Pre-synthesis Optimization of Multiplications to Improve Circuit Performance*

Rafael Ruiz-Sautua, María C. Molina, José M. Mendías, Román Hermida
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid
rsautua@fdi.ucm.es, {cmolinap, mendias, rhermida}@dacya.ucm.es

Abstract

Conventional high-level synthesis uses the worst case delay to relate all inputs to all outputs of an operation. This is a very conservative approximation of reality, especially in arithmetic operations (where some bits are required later than others and some bits are produced earlier than others). This paper proposes a pre-synthesis optimization algorithm that takes advantage of this feature for more efficient high-level synthesis of data-flow graphs formed by additions and multiplications. The presented pre-processor analyzes the critical path at bit-granularity and splits the arithmetic operations into sub-words fragments. In particular, some of the specification multiplications are broken up into several smaller multiplications, additions, and other operations of three new types specially defined to reduce the clock cycle duration. These fragments become the input to any regular high-level synthesis tool to speed up circuit execution times. The experimental results carried out show that implementations obtained from the optimized specification are on average 70% faster and in most cases substantial area reductions are also achieved.

1. Introduction

Conventional high-level synthesis (HLS) algorithms progressively refine a behavioral description to create equivalent register-transfer level (RTL) hardware (HW) implementations of the design subject to various goals, such as performance, area and power consumption. Thus, a trade-off arises among execution times of circuits (latency or number of cycles needed to perform a computation times the clock cycle length), costs of their HW resources and energy consumed. The synthesis process is basically performed in two steps: scheduling (introduces the concept of time and parallelism to set the cycle where every operation starts its execution); and allocation (establishes the HW resources adequate to execute every operation and store and transmit every argument).

Scheduling algorithms assume the worst case delay to relate all inputs to all outputs of an operation. However, in the execution of arithmetic operations neither every input bit is required at the same time, nor every output bit is calculated in parallel. This assumption enormously restricts the design space, and therefore, the set of reachable implementations becomes highly conditioned by the supplied description of the behavior. Traditionally, it has been softened by adding optimization phases at the end of the synthesis process to adjust the design decisions taken [1]-[3]. Furthermore, the increasing complexity of upcoming systems prevents designers from considering the overall effect of description styles on the final implementation. Then, it seems obliged the convenience of HLS algorithms able to produce implementations independent of the descriptive style used to define the behaviors. Until now few design techniques have somewhat taken advantage of this feature: bit-level chaining, non-integer multi-cycle and bit-level synthesis.

Bit-level chaining [4]-[5] allows the execution in the same cycle of several data-dependent operations with rippling effect (e.g., additions and multiplications) to exploit their inherent parallelism. Thus, part of these chained operations can be executed in parallel at the bit-level. Non-integer multi-cycle [4] allows chaining the result produced in one cycle by a multi-cycle operator (that executes one operation across several consecutive cycles) to the next data-dependent operation. In order to reduce the execution time, these two techniques let some bits of several data-dependent operations be computed in parallel. However, in order to chain several operations in one clock cycle, both require that *all* of them finish their executions in the selected cycle. Moreover, multi-cycle adds another disadvantage: the execution of one operation is constrained to consecutive cycles, and the result is solely available the cycle the operation is completed in (even if some result bits have already been computed in prior cycles). Although these two techniques help to reduce either the circuit latency or the cycle duration, the implementations synthesized are still excessively influenced by the description style used in the

