High-Level Allocation to Minimize Internal Hardware Wastage*

M.C. Molina, J.M. Mendías, R. Hermida
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid
{cmolinap, mendias, rhermida}@dacya.ucm.es

Abstract

Conventional synthesis algorithms perform the allocation of heterogeneous specifications, those formed by operations of different types and widths, by binding operations to functional units of their same type and width. Thus, in most of the implementations obtained some hardware waste appears. This paper proposes an allocation algorithm able to minimize this hardware waste by fragmenting operations into their common operative kernel, which then may be executed over the same functional units. Hence, fragmented operations are executed over sets of several linked hardware resources.

The implementations proposed by our algorithm need considerably smaller area than the ones proposed by conventional allocation algorithms. And due to operation fragmentation, in the datapaths produced the type, number, and width of the hardware resources are independent of the type, number, and width of the specification operations and variables.

1. Introduction

Conventional High–Level Synthesis (HLS) allocation algorithms try to obtain RT–level circuits where each operation is executed over a unique functional unit (FU) of its same type and width. When synthesizing heterogeneous specifications some *hardware (HW) waste* (percentage of idle HW resources) appears in almost every cycle. Even more efficient HLS algorithms, able to allocate operations of different widths to the same FU, produce some HW waste if an operation is executed over a wider FU by extending its arguments. In this case some bits of the result are computed but not really needed.

The HW waste due to the *specification heterogeneity* (number of different types and widths in the specification) could be reduced by jointly allocating all *compatible operations* (those with a common operative kernel) independently of their widths. This definition is transitive and considers trivial cases like the compatibility between additions and subtractions, and more complex ones like the compatibility between additions and multiplications. This requires algorithms able to fragment *compatible operations* into their common operative kernel plus some glue logic.

In the datapaths produced by conventional algorithms, the number of FUs of every different type is equal to the maximum number of operations of that type scheduled in the same cycle. Every FU width depends on both the scheduling and the widths of the widest operations. By contrast, in the datapaths designed taking advantage of operations fragmentation, the type, number, and width of the HW resources are in general independent of the specification operations.

To most directly present this new design strategy, the example illustrated in Fig. 1 will be used. Fig. 1a) shows a fragment of a heterogeneous specification and a possible scheduling of it. Fig. 1b) illustrates the implementation proposed by a conventional algorithm, and Fig. 1c) shows our more efficient implementation. In the conventional algorithm implementation the 10×10 multiplier executes operation (A=B×C) in the first cycle and (J=K×L) in the second one; and the 16×4 multiplier executes operation (D=E×F) in the first cycle and (G=H×I) in the second one. Note that in both cycles some partial HW waste appears due to the execution of operations over wider FUs. However, in our approach this HW waste has been minimized by fragmenting the two operations below:

- (A=B×C) has been fragmented into one 10×4 multiplication, one 10×6 multiplication, and one addition of 14 bits, and is executed over the 10×4 and 10×6 multipliers and the 14-bit adder.
- (G=H×I) has been fragmented into one 6×4 multiplication, one 10×4 multiplication, and one addition of 10 bits, and is executed over the 6×4 and 10×4 multipliers and the 14-bit adder.

Note that with this operation fragmentation the HW waste has been completely removed during the first cycle, and highly minimized during the second one. Complete HW waste removal could only be achieved by a specification re-scheduling that fragments some of the operations. However, sometimes data dependencies make datapaths free of some HW waste impossible to construct.

* Supported by Spanish Government Grant CICYT TIC-2002-750

In our example, if we had one addition of m bits scheduled in a third cycle, conventional algorithms would include one adder of m bits in the datapath. Instead, in our solution if $m \leq 14$ then the 14 bit adder would be used. Otherwise one adder of m-14 bits would be included in the datapath, and linked to the existing 14-bit adder to propagate the carry signal, thus the addition would be executed over them.

In the next sections we present a heuristic allocation algorithm especially suited for heterogeneous specifications. Its main aim is to minimize HW waste by fragmenting specification operations into their common operative kernels when necessary.

