Units (FUs), but a thorough review of the synthesis algorithms, too.

The straightforward approach to tackle the abovementioned problem consists of increasing the adders' complexity and thereby, their speed. This will improve performance, while increasing area and power considerably. Various adder designs have been proposed to achieve different trade-offs between hardware complexity and performance [4][15]. All of them exhibit some fixed latency, the main goal being the minimization of the delay along the carry chain. A popular design option is the Carry Save Adder (CSA) [4], where the carry propagation is accelerated at the expense of increasing the number of Full-Adder cells. While recent contributions to Dataflow Graph (DFG) transformation have optimized the use of CSAs [8-10, 23-24], they also involve some drawbacks. For instance, in order to maximally reuse a CSA-tree, the cluster of operations that it implements must appear several times in different steps of the executed algorithm. Furthermore, the use of CSAs requires extra routing and storage resources, which will have a negative impact on the area and delay of the circuit. Finally, every CSA structure requires a last Carry Propagation Stage (CPA), thus limiting performance.

On the other hand, the introduction of Variable Latency Functional Units (VLFUs) has broadened the design space with new implementation alternatives [11-14, 25-28]. These FUs are characterized by an aggressive reduction of the clock period at the expense of requiring a variable number of clock cycles to complete an operation, depending on the input data. So, the latency of a VLFU may range from one very short cycle (the so called short latency mode) to a maximum number of cycles that depends on its internal design (the so called long latency mode). Obviously, the performance of the datapath will be higher if the VLFUs work in the short latency mode most of the time. Although the literature offers several strategies for handling these units [14, 16-18, 22], all of them are subject to the possibility of the worst-case timing, i.e. VLFUs working in long latency mode.

VLFUs consume less area than their fixed-latency counterparts, although some area penalty is caused by the increased complexity of the required controllers, which is independent of the data width. Therefore the area improvements derived from the use of VLFUs will be higher for wider datapaths. On the contrary, fixed-latency FUs, and especially CSA structures, will demand a considerably larger area if the bitwidth is increased, due to the additional multiplexers and registers that will be required, as well as, the complexity of the CSA itself.

In our previous work, we have proposed several designs for VLFUs. These units are based on various forms of speculation over a single carry value [12, 18] or multiple carries at the same time [21, 29]. We shall refer to them as Speculative FUs (SFUs) and Multispeculative FUs (MSFUs), respectively, although we will use MSFUs as the generic term. Such MSFUs work in low latency mode when they correctly predict all the speculated carries. This situation will be referred as a prediction hit, and the opposite as a misprediction. In this work we propose the use of MSFUs in order to optimize the execution of additive structures by leveraging the characteristics of such structures, which will allow us to reduce the number of mispredictions and thus increase overall performance. According to our experiments it is possible to reduce execution time by 24% on average (44% best case) with respect to a baseline implementation with fixed-latency logarithmic FUs.

The rest of the paper is organized as follows: section II discusses the related work and section III introduces the multispeculation concept in datapaths. Section IV generalizes multispeculation over any additive structure. Finally, sections V and VI present our experimental results and conclusions.

II. Related Work

The introduction of VLFUs has attracted a great deal of interest from the field of Design Automation. These FUs offer higher performance at a lower cost, in comparison to the fastest fixed-latency FUs. Nevertheless, as VLFUs possess a dynamic nature, they require rather unconventional controllers [5-7]. Indeed, using these designs requires the inclusion of some additional logic to detect when the FU can safely work in the short latency mode and to provide as many additional cycles as needed, otherwise. Several authors deal with VLFUs as conditional branches within a fully centralized controller [14, 16-17, 22], which has led to the use of very few VLFUs to keep the complexity of the controller manageable, thus limiting the achievable performance. On the other hand, when many VLFUs are working at the same time, the probability of a misprediction in at least one of them quickly raises, which under a centralized controller will penalize the overall performance. Hence, the development of efficient controllers is as critical as the design of fast VLFUs [18].

Although in this paper a centralized controller, similar to those proposed in [14, 16-18], will be adopted in order to take advantage of its simple structure, we will overcome its performance limitations by reducing the number of operations that can cause mispredictions thanks to the management of the multispeculative additive structures that is proposed.

From the point of view of the scheduling and binding of control/data flow graphs, all of the previously mentioned approaches leverage VLFUs for reducing the average latency of the circuit (while keeping cycle times similar to implementations with conventional FUs), in exchange for a slight increase in area. Nevertheless, none of the aforementioned techniques takes into account that in some cases it is not necessary to predict any carry information. In fact, it is quite common to find additive structures where it is possible to pipeline word-level carries through all the inner stages [19] and speculate only in the last one.

In the work presented in [21], the authors defined the generic structure of an n-bit Multispeculative Adder (MSADD), which is composed of n/k k-bit fragments interconnected with predictors, and they proved that a correct result is produced after two short clock cycles at the most for the vast majority of additions. This happens regardless the number of predictors, because if k is big enough, the probability of propagating a misprediction from a fragment to the following is nearly zero. Formal and empirical studies about this probability are presented in [21], providing 90% to 99.99% correct additions after two cycles, depending on the predictor type and the fragment size. Furthermore, it should be observed that in an MSADD composed of n/k fragments the cycle time is dominated by the delay of one k-bit fragment, and thus MSADDs can work with a much faster clock period than their monolithic counterparts. So, even in the event that an extra clock cycle were required to correct mispredictions, the overall computation time will still be considerably shorter in a MSADD than in a monolithic adder.

Besides demonstrating the efficiency of MSADDs as individual entities, these units are also introduced into datapaths for the implementation of addition chains. The basic idea consists of pipelining the carry vector during the first stages of the chain, and applying the so called Static Zero Prediction in the last stage. This means that there will be a misprediction in the last stage iff any of the intermediate carries is a '1'. If this happens, the last stage will be repeated as many times as necessary to produce a correct result, i.e. up to n/k-1 times. Thereby, a conventional addition chain of L n-bit operands will require L-1 cycles, with monocycle FUs, provided that a single adder is reused to implement all the operations. On the other hand, in the multispeculative case, the addition of L operands will need L-1+1=L cycles at the most with a high probability, because a MSADD requires two cycles at the most for producing a correct result in the vast majority of the cases. These ideas constitute the basic principle of multispeculation, and they will be expanded for more complex structures in sections III and IV of this paper. Finally it should be also noted that multispeculative Multipliers (MSMULs) rely upon the same principle, as they are composed of a Partial Product Matrix, under the form of a CSA, and a last MSADD stage.

III. MULTISPECULATIVE ADDITIVE STRUCTURES

MSADDs possess some special properties that can be exploited when applied to implement datapaths with additive structures. MSADDs are divided into several fragments, where each of them receives a carry-in proceeding from a predictor. Independently of the implementation, predictors contain a sequential element to store the prediction, e.g. a D-flipflop.