* Supported by grant CICYT TIN 2005-5619

3-9810801-0-6/DATE06 © 2006 EDAA

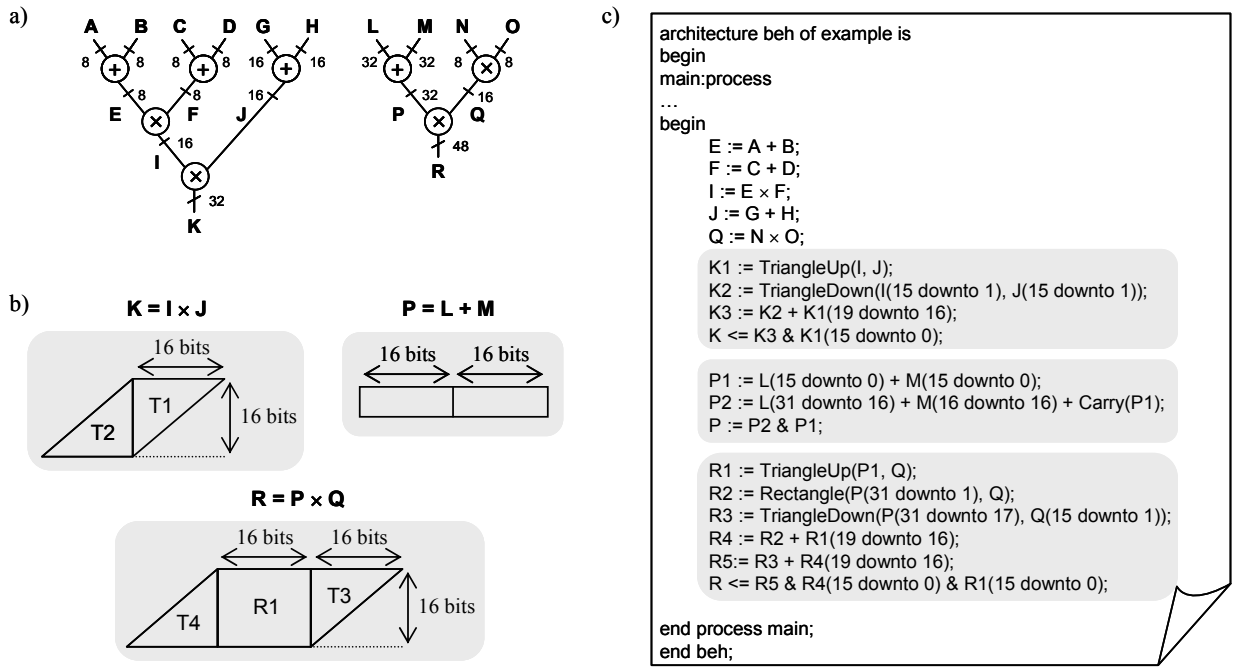

Figure 1. a) Behavioural specification, b) operation fragmentations, c) optimized specification.

specification. Note, for example, that the clock-cycle duration equals the execution time of the longest set of chained operations scheduled in the same cycle.

Bit-level synthesis [6]-[7] aims to minimize the datapath area by increasing the HW reuse at the bit level. In order to achieve this goal, every result bit is considered available (to be used as an input operand) the cycle it is calculated in, even if the overall execution of the operation has not finished yet. Thus, one operation can begin its execution as soon as some bits of its input operands are available, and continue across several not necessarily consecutive cycles as well. Bit-level synthesis algorithms are based on a flow of selective fragmentations of the specification operations. The selected operations are substituted for several chained data-dependent operations whose types and widths may be different from those in the original operation. And some specification operations or operation fragments may be executed over several linked functional units (FUs). Besides, operations with different types, representations, or widths can share the same FU to perform the calculi they have in common. For example, a multiplication comprises several additions that could be executed over the same adders than other additions present in the behavioral specification. This technique overcomes the limitation of conventional allocation algorithms that prevent the binding of two operations to the same FU unless they share their same type, representation and width. However, these bit-level synthesis transformations are mainly focused on HW-reuse gain, and therefore, they do not result adequate for time-constrained synthesis.

This paper presents a pre-synthesis optimization algorithm specially suited for time-constrained designs. It includes a novel method to split multiplications into a set of additions, smaller multiplications, and other operations of three new types specially defined to reduce the clock-cycle duration. The new specification becomes then a better start point for any regular HLS tool to achieve faster circuits.

2. Motivational example

This section provides with the aid of an example, the overlook of how the pre-synthesis fragmentation of multiplications may improve the performance of a circuit without compromising its area. Fig. 1 a) shows a data-flow graph (DFG) of a behavioral description formed by eight operations: four additions and four multiplications. The cycle length of all the schedules of this behavior presented in this section have been calculated taking into account the rippling-effect property of additions and multiplications, which allow the execution in parallel of some of their bits.

Table I shows the operation to cycle assignments synthesized by the commercial tool Synopsys Behavioral Compiler from the behavioral description in Fig. 1 a) with a latency equal to three cycles. Three different synthesis processes have been performed enabling, both separately and jointly, operation chaining and multi-cycle operators. Fig. 1 b) and c) illustrates the operation fragmentations performed to the original specification, and the resultant optimized specification, respectively. Note that

multiplication $K=I\times J$ has been partitioned into two fragments, and has been substituted in the specification for two new operations that execute the calculi included in the triangles marked as *T1* and *T2* plus one addition to sum up the two partial results. Besides, multiplication $R=P\times Q$ has been partitioned in a similar way to produce three fragments, and has been substituted for three new operations that execute the calculi included in the two triangles *T3* and *T4* and the rectangle *R1*, plus two additions to sum the partial results. Finally, addition $P=L+M$ has been also partitioned into two smaller additions.

