Multispeculative Additive Trees in High-Level Synthesis

Alberto A. Del Barrio*, Roman Hermida*, Seda Ogrenci Memik#, Jose M. Mendias*, Maria C. Molina*
*Architecture and Technology of Computing Systems, Universidad Complutense de Madrid (UCM), Spain
#Department of Electrical Engineering and Computer Science (EECS), Northwestern University, Evanston, Illinois
albertodbg@fdi.ucm.es, rhermida@dacya.ucm.es, seda@eecs.northwestern.edu, {cmolinap, mendias}@dacya.ucm.es

[1]*Abstract*— **Multispeculative Functional Units (MSFUs) are arithmetic functional units that operate using several predictors for the carry signal. The carry prediction helps to shorten the critical path of the functional unit. The average performance of these units is determined by the hit rate of the prediction. In spite of utilizing more than one predictor, none or only one additional cycle is enough for producing the correct result in the majority of the cases. In this paper we present multispeculation as a way of increasing 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 in certain selected nodes, thus minimizing 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 on average 24% and 38% execution time, when considering logarithmic and linear modules, respectively.**

Index Terms **— Speculation, operation trees, High-Level Synthesis.**

I. Introduction

There are many DSP and multimedia applications composed of one or several additive structures. Addition chains or trees usually appear in signal processing applications such as ECG [1], numerical integration methods [2], or the ADPCM [3]. Thus, it is crucial to improve the quality of adders and adder-dominated structures without incurring significant area or power overhead.

Historically there have been two points of view to face the abovementioned problem. On the one hand, increasing the adders' complexity and thereby, their speed. Various adder designs have been proposed to achieve different trade-offs between hardware complexity and performance [4]. All such adders exhibit a certain fixed amount of latency. A different design option is the Carry Save Adder (CSA) [4], where the carry propagation is accelerated at the expense of utilizing many Full-Adder cells. Despite recent contributions towards Dataflow Graph (DFG) transformations to optimize the use of CSAs [7], the application of these structures still restricts the use of crucial High-Level Synthesis (HLS) techniques such as module sharing. In order to maximally reuse a CSA-tree, a similar cluster of nodes must appear several times in different control steps, which is not as easy to encounter as in the case of a single addition node.

On the other hand, the introduction of Variable Latency Functional Units (VLFUs) has augmented the possibilities in the design space [8-11]. Although literature offers several possibilities for handling these units [11-14], all of them are subject to the possibility of the worst-case timing, i.e. VLFUs working in long latency mode. Some of the VLFUs available in literature [14] are based on various forms of speculation over a carry value. We shall refer to them as Speculative FUs (SFUs). In this paper, first we extend the concept of SFUs with the application of speculation over multiple points of the addition carry chain. Hence, in this work, Multispeculative FUs (MSFUs) will be described and afterwards utilized to optimize additive structures. We propose to take advantage of these structures for reducing the number of mispredictions and thus increasing overall performance. The basic idea is to pipeline some intermediate carries from one addition to the other, in the case of the inner nodes of the additive structure. In this way, the number of possible mispredictions will be diminished. According to our experiments it is possible to reduce execution time by 24% and 38% on average with respect to a baseline implementation with non-speculative logarithmic and linear FUs, respectively.

The rest of the paper is organized as follows: section II presents the Multispeculative Adder and the additive structures where it can be introduced. Section III describes the application of multispeculation over the datapaths. Finally, sections IV and V present our experimental results and conclusions.

II. Multispeculative Additive Structures

A generic n-bit Multispeculative Adder (MSADD) consists of n/k fragments of width k that operate in parallel, whose carry-in is provided by a predictor. A *hit* signal will indicate when the operation is correct: iff all the true carry-out values of each module are equal to the corresponding prediction of the carry-in values of the following module. Each predictor will be updated with the true carry-out iff there is a misprediction. The main idea for using MSADDs is that despite utilizing more than one predictor, almost every addition will be executed in two very short cycles at the most, because the probability of propagating a misprediction from a fragment to the following one is the same as finding a chain of k propagates signals, which is very low if k is large enough [8,9]. Nevertheless, the objective must be reducing the latency of the whole additive structure, instead of a single addition. Our idea consists of pipelining the inner nodes' carries from a cstep to the following

[1] This work was supported by the Spanish Government Research Grant TIN 2008/00508
[1] 978-3-9815370-0-0/DATE13/©2013 EDAA

[15] during the first cycles, avoiding thus any penalty cycles during this first set of operations. Finally in the last cycles, predictors will be utilized as usual, reducing the critical path of the last stage, which is the main limitation of CSA structures, and thus increasing performance.