Our proposal is based on taking advantage of the flexibility offered by the carry-store elements, which are contained in the predictors of MSADDs. These elements will be used to save the intermediate carries and pipeline them from a cstep to the following [21] during the first cycles of execution, avoiding thus the additional penalty due to mispredictions. Finally in the last

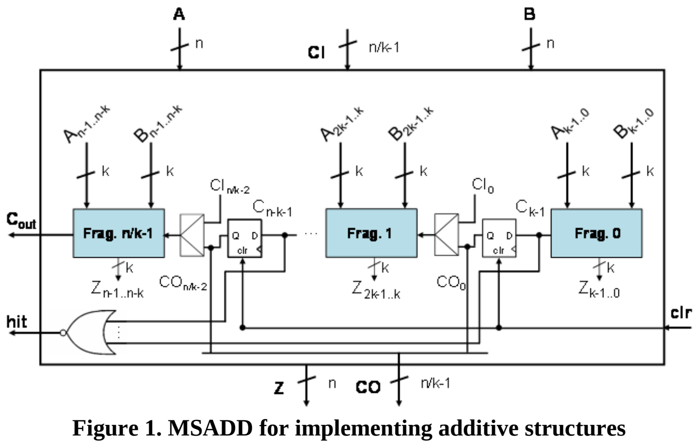


Figure 1. MSADD for implementing additive structures

cycles, predictors will be utilized to speculate carries, reducing the critical path of the last stage, which is the main limitation of CSA structures. In this last stage a Static Zero Prediction will be assumed. Hence, in the last cycles a misprediction will be produced iff a carry-out from any fragment is '1', i.e. different from the prediction.

Figure 1 shows the MSADD that implements the aforementioned ideas. As it can be observed, the MSADD receives two *n*-bit operands *A* and *B*, as well as a Carry In vector (*CI*), and produces an *n*-bit result *Z* and a Carry Output vector (*CO*). These carry vectors are *n*/*k*-1 bit width, i.e. one carry bit per fragment, except the most significant one, which does not require to be accumulated. The *CI* will be a *CO* produced by an addition executed in the same or any other MSADD. In must be noted that if the *CI* of a MSADD is always its own *CO*, the multiplexers that appear on the left of the D-flipflops will be removed. More details about the use of the *CI* input will be given in the following subsections. The D-flipflops implement the dual functionality of either pipelining the carry or Static Zero Prediction. They will be cleared every time the additive structure finishes its execution correctly. In addition to this, the destination register utilized in the last cstep must be used as the source when correction is required. Only a slight modification of the controller is necessary to indicate that predictors, i.e. D-flipflops, must always be written and the carries pipelined during the first csteps, i.e. in *pipeline mode*, and the *hit* signal considered from the last cycle onwards, i.e. in *prediction mode*. Then, in this last cstep a control mechanism similar to [16,17,21] is applied, i.e. if the *hit* signal is false there is a transition to a *correction* state, and otherwise there is a transition to the following state. In the next subsections, a detailed explanation of how these principles apply to concrete structures different to the chains will be presented.

A. Scheduling of Full Binary Addition Trees

The case of full binary addition trees follows the same principle as the chains: pipeline the inner nodes carries and Static Zero Prediction in the root node. However, there are some additional problems to be solved that arise due to the inherent greater complexity of the structure.

With multispeculation, each adder *Ai* is producing a vector of provisional results (*Vi*), and a vector of intermediate carries (*Ci*), such that the final result $S_{Ai}=Vi+Ci$. The problem in a full binary tree structure is that every adder produces two vectors, and can only consume three. Hence, two adders will produce four vectors, and in the following level of the tree only three of these vectors will be consumed. In order to solve this problem we propose to consume these additional vectors through the idle adders by exploiting the associative property of addition. The key idea is as follows: in any full binary adder tree there is always a level above which every level contains more additions than available adders, and below which every level contains more available adders than additions. Hence, it is guaranteed that there will be idle adders available.

In a full binary addition tree, the number of required csteps depends on both the number of operands and the number of available adders, as stated in the following proposition.

Proposition 1. Provided that the number of operands to add is L, and the number of available two inputs monocycle adders is s, with $L \geq s$, the minimum number of required csteps is given by equation *(3)*

$$\lfloor L/s \rfloor + \lceil \log(s + L \bmod s) \rceil - 1 \tag{3}$$

Proof. In order to guarantee the minimum number of csteps, an ASAP schedule will be considered. Thus, given L operands and s adders, in the first level of the tree $2s$ operands will be scheduled and will produce s results. In the next levels, s more operands will be scheduled as well as the s results coming from the previous stage. This must be repeated until the number of unscheduled operations r is lower than s. It will require $\lfloor L/s \rfloor$ -1 csteps.

In the following stage, there will be less than $2s$ operands. The reason is that at this point of the tree there will be $r < s$ unscheduled input operands, so considering the s results coming from the previous stage, in this level there will be $s+r < 2s$ operands. Hence, this subtree will take $\lceil log(s+r) \rceil$ stages, where $r = L \bmod s$. Therefore, the whole tree will need $\lfloor L/s \rfloor + \lceil log(s + L \bmod s) \rceil$ - 1 csteps. □

Therefore, it can be concluded that if a non-speculative full binary tree of L operands requires $\lfloor L/s \rfloor + \lceil log(s + L \bmod s) \rceil$ - 1 cycles, with monocycle FUs, its multispeculative version will take $\lfloor L/s \rfloor + \lceil log(s + L \bmod s) \rceil$ cycles at the most with a high probability. It should be noticed that this additional cycle corresponds to the *recovery cstep* that is required just in case of misprediction, but that is not necessary if there is a hit. In conclusion, in both chains and trees, the latency is increased by only one cycle. In fact, the chain case is an instance of the full binary tree where s=1. Hence, as the expected latency is similar, the expected overall gain in execution time will originate from the reduction in the adder delay.

1) An Illustrative Example

Figure 2 is an example of scheduling, binding and carry propagation of a full binary addition tree with 8 operands. The scheduling is determined by the dotted lines and the binding by the node colors and patterns. Note that the registers are not included for the clarity of the example. Let *Ai* be the i^{th} adder, and let *Ci* be the associated carry-out vector. It must be noticed that these carry vectors are only shown in the speculative cases. In these cases, the *CI* inputs and *CO* outputs are those adder inputs and outputs driven by a green diagonal arrow. In figures 2 a) 2 b) and 2 c), 4, 3 and 2 non-speculative adders have been considered, while in figures 2 d), 2 e) and 2 f) these non-speculative adders have been substituted by MSADDs.

In figures 2 a), 2 b) and 2 c) the datapaths require a fixed amount of cycles, that according to *Proposition 1* are 3, 4 and 4, respectively. On the other hand, the multispeculative trees, corresponding to figures 2 d), 2 e) and 2 f), need only one additional recovery cstep, that in execution time will be translated into an additional cycle at the most with a high probability. Note that these recovery csteps are labeled as csteps C'. In spite of this slight latency increase, we must note that the multispeculative implementations cycle time will be noticeably shorter than in the non-speculative ones.