These multiplication fragmentations may produce operations of three new types: *TriangleUp* that calculates the portion of the multiplication inside a triangle with its right angle situated in the upper left corner part in the figure (triangles *T1* and *T3*), *TriangleDown* that computes the portion inside a triangle with its right angle in the bottom right (triangles *T2* and *T4*), and *Rectangle* that calculates the portion inside a rectangle. In order to synthesize the new specifications, three new components able to execute the new operations have been designed and added to the module library. The schedule synthesized by BC from the optimized specification for latency three cycles (chaining and multi-cycle enabled) is shown in Table II.

Table III summarizes the main features of every implementation. All the values shown have been produced by Synopsys Design Compiler after logic synthesis, and include, in all cases, the routing and controller costs. The execution time of the implementation synthesized from the optimized specification is up to two times faster and it is also the smallest one. Not only the partition of slower operations reduces the cycle length, but also increases the FU reuse among operation fragments (including the new modules as well). These results show that the partition of operations constitutes a great choice to solve the time-constrained scheduling problem at virtually no additional cost in terms of area.

Table I. Conventional schedules of behavior in Fig. 1

	Cycle selected to execute the operation		
Operation	**Chaining**	**Multicycle**	**Chaining + Multicycle**
E = A + B	1	1	1
F = C + D	1	1	2
I = E × F	1	2	2
J = G +H	3	2	2
K = I × J	3	3	3
P = L + M	2	1	1
Q = N × O	2	1	1
R = P × Q	2	2 and 3	2 and 3

Table II. Schedule obtained after optimization

	Variables calculated in every cycle		
Operation	**Cycle 1**	**Cycle 2**	**Cycle 3**
E = A + B	E		
F = C + D	F		
I = E × F		I	
J = G +H		J	
K = I × J		K1	K2, K3
P = L + M	P1	P2	
Q = N × O	Q		
R = P × Q	R1	R2, R4	R3, R5

3. Pre-synthesis optimization method

The proposed algorithm optimizes behavioral specifications in order to obtain faster implementations from conventional synthesis algorithms. This goal is achieved by decomposing slow operations into several smaller, and thus, faster ones, that are assigned to different cycles. The operation types considered are multiplications and additions. The partitions of additions performed are quite similar to the ones proposed in [7]. These new transformations take into account the potential HW reuse as well, in addition to the factors already considered (circuit latency, execution time and mobility of every operation, and data dependencies). This contributes to reduce the circuit area while trying to minimize the execution time. However, the great novelty of this optimization algorithm resides in the multiplication partitioning. Multiplications are decomposed into several new operations of types: addition, multiplication, *TriangleUp*, *TriangleDown*, and *Rectangle*. The three new operation types have been specially defined to obtain the maximum number of result bits of every multiplication fragment, while needing the minimum number of operand bits to be executed. This way, data dependencies among the fragments of every partitioned multiplication are less restrictive, and therefore, the searched design space bigger.

The new algorithm comprises the following phases:

1) *Cycle-length estimation.* The critical path is identified and its duration used to estimate the clock-cycle length.

2) *Operation partitioning.* Some of the operations in the behavioral description are broken up in order to fulfil the time constraint imposed in the previous phase.

Table III. Comparison of implementations

	Execution time (ns)	**Area (equivalent gates)**
Chaining	79,2	11520
Multicycle	53,1	15361
Chaining + Multicycle	52,3	15213
Optimized specification	36,3	11196

3.1 Cycle-length estimation

The first step to estimate the cycle length becomes the identification of the critical path. The critical path of a behavioral description is the path of the DFG that takes the longest time to be executed. In the present version of the optimization algorithm we have considered additive and multiplicative operations, and the time needed to execute every DFG path has been measured in number of 1-bit chained additions. The latency of one *n*-bit addition has been considered equivalent to *n* chained additions of 1 bit, and the latency of one *m*×*n* multiplication equivalent to *m*+*n*-2 chained additions of 1 bit.

To calculate the time consumed by one path, operations are crossed from its output to the input. For every operation crossed, some value is added to the latency of the last operation (the one that produces the path output). The value added in each case depends on the type of the operation crossed, and its successor in the path (the one crossed previously). Four different situations arise:

1) *Addition followed by addition.* The second addition can begin its execution once the least significant bit (LSB) of the first one is calculated. Then the execution time of two chained additions sums one to the latency of the second one.