Figure 1 shows our proposed MSADD. This MSADD is specially oriented to implement additive structures. Hence, as prediction is only applied in the last stage we have chosen the simplest predictors, i.e. D-flipflops, for diminishing as much as possible the critical path of the MSADD. Thereby, in the last cycles a failure will be produced iff a carry-out from any fragment is '1', i.e. different from the prediction. The D-flipflops 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 the D-flipflops must always be written and the carries pipelined during the first csteps, and the *hit* signal considered from the last cycle onwards, i.e. in prediction mode. In this last cstep/state a control mechanism similar to [12,13] is utilized, i.e. if the *hit* signal is false there will be a transition to a correction state, and otherwise there will be a transition to the following state.

In the next subsections, a more detailed explanation of how these principles apply to concrete structures will be presented.

A. Scheduling of Addition Chains

In general, a conventional addition chain of L 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. Note that in the best case the chain only requires L-1 cycles and in the worst case (L-1)+n/k-1 cycles. It should be reminded that in addition to this the MSADD is divided into k-bit fragments, and thus cycle time will become proportional to the k-bit delay, instead of the n-bit delay.

Let us assume an addition chain A+B+C+D with 16-bit operands, i.e. L=4. A conventional chain-like implementation will take 3 cycles. In the multispeculative case, it will require 3+1=4 cycles with a high-probability. Figure 2 depicts the two most common cases when applying multispeculation. Figure 2 a) shows this addition chain in a case of primary hit. During the first 2 cycles the intermediate carries are pipelined, but in the last stage a static zero prediction is considered. As the carry-out signals proceeding from the intermediate fragments are '0' (see the red dotted lines), there is no need for additional cycles.

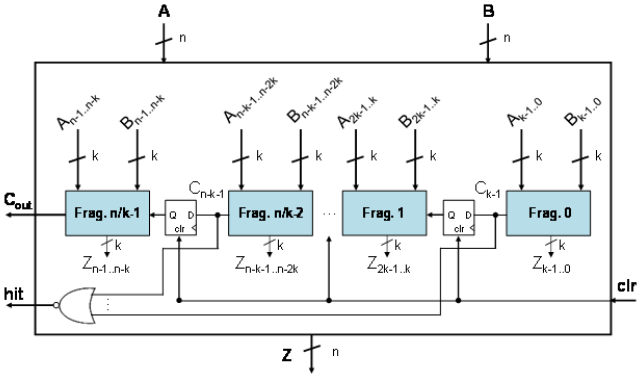

Figure 1. MSADD for implementing additive structures

Figure 2 b) on the contrary, depicts a case of misprediction in cycle 3, because there are two intermediate carries equal to '1'. In the next cycle these logic '1' values are added, and as the resulting carry-out signals are '0', there is a hit in cycle 4.

Finally, let us emphasize the importance of prediction in the last cstep. If no prediction were performed, it would take n/k = 16/4 = 4 cycles to propagate the carries and thus certify the correction of the addition. Overall we would require (L-1)+n/k-1=6 cycles, which is 50% more in comparison with the expected L=4 cycles.

B. Scheduling of Full Binary Addition Trees

Figure 3 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. 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 figures 3 a) and 3 b), 4 non-speculative adders and 4 MSADDs have been considered, respectively.

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 non-active adders, thanks to the associative property of the addition. This is the key idea: in any full binary adder tree there is always a level above which all the previous levels contain more additions than available adders, and below which all the following levels contain more available adders than additions. Hence, it is guaranteed that there will be idle adders available.