Let us consider the case of figure 2 d). As it can be observed, in cstep 1 *C1*, *C2*, *C3* and *C4* are produced, but in cstep 2 only *A1* and *A3* are active, so only *C1* and *C3* can be consumed. In order to solve the problem of consuming *C2* and *C4* vectors, the idle adder *A2* will add both *C2*, already stored in the *A2* D-flipflops, and *C4*. This new addition will be called *carry-propagation addition*. It should be pointed out that the carry-out vector of *A2* in cstep 2 will be full of zeros because *k*>1. Hence, the result vector coming out from *A2* will be enough for accumulating three input vectors. In the worst case, these vectors will only contain a '1' in the least significant position of each MSADD submodule. In cstep 3 the algorithm will proceed in the same manner as with the addition chains, and will force *A1* to work in predictive mode, i.e. considering a Static Zero Prediction. Thus, *C1* will be consumed in its own *A1* adder, while *C3* will be added in *A2*, with a new carry-propagation addition, to the previous result proceeding from cstep 2. The result *Z'* will be equal to *Z* iff there is a hit in *A1*, and if the result coming out from *A2* is equal to zero. Otherwise, a last cstep 3' will be necessary for adding *V1* and *C1* plus *V2*. In cstep 3', *A1* will work in prediction mode until it produces a hit. This will happen with a high probability, since both *C1* and the result coming from *A2* in cstep 3 will usually contain only a few '1's in the least significant positions of every MSADD submodule. Thus, 4 cycles will be sufficient with a high probability.

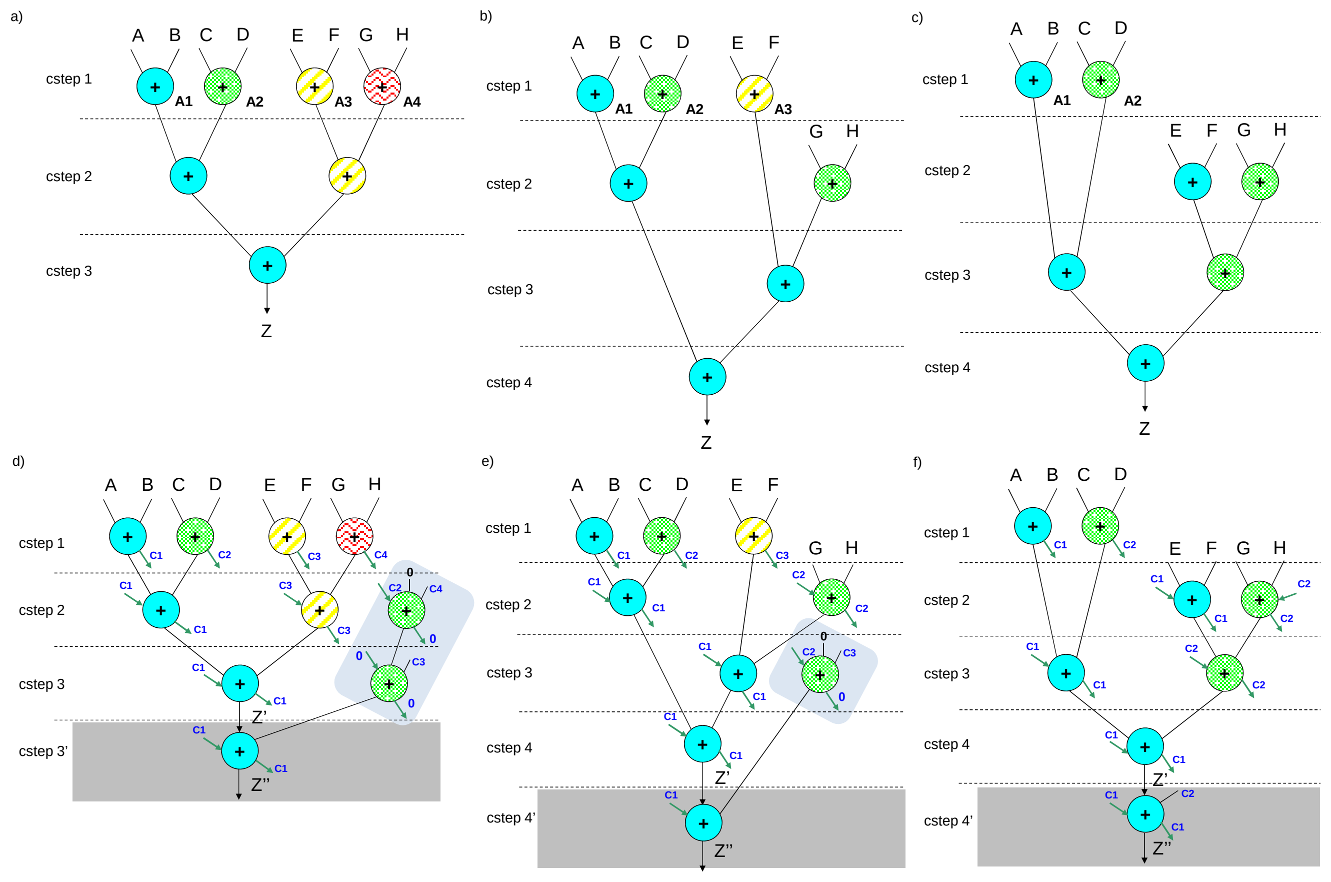


Figure 2. Scheduling, binding and carries propagation of the Z=(A+B+C+D+E+F+G+H) addition tree using a) 4, b) 3, c) 2 non-speculative adders, and using d) 4, e) 3, f) 2 Multispeculative Adders

In figure 2 e), after the execution of cstep 1, three carry vectors are produced, i.e. *C1*, *C2* and *C3*, but in cstep 2 only two of those three can be consumed. Nevertheless, as *C3* cannot be added with any other vector in cstep 2, the scheduling algorithm waits until a cstep where this carry vector can be accumulated with another operand. In cstep 2 two carry vectors are produced, i.e. *C1* and *C2*, but in cstep 3 only *C1* can be consumed. Hence, in cstep 3, the idle adder *A2* is utilized for adding both *C2* and *C3*, producing a zero carry-out vector for the same reasons as in figure 2 d). Finally, in cstep 4, *Z'* will be equal to *Z* if there is a hit in *A1*, i.e. *C1*='0', and if the result coming from *A2* is equal to zero. Otherwise the last cstep 4' will be repeated until *A1* produces a hit, thus guaranteeing *Z*=*Z''*.

In figure 2 f), there is no problem accumulating the carry vectors. Due to the reduced number of adders, the carry vectors are always accumulated in their own adders, i.e. *A1* or *A2*. In cstep 4, *Z'* will be equal to Z iff *A1* produces a hit, and if the carry-out vector of *A2* is full of zeros, i.e. *A2* produces a hit in cstep 3. Otherwise, cstep 4' will be repeated until *A1* produces a hit.

B. Additive Binary Trees

Multispeculative full binary trees can be extended to any other additive tree structure. An additive tree is one that is composed only of addition nodes, except for the leaf nodes, that can also be products. For example, let us consider the *dot* product, which possesses the same structure as the ones shown in figure 2 but substituting the first four additions by products. In this case, our proposal relies upon the fact that multipliers are usually implemented with a partial product matrix in CSA form and a last CPA stage. If this last CPA stage is a MSADD, the result produced by the multiplier *Mi* can be expressed as the sum of two bit vectors *Vi* and *Ci* such that $S_{Mi}=Vi+Ci$, similar to the case of MSADDs. Hence, in the multispeculative *dot* product, vectors can be consumed in the same fashion as shown in figures 2 d), 2 e) or 2 f). In order to achieve this, it is only necessary to properly select the *CO* output of the MSFU that will feed the *CI* input of the MSADD under consideration.