2) *Addition followed by multiplication.* The LSB of the multiplication can be calculated once the LSB of the addition is available. However, because the time required to calculate the multiplication LSB is nearly negligible compared to the delay of 1-bit addition, we can assume that the calculus of the LSB of both operations can be performed in the time required to execute 1-bit addition. The calculus of the second bit of the multiplication result can begin once the two LSBs of the previous addition are available, and can be executed in parallel with the third bit of the addition. Thus, the execution time adds two to the multiplication latency, and equals the number of bits of the multiplication output.

3) *Multiplication followed by addition.* The two LSBs of the multiplication and the LSB of the addition can be calculated in the time required to compute 1-bit addition. The reason is again that the time needed to calculate the LSB of one multiplication is nearly negligible compared to the delay of 1-bit addition. Thus, the execution time coincides with the delay of the addition.

4) *Multiplication followed by multiplication.* Every result bit of the second multiplication can be calculated once the same bit of the previous multiplication is available. Then two chained multiplications add one to the latency of the second multiplication.

Once the execution time of every different path in the DFG is calculated, the one with the biggest delay is selected as the critical path. Its execution time is used then to estimate the cycle length (measured in number of 1-bit chained additions). It becomes the critical path delay divided by the circuit latency (λ).

3.2 Operation partitioning

The execution time of some specification operations may take longer than the clock cycle length estimated in the previous phase. In order to meet the time constraint imposed, they must be broken up, and their fragments executed in different cycles. At most, the number of partitions obtained from every specification operation equals its latency divided by the estimated cycle length. The algorithm always breaks up every operation to obtain this minimum number of fragments, avoiding in this way an excessive number of operations in the final optimized specification. Two fragments of the same operation have small probability to be scheduled in the same cycle. Therefore the algorithm tries to partition operations into fragments of similar sizes that can share the same set of HW resources, contributing in this way to reduce the implementation area. These fragmentations produce new data dependencies among operations and operation fragments. The execution of one fragment requires the previous execution of the precedent LSB of the same operation (to use the carry out produced as its carry in), and also the bits used as input operands.

3.3 Multiplication partitioning

The number of partitions to be obtained from the fragmentation of one *m*×*n* multiplication equals the division of the multiplication latency, measured in number

Table IV. Features of the modules designed to execute the new operations

	# bits operand 1	# bits operand 2	# result bits	# adders	Set of adders widths
TriangleUp	k	k	k+1	k-1	$\bigcup_{i=2}^{k} \{ i \}$
TriangleDown	k	k	$k + \lceil \log_2 k \rceil$	k-1	$\bigcup_{i=1}^{k-1} \{ i + \lceil \log_2 i \rceil \}$
Rectangle	k	p	$k + \lceil \log_2 p \rceil$	p-1	$\bigcup_{i=1}^{p-1} \{ k + \lceil \log_2 i \rceil \}$