In figure 3 a) the datapath only requires 3 cycles. On the other hand, 4 short cycles are enough in figure 3 b), with a high probability. As it can be observed, in cstep 1 C1, C2, C3 and

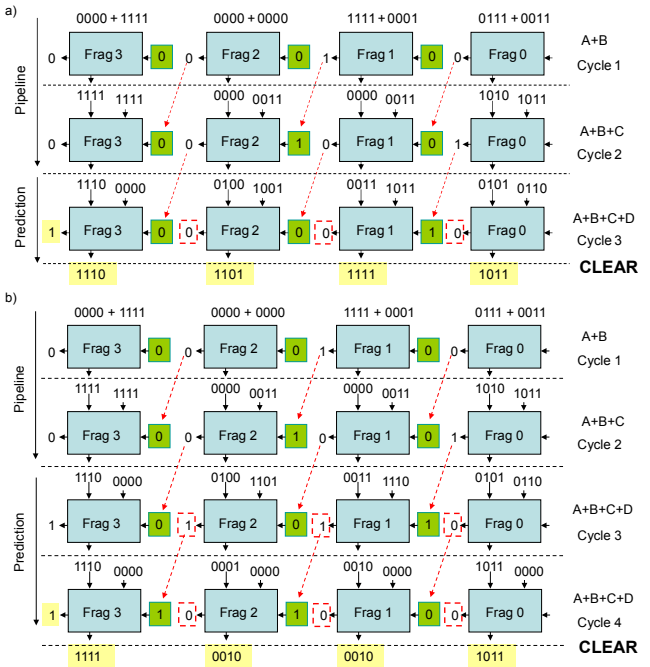

Figure 2. Multispeculative flow for three 16-bit additions with 4-bit blocks, a) primary hit, b) failure and one correction cycle cases

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 D-flipflops of A2, and C4. It should be pointed out that the carry-out vector of A2 in cstep 2 will be full of zero's because k>1. Hence, the result vector coming out from A2 will be enough for accumulating three input vectors that in the worst case will only contain a '1' in the least significant position of each MSADD submodule. This new addition will be called *carry-propagation addition*. In cstep 3 we will proceed in the same manner as with the addition chains, and A1 will 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 both output vectors from A1 plus the accumulated result from A2. In cstep 3', A1 will work in prediction mode until it produces a hit. Hence, if the carry-out vector C1='0', the result Z'' will be equal to Z. 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.

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.

Lemma 1. Provided that the number of operands is $L=2^m$, and the number of available adders is $s=2^r$, where $m,r>=0$, the number of required csteps is given by equation (1).

$$L/s\left(\sum_{i=1}^{\log(L/s)} 2^{-i}\right) + \log(s) = L/s + \log(s) - 1 \qquad (1)$$

Proof. Provided that $L=2^m$, $s=2^r$, in every level i of the tree there will be $(L/2^i)$ additions. Hence every level will take $(L/2^i)/s$ csteps. This will be the case until $s=L/2^i$, i.e. $i=log(L/s)$. This sums L/s-1 csteps. From this tree level onwards, log(s) csteps will be required. Note that the similar demonstration with $2^m < L < 2^{m+1}$ and $2^r < s < 2^{r+1}$ cannot be presented in this paper due to space restrictions. □

In the example depicted in figure 3 a), L=8 and s=4, thus 8/4+log(4)-1=3 csteps are required. On the other hand, the multispeculative version shown in figure 3 b) takes 4 cycles with a high probability. Therefore, it can be concluded that if a non-speculative full binary tree of L operands requires L/s+log(s)-1 cycles, with monocycle FUs, its multispeculative version will take L/s+log(s) cycles at the most with a high probability. It should be noticed that this additional cycle corresponds to cstep 3' in figure 3 b). This is what we shall call 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, an additions' chain is a simplified case of tree where s=1. On the other hand, with multicycle FUs multispeculation will also reduce the average latency of the circuit. However, this reduction cannot be exactly quantified because it depends on the latency of the individual FUs. The example in section III.A will serve to illustrate this latency reduction.