Therefore, it can be concluded that addition chains and full binary addition trees are actually instances of additive binary trees, so in the following we shall refer to additive binary trees as the general term for all these structures.

IV. ON THE APPLICATION OF MULTISPECULATION TO HIGH-LEVEL SYNTHESIS

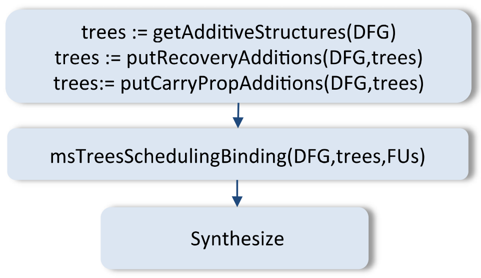
In this section, the principles and methodology to incorporate multispeculation to HLS will be generalized and explained in detail. Although all the aforementioned techniques can be applied directly to certain DFGs that are exactly in the form of additive binary trees, there are cases where the DFG contains several additive trees as subcomponents. In such cases, the use of multispeculation will be restricted only to those additive trees. For this purpose, it is necessary to design an algorithm to identify those additive structures where multispeculation can be applied, to develop an automated procedure to keep the datapath in a correct state, and finally to define an interface for communicating with the controller. In order to implement this interface, a method similar to the ones proposed for the control of Speculative FUs with single predictor units is adopted. This solution entails a *hit* signal collected from each unit. The implementation of the controller will be similar to the ones presented in [16,17,21]. If there is a failure there will be a transition to a *correction state*, and otherwise a transition to the following state. Besides, some special cases will be considered in order to skip the correction state or not. These cases will be explained in section IV.B.

In order to apply multispeculation to DFGs, the HLS flow must be adapted to face the new restrictions imposed by the use of MSADDs inside additive structures. The fact that must be taken into account is that the availability of the operators is somehow restricted once they are utilized inside an additive tree. The reason is that a given adder will not be available for a different additive structure until its carry vector has been consumed. However, as it will be shown in the experiments section, with a reasonable number of resources, latency will not be penalized.

Figure 3 a) illustrates our overall design flow. First all the additive binary trees are identified. Second, a *recovery addition* per tree is included in the DFG. And third, the *carry-propagation additions* are introduced where required. The introduction of these recovery and carry-propagation additions constitutes the mechanism for keeping the datapath in a correct state, as in the full binary addition trees case. Afterwards, the scheduling and binding are performed and the datapaths and controllers synthesized.

The algorithm for detecting the additive binary trees has to determine those subsets of nodes where all internal nodes are strictly additions and no multiplications are encountered. It processes the DFG in a depth-first fashion, starting at the outputs, and stopping in each branch once a product node is found. The procedure is repeated until every node has an additive tree to belong to. The introduction of carry-propagation addition is performed afterwards. Every tree is processed until no more carry-propagation additions are needed. Regarding the scheduling and binding, due to the aforementioned operators' restriction, a combined resource constrained scheduling and binding algorithm [6,7] has been applied. The pseudocode corresponding to this algorithm is shown in figure 3 b). In order to schedule operations, first they are ordered according to the inverse of their mobility, and then they are scheduled and bound iff there is a suitable free FU. The main difference with regard to a conventional flow comes when

a)


trees := getAdditiveStructures(DFG)
trees := putRecoveryAdditions(DFG,trees)
trees:= putCarryPropAdditions(DFG,trees)
msTreesSchedulingBinding(DFG,trees,FUs)
Synthesize

b)

```
msTreesSchedulingBinding(DFG,trees,FUs)
begin
 asaps := calculateASAP(DFG);
 alaps := calculateALAP(DFG);
 latency := 1;
 while (unscheduledOperations(DFG)) do
   readyOps := getReadyOperations(DFG);
   ordReadyOps := ordByPrior(readyOps,asaps,alaps);
   foreach operation in ordReadyOps do
     fu := enoughResources(operation,FUs)
     if (fu!=null) && carryConsumed(fu) then
       scheduleOperation(readyOp);
     end if;
   end for;
   update(asaps,alaps);
   latency := latency + 1;
 end while;
end;
```

Figure 3. a) Design flow for multispeculative additive structures, b) scheduling and binding pseudocode for Multispeculative Trees

the operation to be scheduled in a given cstep and bound to a given FU, and the previous operation bound to this FU lie in different trees. In such case, it must be determined whether the carry-out vector associated to the FU has been consumed or not. If this carry-out vector has not been consumed yet, then the FU is not ready to be bound to this new operation, because the carry-out vector must be added in the same tree where it was generated. Otherwise, if the operation belongs to the same tree as the previous operation bound to the FU, the carry-out vector can be accumulated in this FU. Thereby, in this case the function *carryConsumed* returns true.

A. An Illustrative Example

In order to show the principles of multispeculation over complete datapaths, our techniques will be applied to the Discrete Wavelet Transform (DWT) [5] benchmark. The set of operators is composed of 16-bit logarithmic adders and Wallace Multipliers with a 32-bit logarithmic adder in the CPA stage. In both cases, *k*=4 and Kogge-Stone (KS) basic blocks are considered. Thereby, each 16-bit MS-Kogge-Stone adder (MSKS) [21] contains 3 predictors, while each 32-bit MSKS, which are embedded into the corresponding multiplier, contains 7 predictors. In this example the following assumptions have been made:

1) A non-speculative adder will take 2 cycles, and a non-speculative multiplier will take 4 cycles.

2) A speculative adder will take 1 cycle, and a non-speculative multiplier will take 3 cycles. Each additional carry correction will require a cycle.

Figure 4 a) depicts the DWT DFG. Operations are labelled with numbers 1 through 17. Note that every operation needs two operands, but for the sake of clarity, primary input' arrows are not shown. Figure 4 b) corresponds to a conventional resource constrained scheduling [6,7] using 2 non-speculative multipliers and 1 adder. As it can be observed, it takes 28 csteps.

Figures 4 c), 4 d) and 4 e), illustrate the application of our methodology. First, all the additive binary trees are identified. Figure 4 c) depicts the 6 trees that comprise the DWT DFG. As it can be observed, products can only be leaf nodes, since multiplications in the interior nodes are not permitted as explained earlier. Second, a recovery addition is introduced at the end of every tree. This is shown in figure 4 d), where the new recovery additions (***Operations 4'***, ***7'***, ***8'***, ***11'***, ***12'*** and ***17'***) are highlighted in darker shade colour. Recovery additions will be later executed as many times as needed for producing the correct result of the tree. Third, the carry-propagation additions are introduced where necessary. In this case, they are not required however. Next, the new DFG is scheduled and bound considering that all the operations will produce a hit in the prediction. Our resources set is composed of 2 multipliers and 1 adder, as in the non-speculative case. The resulting schedule and FU-binding is shown in figure 4 e). Note that the two multipliers bindings are depicted with the use different filling colors/patterns for the nodes: solid shaded nodes map to M1 and striped nodes map to M2. Addition nodes are bound to the only adder A1.

In order to understand the algorithm several cases related to figure 4 e) will be explained in detail. Dotted arrows denote how the carry vectors are propagated. Note that recovery additions do not have an output dotted arrow because they produce a correct result. Furthermore, the csteps are numbered with a label C or C' in the leftmost part of figure 4 e). The C' csteps are recovery csteps that only contain recovery additions. Hence, these C' csteps can be skipped in execution time if the controller detects a hit.