2. Related work

The HW waste problem appears when heterogeneous algorithms are executed over HW architectures with shared resources. Especially relevant is the case of DSP algorithms (e.g. an ADPCM encoder includes around 20 different data representations and 40 different operations types), which has been addressed from different perspectives:

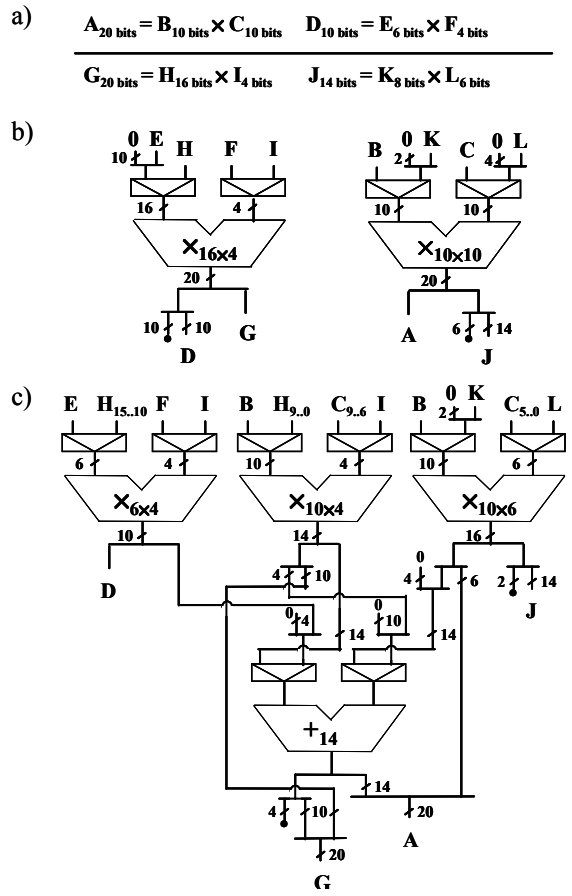

Fig. 1. a) Scheduling of a heterogeneous specification, b) implementation proposed by conventional algorithms, c) more efficient implementation proposed by our approach.

a) *DSP processor programming*. HW waste appears because the DSP software computational model consists of a set of pre-designed fixed word-length computational units responsible of executing all the operations. So, heterogeneous specifications are transformed into other ones whose operation types and widths match these of the computational units. Truncation, extension, rounding, and conversion operators must be applied in order to adjust operation widths [1].

b) *RT-level synthesis of DSP algorithms*. The most common RT-level computational models are: *bit-parallel* (processes a complete word of the input sample per cycle), *bit-serial* (processes a bit of the input sample per cycle) and *digit-serial* (processes a word fragment called *digit* per cycle). The heterogeneity problem appears only in *bit* and *digit-serial* implementations, and it is solved by fragmenting the specification operations into new ones whose widths allow the maximum bit-level reuse of HW resources [2][3].

c) *HLS of DSP algorithms*. Conventional allocation algorithms synthesize heterogeneous specifications by binding operations to FUs of the same width [4][5]. More efficient algorithms allow the execution of operations over wider FUs [6][7]. And in order to obtain smaller datapaths some authors propose algorithms that perform the scheduling and allocation phases at the same time [7]. Another approach [8][9], especially suited for cell-based technologies, minimizes HW waste by using bit-level scheduling and allocation algorithms. It transforms every specification operation into a set of new operations of a unique type (additions), and afterwards performs jointly their allocation (independently of their widths). This binding style leads to datapaths with a unique FU type (adders), and where all complex operations have been fragmented into a set of simpler ones (e.g. one $m \times n$ unsigned multiplication is transformed into n-1 additions of m bits). In this approach unnecessary fragmentation occurs when an equal number of operations of the same type and width can be scheduled in every cycle, or in most cycles. In these cases a selective operation transformation would be required to maintain the benefits of structured designs (regularity, locality…).

3. Proposed algorithm

In the present version of the algorithm the following types of operations have been taken into account: two-complement signed multiplications, unsigned multiplications, additions and *additive* operations (those which can be transformed into additions, e.g. subtractions, comparisons, maximum, minimum, etc). The algorithm works in three phases.

1) *Multiplier selection and binding*. During this phase a set of two-complement both signed and unsigned multipliers is selected. Some specification multiplications are bound to them, other ones are fragmented into smaller multiplications and additions to increase the multipliers reuse, and the remaining ones are transformed into additions that are allocated during the next phase. At the

time of fragmenting multiplications, the algorithm takes into account the following features:

- Any unsigned multiplication may be fragmented into a set of smaller unsigned ones plus some additions.
- Any pair of unsigned multiplications may be fragmented to obtain in each case another unsigned one of the same width. Be $m \times n$, and $k \times l$ ($m \geq n$ and $k \geq l$) the widths of two unsigned multiplications, the biggest common unsigned multiplication which may be obtained from their fragmentations is:

  $m \times n$ if $m \leq k$ and $n \leq l$, $m \times l$ if $m \leq k$ and $l \leq n$

  $k \times n$ if $k \leq m$ and $n \leq l$, $k \times l$ if $k \leq m$ and $l \leq n$
- Any unsigned multiplication may be fragmented into a set of additions, as it is shown in Fig. 2.
- Any two-complement signed multiplication may be fragmented into one unsigned multiplication and two additions, as shown in Fig. 3.