III. 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 when the DFG is composed of several additive trees. In such cases, we will restrict the use of multispeculation only to those additive trees. Similar principles as those described for full binary trees, will be applied to generic additive trees. An additive tree is one that is composed only of addition nodes, except for the leaf nodes, that can also be products. In this case, we can take advantage from 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 a multiplier *Mi* can be expressed as the sum of two bit vectors *Vi* and *Ci* such that $S_{Mi}=Vi+Ci$. Only a slight modification of the MSADD presented in figure 1 is required. A multiplexer at the input of every submodule will be required in order to select either the carry stored in the own D-flipflop of the MSADD, or another carry coming from a multiplier.

Hence, it is necessary to design an algorithm to identify those additive structures where multispeculation can be applied and develop an automated procedure to keep the datapath in a correct state. Finally an interface for communicating with the controller must be defined. This interface will be composed of

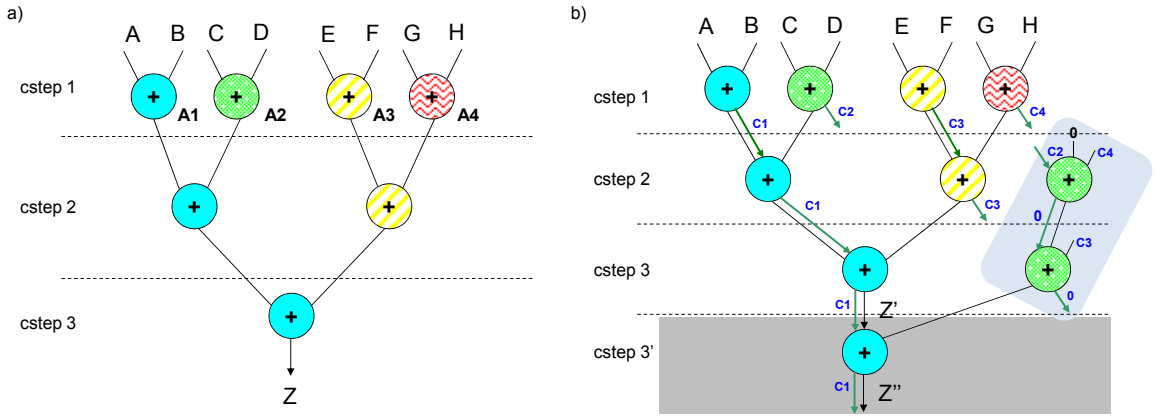

Figure 3. Scheduling, binding and carries propagation of the Z=(A+B+C+D+E+F+G+H) addition tree with a) 4 non-speculative adders, b) 4 MSADDs

a *hit* signal collected from each unit [14]. The controller will be similar to the one presented in [12,13], i.e. if there is a failure there will be a transition to a correction state, and otherwise a transition to the following state.

Moreover, the HLS flow must be adapted to the special features of multispeculative trees with new algorithms. 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, because they will not be available for a different additive structure until the associated carry vector of the operator under question has been consumed. However, as it will be shown in the following discussion, with a reasonable number of resources a penalty on the latency can be avoided.

Figure 4 illustrates our design flow. First, all the additive binary trees are identified, by finding those node sets where any internal result is never multiplied. Second, a *recovery addition* per tree is included in the DFG. And third, the *carry propagation additions* are introduced where required, e.g. in the example of figure 3 b). 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 DFG is scheduled and bound. In our case, due to the aforementioned operators' restriction, we have applied a combined resource constrained scheduling and binding algorithm [6]. 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 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, the FU is not ready to be bound to this new operation, as 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.