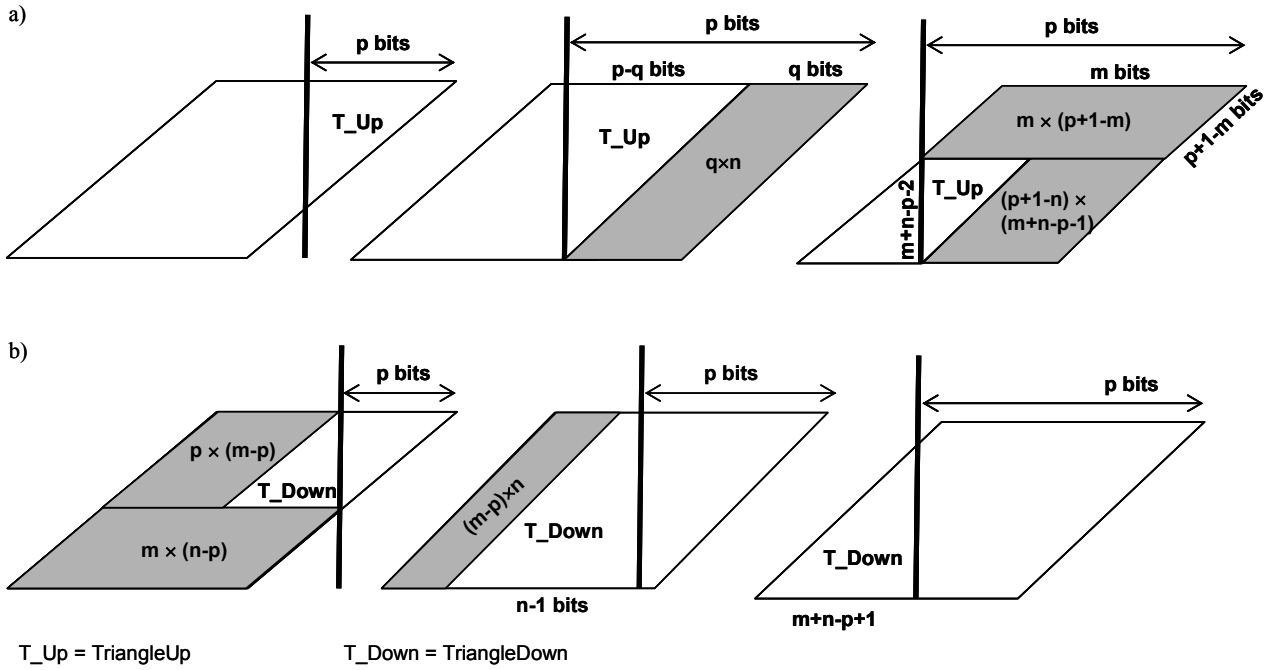

Figure 2. Fragmentations proposed for the a) LSPs, and b) MSPs of one multiplication.

of 1-bit chained additions (m+n-2) by the estimated cycle length. All the partitions are obtained from the vertical fragmentation of the calculus matrix of the multiplication, such that the execution of every fragment produces some bits of the multiplication output. The number of result bits calculated by every fragment is obtained after dividing the width of the multiplication result by the number of partitions. If the number of partitions is a divisor of the width of the multiplication result, then every fragment produces the same number of bits of the multiplication output. Otherwise, some fragments can produce one bit more than others.

The vertical partitions are performed taking into account the number of output bits calculated by every fragment. These partitions do not always directly produce smaller multiplications and additions, and, in most cases, additional fragmentations are required to obtain new operations. The operation types considered in these partitions are: addition, multiplication, *TriangleUp*, *TriangleDown*, and *Rectangle*. *TriangleUp* and *TriangleDown* operations calculate the multiplication portion inside a triangle with a right angle in the upper/lower part of the geometric figure, and *Rectangle* operations calculate the multiplication portion inside a rectangle. These operation types have been specially defined to avoid the excessive fragmentation that occurs when partitioning vertically a multiplication to obtain just new multiplications and additions. They also constitute a regular pattern that appears in most multiplication fragments, contributing in this way to increase the HW reuse and, in consequence, to reduce the circuit area. In order to take advantage of the definition of these new operation types, it is necessary to design and add to the module library new components able to execute them. Table IV shows some features of these three new modules: the number of bits of the input operands and the result, the number of chained adders that compose the partial results, and their widths.

There are three different partition types, depending on their situation in the multiplication matrix: the least significant partition (LSP), central partitions (CPs) and the most significant partition (MSP). The set of operations obtained from every partition type depends on the multiplication and fragments sizes. Fig. 2 a) and 2 b) shows the different LSPs and MSPs captured from the vertical partition of one multiplication and the set of operations given in each case, respectively. The fragmentations obtained from the CPs partitions can be calculated in a similar way. Note that, although the operations shown in the figure are only multiplications, *TriangleUp*, *TriangleDown*, and *Rectangle*, every partition also includes a set of additions needed to sum the partial results.

4. Experimental results

The commercial synthesis tool Synopsys Behavioral Compiler (BC) version 2001.08 has been used to judge the quality of the optimized specifications given by the proposed algorithm. We have synthesized with BC every original specification and the one optimized. The execution time and area of the implementations

synthesized in both cases have been computed with Synopsys Design Compiler (DC). The experimental work includes the optimization and subsequent synthesis of several classical HLS benchmarks [8], and part of a real application.

The classical benchmarks synthesized are a fifth order elliptical wave filter (elliptic), a differential equation solver (diffeq), a fourth order IIR filter (iir4), and a second order FIR filter (fir2). Table V shows the clock cycle duration and the datapath area comparison for several different latencies (λ). Performance has been improved 74% on average, and reductions of the cycle length of up to 88% have been obtained. The datapath area has also been reduced in all cases around 14% on average. The number of operations in the transformed specification is around 36% larger on average.