For instance let's consider the tree composed of ***Operations 1***, ***2***, ***3***, ***4*** and ***4'***. As there are enough resources, M1 and M2 are utilized for binding both ***Operations 1*** and ***3***. Now, let's consider the tree composed of ***Operations 6***, ***8*** and ***8'***. As M1 is free in cstep 4, and its carry-out vector is consumed by ***Operation 2*** in cstep 4, it is possible to bind ***Operation 6*** to multiplier M1. On the contrary, it must be observed that the M2 multiplier remains locked until ***Operation 4'*** is bound. That is why ***Operation 10*** cannot be scheduled nor bound until cstep 7. Following this procedure with the rest of the operations, the scheduling and binding depicted in figure 4 e) is produced. As it can be observed in this figure, if ***Operations 7***, ***11*** and ***17*** produce a hit, csteps 10', 14' and 19' will actually not be necessary. On the other hand, besides the C' csteps, every cstep containing a recovery addition will become a recovery cstep, i.e. a *repeatable cstep*. These csteps are highlighted with a solid background. Thus, if any of the ***Operations 4, 7, 8, 11, 12,*** or ***17*** produce a misprediction, the corresponding recovery cstep will be repeated until producing a hit. As in the addition chains and trees, the worst case would require *n*/*k*-1 additional cycles for certifying a correct result. Hence, the

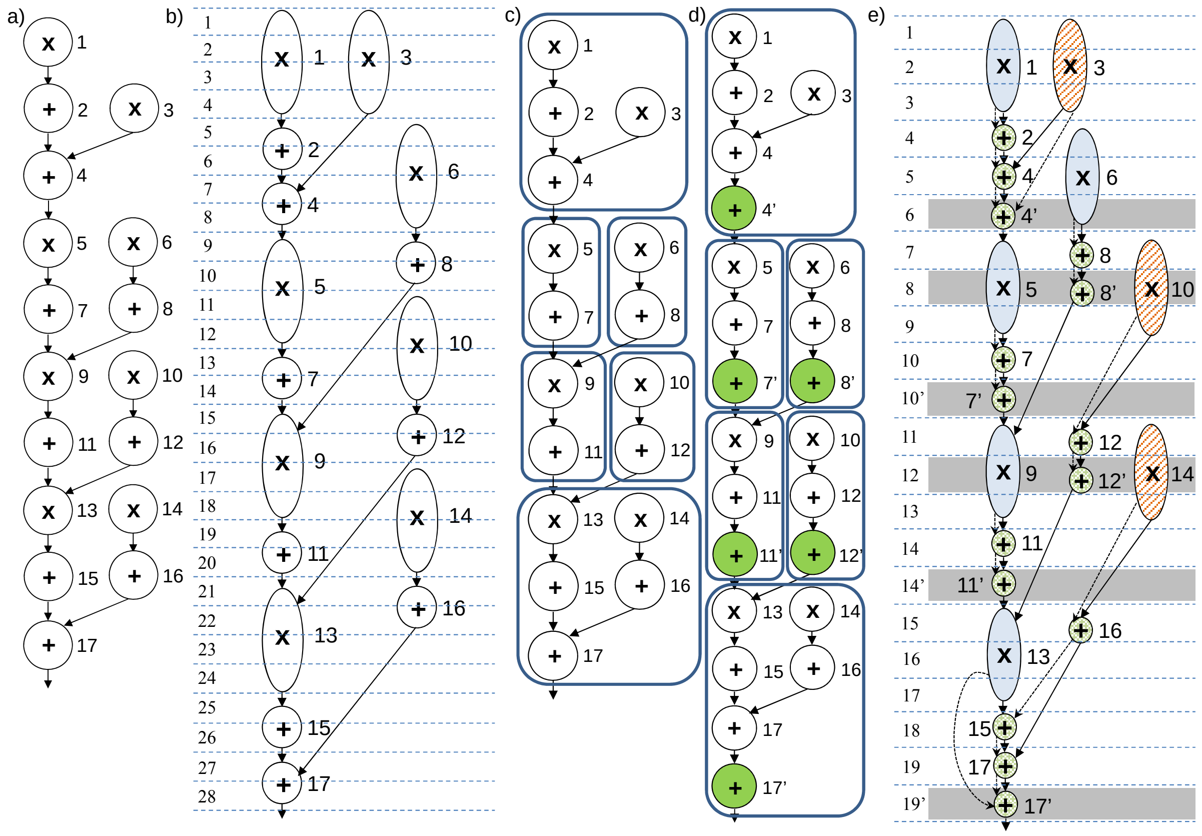


Figure 4. a) Discrete Wavelet Transform (DWT) DFG [5], b) conventional scheduling with non-speculative FUs, c) additive structures in the DWT DFG without and d) with the recovery additions, e) scheduling and FU-binding with multicycle MSFUs

scheduling shown in figure 4 e) will take 19 cycles in the best case, and 19+(16/4-1)*6 = 37 cycles in the worst case. Nevertheless, as stated in section III, and according to our experiments, none or only one recovery cycle will usually be enough for certifying the correct result, so the average latency will range between 19 and 25 cycles, which is better than the conventional case. Furthermore, it must be taken into account that ***Operations 4'***, ***8'*** and ***12'*** are always executed, because they are scheduled in csteps that also execute non-recovery operations. In other words, a 1-cstep *recovery slot* is available for these operations *for free*. Thus, provided that a cycle is enough for correcting the results, ***Operations 4'***, ***8'*** and ***12'*** will serve to correct possible mispredictions coming from ***Operations 4***, ***8*** and ***12***, respectively. Therefore, the actual average latency would range between 19 and 22 cycles.

In conclusion, the application of multispeculation over additive binary trees has reduced the instances of actual mispredictions. If the existence of additive trees had been ignored, every operation may have produced a misprediction, while in our case only 6 operations can mispredict, and furthermore even 3 of them can hide one recovery cycle, which will diminish average latency even further.

B. A Comparison with CSA

Figure 5 shows the scheduling corresponding to the DWT benchmark, but considering CSAs. Figure 5 a) depicts a first approach where all the addition trees are replaced by CSA Wallace trees [4]. This shall be named Moderate CSA (*Mod-CSA*) implementation. The CSA trees are those nodes highlighted in solid red. These nodes produce two *n*-bit output vectors, so a CPA addition is required to calculate the result. Let us assume that every CSA tree takes 1 cycle long. As it can be observed, the use of CSA trees is restricted by the products, because, as in the case of multispeculative trees, the addition' results must be correct before the beginning of each multiplication. There is a latency decrease with respect to the conventional approach (26 vs 28 cycles), but two multipliers and one adder are still necessary, plus the 4:2 CSA tree and additional routing and storing resources.

Figure 5 b) depicts a schedule where the multipliers have no last CPA stage, i.e. multiplicative nodes only consist of the Partial Product Matrix accumulation (PPM). Thus, they produce the result in CSA format, so in general more registers will be necessary. This shall be named Extreme CSA (*Extr-CSA*) implementation. Let us consider that they take 2 csteps. Latency is further reduced than in the previous case (20 cycles), but besides the 2 multipliers and the adder, a 7:2 CSA tree is required. Moreover, as several nodes reuse the CSA tree and the registers are shared among more variables, the number of necessary multiplexers will be increased considerably.