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. We have used logarithmic like non speculative adders and multipliers that take 2 and 4 cycles, while their multispeculative counterparts require 1 cycle and 3 cycles, respectively, in case

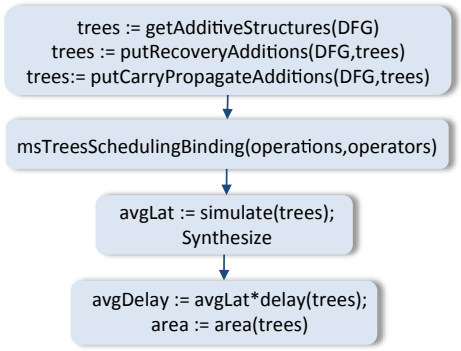

Figure 4. Design flow for multispeculative additive structures

of a primary hit. It must be noticed that multispeculative multipliers are composed of a CSA matrix and a MSADD last stage. Hence, the non speculative latencies have been derived as $log(n)$ and $2log(n)$, and the speculative ones as $log(k)$ and $log(n)+log(k)$, for both the adders and multipliers, respectively. Thereby, using n=16 and k=4, it is possible to obtain the aforementioned values.

Figure 5 a) depicts the DFG of the DWT. Operations are labeled with numbers 1 through 17. Figure 5 b) corresponds to a conventional resource constrained scheduling [6] using 2 non-speculative multipliers and 1 adder. As it can be observed, it takes 28 csteps. Figures 5 c), d) and e), illustrate the application of our methodology. First, all the additive binary trees are identified. Figure 5 c) depicts the 6 trees that comprise the DWT DFG. Second, a recovery addition is introduced at the end of every tree. This is shown in figure 5 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 must be introduced when necessary. In this example, as the carry propagation can be performed through their own tree nodes, there is no need to introduce any one. Finally, this new DFG is scheduled and bound considering that all the operations will produce a hit in the prediction. We have utilized the algorithm described in section III, using 2 multipliers and 1 adder, as in the non-speculative case. The resulting schedule and FU-binding is shown in figure 5 e). Note that the FU-binding to the two multipliers is depicted with the use of 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 5 e) will be explained. 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. Finally, let´s consider ***Operation 14***. It could have been scheduled in cstep 10 because M1 is free in this cstep and its carry-out vector is being consumed by ***Operation 7***. However, as ***Operation 9*** belongs to a different unscheduled tree to that one of ***Operation 14***, and it has more priority than ***Operation 14***, it cannot be scheduled yet. Hence, it is scheduled in cstep 11, after scheduling ***Operation 9***.

Dotted arrows in figure 5 e) denote how the carry vectors are propagated. It should be noted that recovery additions do not have an output dotted arrow because they produce a correct result. The csteps are numbered with a label *C* or *C'* in the leftmost part of figure 5 e). The *C'* csteps are recovery csteps that only contain recovery additions. Hence, these *C'* csteps can be skipped in execution if the controller detects a hit. Thus, in figure 5 e), if ***Operations 7***, ***11*** and ***17*** produce a hit, csteps 10', 14' and 19' will actually not be necessary. Besides the C' csteps, every cstep containing a recovery addition will become a recovery cstep. These csteps are highlighted with a solid background in figure 5 e). Hence, 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 19+6 recovery csteps = 25 cycles, which is better than the conventional case. Nevertheless, 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 ***Operations 4***, ***8*** and ***12***. 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 will range between 19 and 22 cycles.

In conclusion, as it can be observed, the application of multispeculation over additive binary trees has reduced the instances of actual mispredictions. If the existence of additive trees had been ignored [14], every operation might 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 diminishes average latency even further.

IV. Experiments

In this section we first describe our experimental framework. Next, we discuss our experimental results.

A. Framework

Our implementations have been generated following the flow detailed in figure 4. A simulator has been built in order to measure the number of hits and mispredictions, as well as the average number of cycles. The benchmarks are 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 [16] by Brooks and Martonosi. Taking into account the data presented in [16], 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. Afterwards the resulting datapaths have been automatically coded in VHDL and synthesized with *Synopsys Design Compiler*, with a 65 nm library. Area is measured in μm^2 and delay in nanoseconds. Finally, average execution time is obtained as the product of both the average latency and the delay 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 [5].
- The 16-bit Autoregressive Filter (*ARF*) [5].
- The 32-bit Simpson 3/8 integration (*Simpson38*) [2].
- The 32-bit Trapezoid integration (*Trapezoid*) [2].
- The 16-bit dot product of two vectors, each of 8 components (*Dot_8*).