2) *Adder selection and binding*. During this phase a set of adders is selected and every addition bound to it. These additions may come from multiplication fragmentations, additive operation transformations, or be present in the original specification.

3) *Routing selection and binding.* A set of multiplexers is instanced and allocated.

The storage units selection and binding take place during the first and second phases of the algorithm. Each time one operation is allocated, the registers used to store its operands and result are selected. The heuristic algorithm used [10] guarantees the maximum bit-level reuse of registers. In the datapaths obtained some variables may be stored simultaneously in the same register, and some variables may be fragmented and every fragment stored in a different register.

The next subsections explain in detail the central phases of the algorithm proposed.

3.1. Multiplier selection and binding

In order to obtain structured designs, excessive multiplication fragmentation should be avoided, and therefore some HW waste tolerated. For this purpose, the Maximum Internal Wastage Allowed (MIWA) parameter is introduced to quantify the maximum HW waste allowed by the designer in every design. Be the *Internal Wastage* (IW) of a multiplier in a cycle the percentage of bits discarded from the result in that cycle (due to the execution of one multiplication over a wider multiplier). Hence, the IW average of every multiplier in the datapath should not exceed MIWA. Its value ranges between:

- 0%. Every multiplier in the datapath must execute an operation of its same width in every cycle.
- 100%. No restriction applies.

Next some concept definitions follow in order to ease the understanding of this phase of the algorithm.

- *Multiplication order*: Be $m \geq n$ and $k \geq l$ then $m \times n > k \times l$ ($m \times n$ is bigger than $k \times l$) if ($m > k$) or ($m = k$ and $n > l$).
- *Occurrence* of width $m \times n$ in cycle c: number of $m \times n$ multiplications scheduled in cycle c.

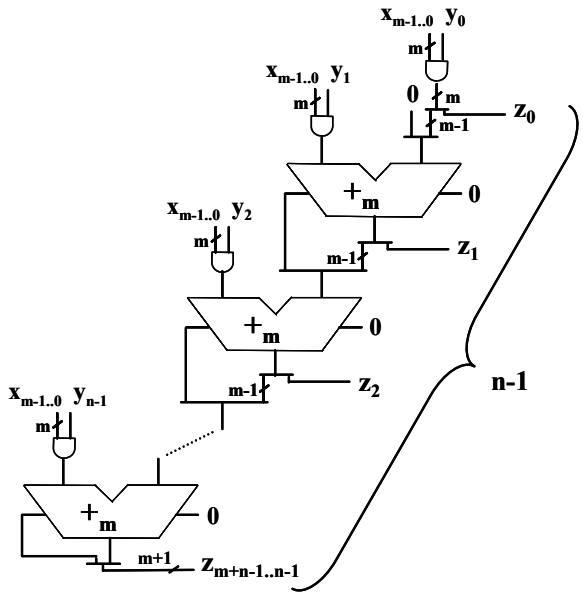

Fig. 2. Fragmentation of a $m \times n$ bits unsigned multiplication into additions.

- *Candidate*: set of operations formed by zero or one operation scheduled in every cycle. Of course, there are many different operation alignments of every candidate formed by operations of different widths. To reduce the algorithm complexity we have only taken into account the ones with the least significant bits of the operands aligned. Thus, if one operation is executed over a wider FU the most significant bits of the result produced are discarded.
- *Interconnection saving* of C candidate IS(C):

  IS(C) = BitsOpe(C) + Bits_Res(C) where

BitsOpe(C): number of bits of the C candidate left and right operands that may come from the same sources.

BitsRes(C): number of bits of the C candidate results that may be stored in the same register.