Table I contains a summary with the number of resources. The first column depicts the implementation type. Columns 2, 3 and 4 are the number of 16-bit adders, 16x16 multipliers and equivalent 16-bit 3:2 CSAs. Columns 5, 6 and 7 show the number of equivalent 16-bit 2:1 multiplexers, 16-bit registers and 1-bit predictors. The rightmost column is the latency of the circuit. As it is

Table I. DWT Implementation styles summary

Implementation	#Add	#Mul	#3:2	#Mux	#Regs	#Preds	Lat
Conventional	1	2	0	32	4	0	28
Mod-CSA	1	2	2	35	4	0	26
Extr-CSA	1	2	6	46	4	0	20
MS-Trees	1	2	0	34	4	9	19-22

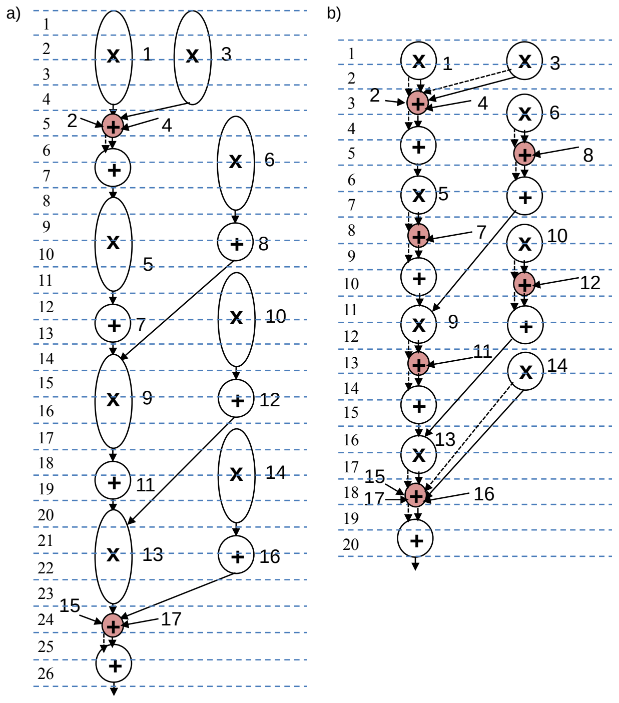


Figure 5. DWT with a) moderate CSA, b) extreme CSA

observed, although the average latency of the multispeculative implementation (*MSTrees*) depends on actual inputs, it is expected to achieve a similar performance to the Extr-CSA, but with fewer resources. The MSTrees penalty is only due to the predictors, their routing and the routing of the recovery addition destination register. However, if this register is already being used as a source, there will be no penalty because of that. On the contrary, in the Extr-CSA, if a CSA tree is reused, it requires a noticeable amount of additional routing resources. Besides, when utilizing CSA trees, there will be more *n*-bit outputs to store.

Therefore, MSTrees are an efficient implementation style, as they combine both a good performance and low area overhead. On the contrary, CSAs introduce a higher area overhead when optimizing whole circuits for high performance, and possess a bad performance when considering a less penalizing implementation choice.

V. EXPERIMENTS

In this section our experimental framework is first described. Afterwards, our experimental results are discussed.

A. *Framework*

Our RT-level implementations have been generated following the flow detailed in figure 3. The resulting datapaths and controllers have been automatically coded in VHDL and synthesized with *Synopsys Design Compiler*, with a 65 nm library. Area, including routing and controller, is measured in μm^2 and delay in nanoseconds. A simulator has been built in order to measure the number of hits and mispredictions, as well as the average number of cycles. The same FU latencies as in subsection IV.A have been considered, and the benchmarks have been simulated for 10^6 iterations to obtain execution time and area results. During the simulation, inputs are modeled stochastically, in a similar fashion to the profiling information obtained in [20] by Brooks and Martonosi. Taking into account the data presented in [20], the most significant fragment of the two operands consists of a sign extension with a high probability (>0.9). On the contrary, the least significant fragments behave roughly random.

Finally, average execution time is obtained as the product of both the average latency and the cycle time given by *Synopsys*.

B. *Results*

In order to check the performance and area of the MSADDs, several experiments have been performed.

1) *Area-Delay results*

Several benchmarks have been synthesized in order to get area and delay results, namely:

- The 16-bit *Dilation* inner loop code, used in the ECG [1].
- The 16-bit *Accum* submodule in the ADPCM decoder [3].
- The 16-bit Discrete Wavelet Transform (*DWT*) [5].
- The 16-bit Finite Impulse Response (*FIR*) filter [23].
- The 16-bit Autoregressive Filter (*ARF*) [5].
- The 32-bit Simpson 3/8 numerical integration (*Simpson38*) [2].
- The 32-bit Trapezoid numerical integration (*Trapezoid*) [2].
- The 16-bit *dot* product of two vectors, each of 8 components (*Dot_8*).

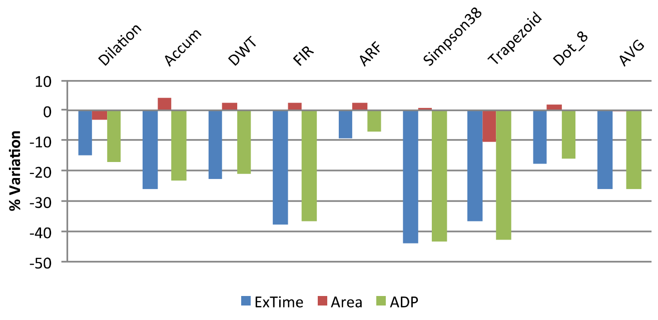


Figure 6. Execution time gain, area penalty and Area Delay Product variation percentages

In this first experiment a comparison between the baseline implementation and our proposal (*MSTrees*) is performed. The baseline flow is composed of conventional list-scheduling and left-edge binding algorithms using non-speculative FUs, while our flow is the one explained in section IV. Figure 6 depicts the execution time, area penalty and *Area Delay Product* (ADP) variation percentages. It should be noted that the rightmost set of bars in figure 6 shows the average results. These percentages have been calculated with respect to the aforementioned baseline implementation.

As it is shown, multispeculative results improve execution time by 26% (44% in the best case) on average. In terms of area, the average area penalty of the multispeculative results is close to zero. This is due to the fact that MSFUs require less area then their high performance non-speculative counterparts and thus help to compensate for the overheads of any other additional structures . This fact compensates for the overhead due to the additional routing and control. Finally, when considering ADP, there is a reduction by 26%.

2) About the impact of Carry-Propagate Additions

In this second experiment, the impact of the additional carry-propagate additions over the circuit latency is evaluated. As it has been previously mentioned in section IV, one of the MSTrees counterparts is the introduction of additional operations, namely: recovery additions or carry-propagate additions. The number of recovery additions is fixed and it is equal to the number of additive trees in the circuit. However, the number of carry-propagate additions is a parameter that depends on the resource allocation, as it has been shown in figure 2.