Tables I a) and b) present the simulation and synthesis results of these benchmarks for both logarithmic (Kogge-Stone) and linear (Ripple Carry) modules [4], respectively. In both tables, the leftmost column indicates the name of the benchmark, while columns 2 and 3, and columns 4 and 5 contain the execution time and area of the baseline implementation (*Conv*), and the percentage variation of the multispeculative one (*MS*), respectively. This baseline flow is composed of conventional list-scheduling and left-edge binding algorithms using non-speculative FUs.

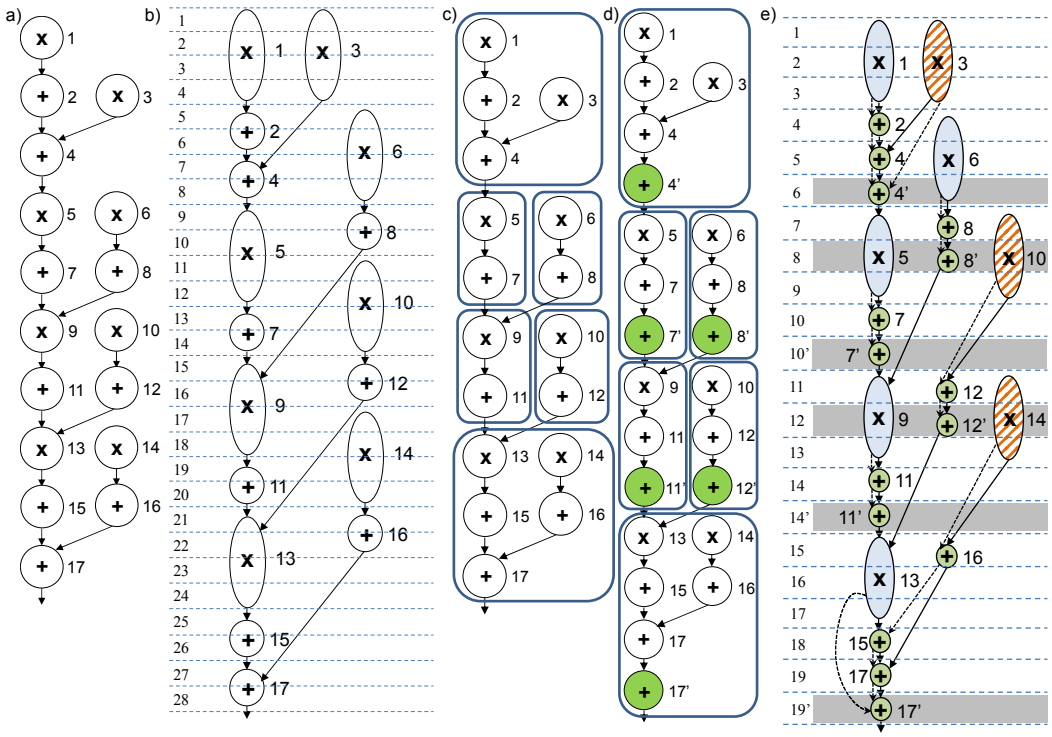

Figure 5. a) Discrete Wavelet Transform (DWT) DFG [5], b) conventional scheduling with non-speculative FUs, c) additive structures in the DWT DFG without and d) with the recovery additions, e) scheduling and FU-binding with multicycle MSFUs

As it is shown, multispeculative results improve execution time by 24% (44% in the best case) and by 38% (63% in the best case) on average for logarithmic and linear modules, respectively. In terms of area, the average area penalty of the multispeculative results is close to zero and 6% for logarithmic and linear implementations, respectively. As it can be observed, execution time reduction and area penalty are lower for logarithmic modules than the linear ones. The reason is that the introduction of multiple prediction points in the logarithmic FU design reduces the number of levels in the carry tree propagation, and hence their area and execution time. This fact compensates for the overhead due to the additional routing and control. On the other hand, linear multispeculative implementations enjoy larger benefits in execution time, because in the baseline design the carry propagation is not as optimized as in the logarithmic case.