Some modules of a real circuit description, the adaptive differential pulse-code modulation (ADPCM) encoding and decoding algorithms have also been synthesized. The set of synthesized modules includes: *Inverse Adaptive Quantizer* (IAQ), *Tone & Transition Detector* (TTD), *Output PCM Format Conversion* (OPFC), and *Synchronous Coding Adjustment* (SCA). OPFC and SCA modules have been synthesized together, and IAQ and TTD independently. The latencies used to synthesize the original and the optimized specifications are the ones selected by BC in the conventional schedule (via the command schedule –io_mode free_floating). Table VI shows the cycle length of the schedules obtained from both specifications. The circuit performance has been improved 66% on average. Additionally, the circuit area has been reduced 8% on average. The number of operations in the optimized specification has augmented less than 1/3.

The experimental work shows that the increment in the number of operations in the optimized specification does not complicate the posterior synthesis process and does not augment the design time. The reason is that most of the new operations are born already scheduled, and the mobility of some others has been also reduced.

Table V. Synthesis of some classical HLS benchmarks

	λ	Cycle duration (nanoseconds)			Area reduction
		Original	Optimized	Saved	
elliptic	11	51.59	9.52	81.54 %	11.51 %
	6	60.45	18.03	70.17 %	9.7 %
	4	68.2	19.4	71.55 %	5.25 %
diffeq	6	94.45	33.56	64.46 %	14.85 %
	5	97.56	42.2	56.74 %	19.43 %
	4	101.34	46.43	54.18 %	8.71 %
iir4	6	93.6	11.23	88 %	21.08 %
	5	93.6	16.34	82.54 %	13.25 %
fir2	5	94.57	11.9	87.41 %	17.52%
	3	94.57	17.31	81.69 %	22.24%

Table VI. Synthesis of some modules of ADPCM decoder

Module	λ	Cycle duration (nanoseconds)			Area saved
		Original	Optimized	Saved	
IAQ	3	6.96	2.5	64.08	4.71 %
TTD	5	9.28	3.68	60.34	10.52 %
OPFC +	12	9.39	2.42	74.22	9.18 %

5. Conclusion

The proposed pre-synthesis optimization method improves the results obtained by regular HLS algorithms to synthesize DFGs including multiplications and additions. It analyzes the critical path at bit-granularity and splits some arithmetic operations into sub-words fragments. The fragmented multiplications are substituted for sets of smaller operations whose types are addition, multiplication, *TriangleUp*, *TriangleDown,* and *Rectangle*. The last three types have been specially defined to reduce the clock cycle duration while minimizing the number of new operations split from every fragmented multiplication. They also constitute a regular pattern that appears in most multiplication fragments in order to contribute to increase the HW reuse and reduce the circuit area. The optimized specifications become the input to any regular high-level synthesis tool to speed up circuit-execution times without increasing the datapath area. The experimental results carried out show that implementations obtained from the optimized specification are on average 70% faster and, in most cases, substantial area reductions are also achieved.

References

[1] Z. Yu, K. Khoo, and A. Wilson, Jr. "The Use of Carry-Save Representation in Joint Module Selection and Retiming". In Proc. of Design Automation Conference, DAC 2000.

[2] V. Raghunathan, S. Ravi, and G. Lakshminarayana. "Integrating Variable-Latency Components into High-Level Synthesis". IEEE Transactions on Computer Aided Design, October 2000.

[3] J. Zhu, and D.D. Gajski. "Soft Scheduling in High Level Synthesis". In Proc. of Design Automation Conference, DAC 1999.

[4] S. Park, and K. Choi. "Performance-Driven High-Level Synthesis with Bit-Level Chaining and Clock Selection". IEEE Transactions on Computer Aided Design, February 2001.

[5] P. Marwedel, B. Landwehr, and R. Dömer. "Built-in Chaining: Introducing Complex Components into Architectural Synthesis". In Proc. of Asia Pacific Design Automation Conference, ASPDAC 1997.

[6] M.C. Molina, J.M. Mendías, R. Hermida. "Bit-level Scheduling of Heterogeneous Behavioural Specifications". In Proc. of International Conference on Computer Aided Design, ICCAD 2002.

[7] R. Ruiz-Sautua, M.C. Molina, J.M. Mendías, R. Hermida. "Behavioural Transfromation to Improve Circuit Performance in High-Level Synthesis". In Proc. of Design Automation and Test in Europe, DATE 2005.

[8] N. Dutt, "High-level Synthesis Workshop Benchmarks". Univ. California, Irvine, CA, Technical Report, 1992.