During this phase the algorithm only handles the yet unallocated multiplications. It consists of a loop that

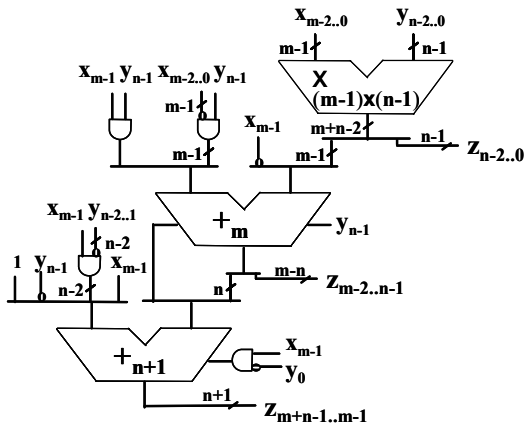

Fig. 3. Fragmentation of an $m \times n$ two complement signed multiplication into one unsigned multiplication and two additions.

finishes when either there are not remaining unallocated multiplications left, or when it is not possible to instance a new multiplier without exceeding MIWA parameter (due to the given scheduling). These conditions are checked in each loop iteration at the end of every step executed by the algorithm.

The first and second steps are performed twice during the first iteration, initially to instance and bind two-complement signed multipliers, and secondly, unsigned multipliers. Only during the first iteration, at the end of the second step every unallocated two-complement signed multiplications is transformed into one unsigned multiplication and two additions, as Fig. 3 shows. In the remaining iterations of the algorithm every step is performed only once. We detail now these steps.

Instancing and allocating multipliers without IW. For every different width $m\times n$ the algorithm instances as many multipliers of that width as the minimum occurrence of width $m\times n$ per cycle. Next, the algorithm allocates operations to them. For every instanced multiplier of width $m\times n$, it calculates the candidates formed by as many multiplications of width $m\times n$ as the circuit latency, and the IS of every candidate. The algorithm allocates to every multiplier the operations of the candidate with greater IS.

Every multiplier instanced in this step executes one operation of its same width per cycle, consequently its IW is zero in all cycles.

Instancing and allocating multipliers with some IW. For every different width $m\times n$, and from the biggest, the algorithm checks if it is possible to instance one $m\times n$ multiplier without exceeding MIWA parameter. During this checking the algorithm considers in every cycle, the operation (able to be executed over an $m\times n$ multiplier) that produces the least IW of an $m\times n$ multiplier. After every successful checking the algorithm instances one multiplier of the checked width, and allocates operations to it. Now the candidates are formed by as many operations as the number of cycles in which there is, at least, one operation that may be executed over an $m\times n$ multiplier. The width of the candidate operation scheduled in cycle c is that of the operation used in cycle c to perform the checking. Hence, each candidate has the same number of operations of equal width.

Once calculated all candidates, the algorithm computes their corresponding IS. Next, the operations of the candidate with the greatest IS are allocated.

Multipliers instanced in this step may be unused during several cycles, and may also be used to execute narrower operations. Nevertheless the IW average of these multipliers is always MIWA compliant.

Fragmenting multiplications. This step of the algorithm is only reached when it is not possible to instance a new multiplier of the same width as any of the yet unallocated multiplications (at this stage the only yet unallocated multiplications are unsigned ones) without exceeding MIWA parameter. In this step the algorithm selects an operation width $m\times n$ and a fragment width $k\times l$, and fragments some of the $m\times n$ multiplications into at least one $k\times l$ multiplication. These fragmentations increase the number of $k\times l$ multiplications, which may result in the final instance of a multiplier of that width (during steps one or two).

▪ *How are multiplications fragmented?, how many fragments are obtained?, which are the type and width of every fragment?*

There are many ways of fragmenting one unsigned multiplication into a set of narrower multiplications and additions. In particular, the number of different fragmentations of an $m\times n$ multiplication into five multiplications and four additions, obtaining at least one $k\times l$ fragment is $(m-k+1)\times(n-l+1)\times 16$ (being $m\geq k$ and $n\geq l$). The number of different fragmentations grows with the width of the operation to be fragmented, and decreases as the width of the operation fragment augments.

In the above formula $(m-k+1)\times(n-l+1)$ is the number of different ways of choosing a $k\times l$ multiplication fragment from an $m\times n$ multiplication. And for each there are 16 different fragmentations of the remaining part of the original multiplication. Fig. 4 a) shows the structure of a 7×7 multiplication and one of the 36 different ways of extracting a 2×2 multiplication from it. For every of these 36 possibilities there are 16 different ways of fragmenting the rest of the 7×7 multiplication, to produce the minimum number of fragments (9 operations). These 16 possibilities correspond to all different ways of selecting either a horizontal or oblique line in every vertex of the parallelogram occupied by the selected 2×2 multiplication. Fig. 4 b) shows some of these 16 possible fragmentations.