2) Scalability study

In this last experiment, the variation of area and execution time with respect to the number of resources has been studied. Figure 6 shows the execution time, area and Area Delay Product (ADP) variation percentages in the *Dot_8* benchmark utilizing logarithmic modules. The X axis depicts the number of adders and multipliers. Note that other configurations with more resources produce the same execution time results than those depicted in figure 6, and that is why they are not shown. The following trend can be observed from these results: the reduction in execution time increases as the number of available MSFUs increases. The only exception happens with the (2,8) configuration. This is because the number of adders is low with respect to the number of multipliers, and hence the multispeculative implementation needs some extra csteps for introducing the carry-propagation additions. Nevertheless, this kind of configuration with such an unbalanced set of FUs is not usual. On the other hand, area penalty remains around 1.5%-2% for all configurations. Hence, ADP gain will increase along with the execution time. Therefore, it can be concluded that multispeculation scales well.

V. 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 increase performance.

Table I. Execution time and area results for a) logarithmic and b) linear-like modules

a)

	Conv		MS	
Benchmark	ExTime	Area	%ExTime	%Area
Dilation	4.28	653	-14.79	-2.87
Accum	3.37	1552	-26.17	3.96
DWT	15.84	9602	-22.63	2.25
FIR	12.34	10091	-20.96	2.36
ARF	19.96	11330	-9.38	2.43
Simpson38	4.87	3007	-43.72	0.36
Trapezoid	3.70	807	-36.48	-10.35
Dot_8	13.16	9721	-17.53	1.78

b)

	Conv		MS	
Benchmark	ExTime	Area	%ExTime	%Area
Dilation	6.16	626	-35.13	2.41
Accum	4.78	1499	-42.63	8.63
DWT	27.11	9437	-29.80	5.13
FIR	21.19	9899	-28.35	5.46
ARF	34.93	11139	-17.16	5.19
Simpson38	8.11	2954	-63.08	5.86
Trapezoid	6.94	687	-59.27	7.25
Dot_8	22.02	9556	-29.41	5.11

REFERENCE

[1] Y. Sun, K.L. Chan, S.M. Krishnan, "ECG signal conditioning by morphological filtering", Computers in biology and medicine, 2002, vol. 32, no. 6, 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] A.K Verma, P. Brisk, P. Ienne, "Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits", IEEE TCAD, Vol. 27, no. 10, pp. 1761-1764, Oct. 2008.

[8] S.M. Nowick, "Design of a low-latency asynchronous adder using speculative completion", IEE Proc. Comput. Digit. Tech., 1996, vol 143, no. 5, pp. 301-307.

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

[10] A. Cilardo, "A New Speculative Addition Architecture Suitable for Two's Complement Operations", DATE 2009, pp. 664-669.

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

[12] L. Benini, E. Macii, M. Poncino, G. De Micheli, "Telescopic Units: A New Paradigm for Performance Optimization of VLSI Designs", IEEE TCAD, 1998, vol. 17, no. 3, pp. 220-232.

[13] V. Raghunatan, S. Ravi, and G. Lakshminarayana, "Integrating Variable Latency Components into high-level synthesis", IEEE TCAD, vol. 19, no. 10, pp. 1105–1117, Oct. 2000.

[14] 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 TCAD, Vol. 30, no. 3, pp. 350-363, 2011.

[15] L. Dadda, V. Piuri, "Pipelined Adders", IEEE TOC, 1996, vol 45, no. 3, pp. 348-356.

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

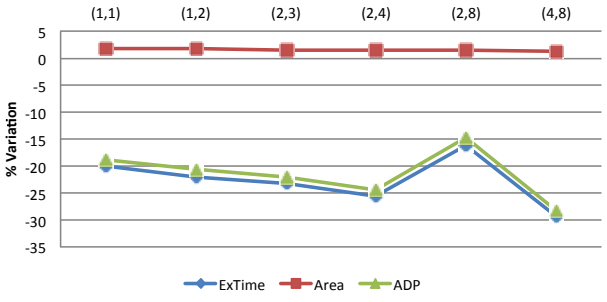

Figure 6. Execution time gain, area penalty and Area Delay Product percentages variation for the dot product with a different set of (adders, multipliers)