In order to prove that the additional carry-propagate additions have no impact over the circuit performance, table II contains a latency study in the case of the FIR filter, considering several resource sets for our scheduling and binding algorithm. The two leftmost columns depict the number of adders and multipliers, while the third column refers to the number of required carry-propagate additions (*CP*). It should be noted that the number of recovery additions is fixed, as this benchmark possess 9 additive trees, i.e. recovery additions, regardless the resources set. Finally, the three rightmost columns show the latency in the case of a conventional implementation, and the expected and measured latency with the MSTrees implementation, respectively.

As it can be observed, the number of additional carry-propagate additions is never greater than 2, and latency is reduced 40%

Table II. Resource allocation impact over the FIR filter latency

#+	#*	CP	Lat	MS-Lat Exp	MS-Lat Real
1	1	0	38	26-28	26.51
2	1	0	38	27-29	27.60
1	2	1	30	23-25	23.85
2	2	1	28	15-17	15.84
3	2	1	28	15-17	15.54
2	3	1	26	14-16	14.45
3	3	1	24	12-14	12.74
4	2	1	30	15-17	15.72
4	3	1	26	12-14	12.76
2	4	2	24	14-16	14.65
3	4	2	22	11-13	11.57
4	4	2	20	11-13	11.48

on average. Besides, as it can be observed, the real latency is always closer to the lower bound of the expected latency range. Hence, in spite of containing from 9 to 11 extra additions, MSTrees have proved to be really efficient, as they allow all the operations to be executed in short latency mode. Thus, the possible latency penalty due to recovery additions or to the extra carry-propagation additions is clearly compensated.

3) Other Speculative Implementations

In this third experiment, we have compared the execution time and area variation percentages of two multispeculative implementations without [18] and with Multispeculative Trees management (*MSTrees*), utilizing logarithmic modules in both options. The baseline multispeculative controller is a *stop the world* method, which introduces a stall in the datapath whenever there is a misprediction regardless of the FU. Figure 7 shows these results. Every set of columns depicts first the percentage variation in execution time without and with MSTrees, and second the percentage area variation without and with MSTrees, respectively. As it can be observed, our proposal reduces 12% more execution time than previous controllers with no management of the additive trees, because predictions do not have to be considered in pipelined mode. This fact is the main reason of the decrease in the number of mispredictions, which leads to the execution time reduction. Also, our proposed solution incurs 5% less area overhead, because of the reduction in the number of structures utilized for checking mispredictions.

4) Comparison with CSA implementations

In order to check the impact of using CSAs, two sets of experiments have been performed. On the one hand, figure 8 a) shows the execution time, area and ADP variation percentages of pure CSA benchmarks, i.e. without multipliers. Conventional implementations require a CPA in the last stage, while the multispeculative ones substitutes it by a MSKS. In this case, it can be observed how the MSKS reduce the delay of the last stage. Area overhead is negligible. Besides, figure 8 a) depicts that the execution time decrease is greater in the case of 32-bit implementations, as explained in figure 6. On the other hand, figure 8 b) shows the same results but with 16-bit circuits which contain products. In this case, conventional implementations are implemented with the Extreme CSA style, as explained in section IV. C, while the multispeculative ones are the same as in figure

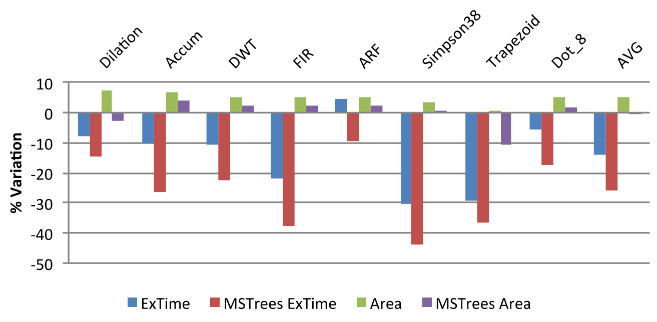


Figure 7. Execution time and area percentages variation with respect to a conventional implementation without and with Multispeculative Trees

6. It can be observed, that CSA implementations are 5% faster on average (10% in the worst case). The reason is that CSAs are limitted by the CPA in the last stages of the additive trees. However, they require a larger area overhead, because of the additional CSA trees, routing resources and registers. In fact, the ADP is 7% lower on average in the MSTrees case.

Figure 8 c) presents the same results as figure 8 b), but with 32-bit precision. As it can be observed, the MSTrees execution time percentage variation is larger than with 16-bit in all the benchmarks but the *Dot_8* one. With larger precision, the delay reduction of the CPA stages becomes more significant. However, the *Dot_8* benchmark is a quasi-pure CSA one, as every product appears at the leaf nodes of the DFG. Hence, substituting the last CPA by a MSADD as in figure 8 a) will produce better results, as MSADDs are faster than non-speculative adders as a single entity [21]. In terms of area, with 32-bits the reduction percentage is lower than with 16-bits because of the MSTrees' weight diminish with respect to the overall value.

Hence, after this experiment, it has been concluded that there are two main factors which affect the quality of each implementation choice:

- The amount of products. If there are many products, in general an Extreme CSA implementation will work better, as the Partial Product Matrix (PPM) calculation is faster than the PPM+MSADD.
- The number of CPAs. If there are many CPAs, in general MSTree implementations will work better, because MSADDs are faster than CPAs, and multispeculative multipliers are not much slower than only the PPM.

Nevertheless, as both factors are related, it is not easy to establish a general rule. If there are many products, the previous

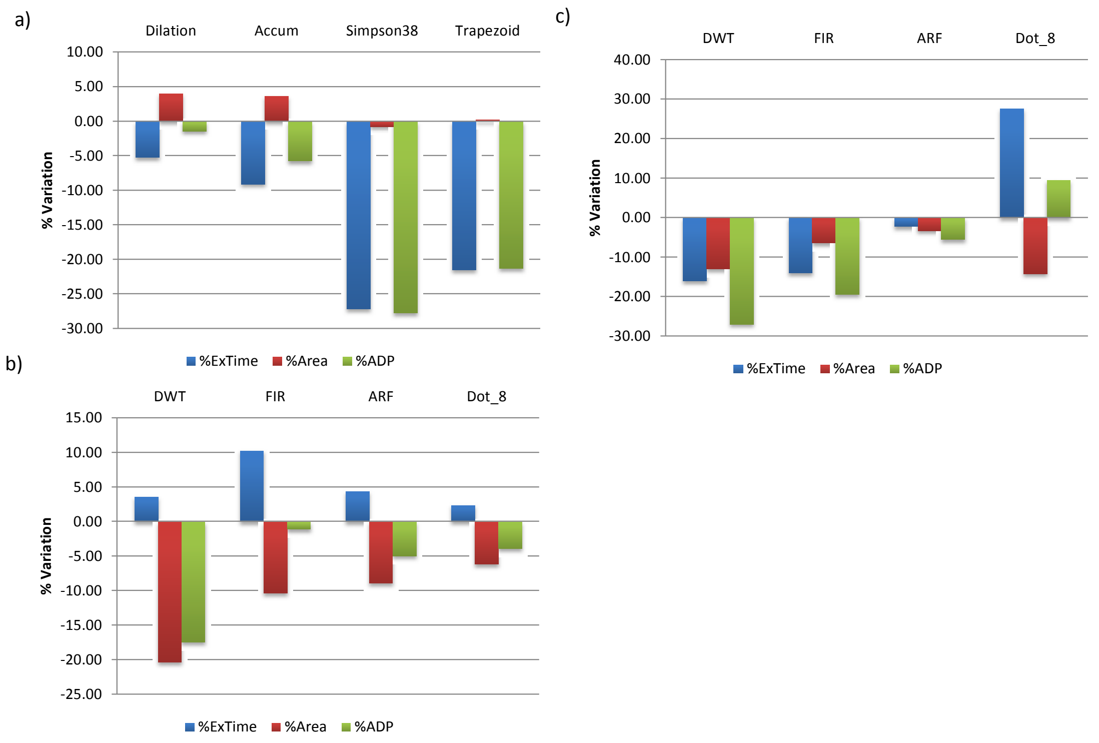


Figure 8. Execution time, area and Area Delay Product of a) pure CSA benchmarks and b) 16-bit and c) 32-bit non-pure CSA benchmarks

additions will need to be CPAs, which is profitable for MSTrees. On the other hand, if the products appear in the early stages, as the *Dot_8* case, an Extreme CSA implementation will work better, as there will be few CPAs before the products. Furthermore, datawidth also impacts on the weight of these factors.

VI. CONCLUSIONS

This work introduces multispeculation in additive binary trees. The main idea consists of pipelining the carry vectors in the inner nodes of the trees. Thanks to the associative property of addition, carry vectors that cannot be consumed by active adders can be accumulated at a later stage. In this way, only the last addition of the entire structure can potentially suffer a misprediction. However, as stated in the paper, an additional cycle is enough for correcting all the mispredictions with a high probability. Hence, it is possible to reduce the number of operations that could potentially mispredict, and hence increase performance. Our experimental results show an average 26% improvement in execution time when considering logarithmic modules, with no area overhead. Besides, our flow is more efficient than previous approaches utilizing Speculative FUs.

On the other hand, as it has been shown in figure 8, MSADDs can also be combined with CSA implementations, or even improve CSA results for certain benchmarks. In the future, the application of Distributed Management [18] techniques will also help to lessen the number of mispredictions and reduce the critical path of CSA-based implementations.

REFERENCES

[1] Y. Sun, K.L. Chan, S.M. Krishnan, “ECG signal conditioning by morphological filtering”, Computers in biology and medicine, 2002, vol. 32, pp. 465-479.

[2] R.L. Burden, J.D. Faires, “Numerical Analysis”, Brooks/Cole Cengage Learning, 9th ed., 2000.

[3] 40, 32, 24, 16 kbits/s Adaptative Differential Pulse Code Modulation (ADPCM). Recommendation G.726. ITU.

[4] I. Koren, “Computer Arithmetic Algorithms”, A K Peters, 2nd ed, 2002.

[5] S.P. Mohanty, N. Ranganathan, E. Kougianos, P. Patra. “Low-Power High-Level Synthesis for Nanoscale CMOS”, Springer, 2008.

[6] P. Coussy, A. Morawiec, “High-Level Synthesis. From Algorithm to Circuit Design.”, Springer, 2008.

[7] G. De Micheli, “Synthesis and Optimization of Digital Circuits”, 1st edition, McGraw-Hill, 1994.

[8] A.K Verma, P. Brisk, P. Ienne, “Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits”, IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, no. 10, pp. 1761-1764, Oct. 2008.

[9] H. Parandeh-Afshar, A.K. Verma, P. Brisk, P. Ienne, “Improving FPGA Performance for Carry-Save Arithmetic”, IEEE TVLSI, Vol. 18, pp. 578-590, 2010.

[10] T. Kim, J. Um, “A Practical Approach to the Synthesis of Arithmetic Circuits Using Carry-Save-Adders”, IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 5, pp. 615-624, May. 2000.

[11] S.M. Nowick, “Design of a low-latency asynchronous adder using speculative completion”, IEE Proc. Comput. Digit. Tech., 1996, vol 143, pp. 301-307.

[12] A.A. Del Barrio et al. “Applying speculation techniques to implement functional units”, ICCD, pp. 74-80, 2008.

[13] A.K. Verma, P. Brisk, P. Ienne, “Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design”, DATE 2008, pp. 1250-1255.

[14] D. Bañeres, J. Cortadella, M. Kishinevsky, "Variable-Latency Design by Function Speculation", DATE 2009, pp. 1704-1709.

[15] H.F. Ugurdag, O. Baskirt, "Fast parallel logic circuits for n2n round-robin arbitration", Microelectronics Journal, vol. 43, pp. 573-581, 2012.

[16] L. Benini, E. Macii, M. Poncino, G. De Micheli, "Telescopic Units: A New Paradigm for Performance Optimization of VLSI Designs", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, 1998, vol. 17, no. 3, pp. 220-232.

[17] V. Raghunatan, S. Ravi, and G. Lakshminarayana, "Integrating Variable Latency Components into high-level synthesis", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 10, pp. 1105–1117, Oct. 2000.

[18] A.A. Del Barrio, S. O. Memik, M.C. Molina, J.M. Mendias, R. Hermida, "A Distributed Controller for Managing Speculative Functional Units in High-Level Synthesis", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, Vol. 30, no. 3, pp. 350-363, 2011.

[19] L. Dadda, V. Piuri, "Pipelined Adders", IEEE Trans. On Computers, 1996, vol 45, no. 3, pp. 348-356.

[20] D. Brooks, M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance", HPCA 1999, pp. 13-22.

[21] A.A. Del Barrio, R.Hermida, S. O. Memik, J.M. Mendias and M.C. Molina, "Multispeculative Addition Applied to Datapath Synthesis", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, Vol. 31, no. 12, pp. 1817-1830, 2012.

[22] G. Lakshminarayana, A. Raghunathan, N.K. Jha, "Incorporating Speculative Execution into Scheduling of Control-Flow-Intensive Designs", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 3, pp. 308-324, March 2000.

[23] J. Xie, J. He, G. Tan, "FPGA realization of FIR filters for high-speed and medium-speed by using modified distributed arithmetic architectures", Microelectronics Journal, vol. 41, pp. 365-370, 2010.

[24] J. Um, T. Kim, "An Optimal Allocation of Carry-Save-Adders in Arithmetic Circuits", IEEE Trans. On Computers, vol. 50, no. 3, pp. 215-233, March 2001.

[25] S.L. Lu, "Speeding Up Processing with Approximation Circuits", IEEE Computer, vol. 37, no. 3, pp. 67-73, 2004.

[26] K. Du, P. Varman, K. Mohanram, "High Performance Reliable Variable Latency Carry Select Addition", DATE 2012, pp. 1257-1262.

[27] D. Koes, T. Chelcea, C. Oneyeama, S.C. Goldstein, "Adding Faster with Application Specific Early Termination", Carneggie Melon University Technical Report, no. CMU-CS-05-101, May 2005.

[28] S.Almukhaizim, P.Drineas, Y.Makris,"Compaction-based concurrent error detection for digital circuits",Microelectronics Journal,vol. 36, pp. 856-862, 2005.

[29] A.A. Del Barrio, R. Hermida, S.O. Memik, J.M. Mendías and M.C. Molina, "Multispeculative Additive Trees in High-Level Synthesis", DATE, pp. 188-193, 2013.