The calculus of all possible fragmentations requires exponential time, so our algorithm only takes into account a reduced set of them. This set is formed by the particular cases of the general method explained above, which fragment minimally the original multiplication. Fig. 5 shows the set of fragmentations considered. Each of these 8 ways fragments the original multiplication into a set of 3 multiplications and 2 additions.

▪ *How are the operations to be fragmented selected?, which is the fragment width?*

First the algorithm selects both the width of the operations to be fragmented and the fragment width, and afterwards a set of multiplications of the selected width, which are finally fragmented.

Multiplication fragmentation increases the number of narrower multiplications, which may result in the instance of a new multiplier. Thus the algorithm selects as the width of the operations to be fragmented, the width $m\times n$ of the biggest multiplication that satisfies the next two conditions:

a) There is at least one $k\times l$ multiplication, being $k\times l<m\times n$, that can be executed over an $m\times n$ multiplier (i.e. $m\geq k$ and $n\geq l$).

b) At least in one cycle there is one $m\times n$ multiplication scheduled and there are not $k\times l$ multiplications scheduled.

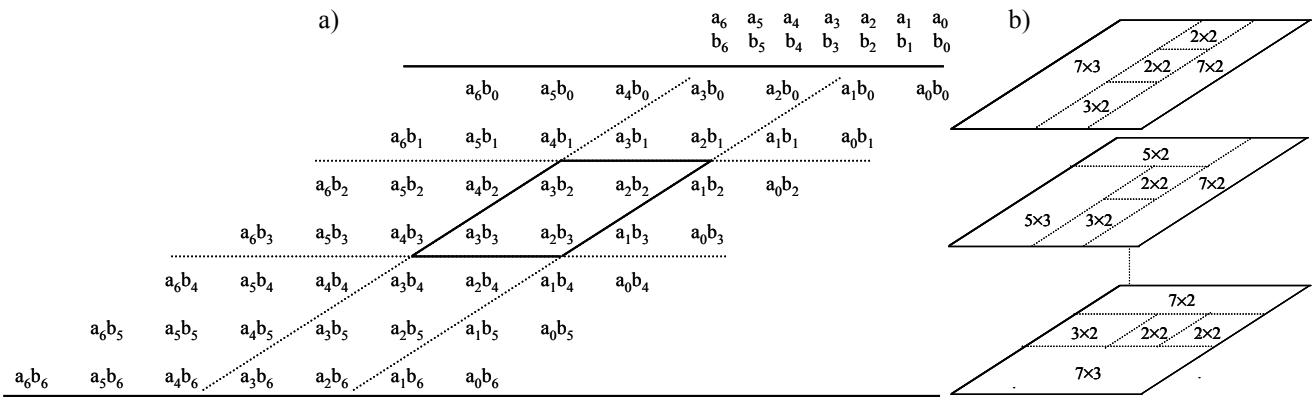

Fig. 4. a) Structure of a 7×7 unsigned multiplication, and one way of extracting a 2×2 multiplication fragment from it, b) 3 different ways (out of 16 possibilities) of fragmenting the rest of the 7×7 multiplication.

Once chosen the width $m \times n$ of the multiplications to be fragmented, the algorithm selects the fragment width $k \times l$ of the biggest multiplication satisfying the above conditions. Next the algorithm selects the set of the operations to be fragmented. This set is formed by one $m \times n$ multiplication per cycle in which there are not $k \times l$ multiplications scheduled.

If there are not two operations satisfying the above conditions among the yet unallocated multiplications, then the algorithm selects both two different widths as the widths of the operations to be fragmented, and a fragment width independent of the remaining unallocated multiplications. The widths selected as the widths of the operations to be fragmented $m \times n$ and $k \times l$, are those of the biggest multiplications which satisfy the next conditions:

a) $m \times n \neq k \times l$

b) At least in one cycle there is one $m \times n$ multiplication scheduled and there are not $k \times l$ multiplications scheduled.

c) At least in one cycle there is one $k \times l$ multiplication scheduled and there are not $m \times n$ multiplications scheduled.

In this case the fragment width equals the maximum common multiplicative kernel of $m \times n$ and $k \times l$ multiplications, i.e. $\min(m,k) \times \min(n,l)$. Now the set of operations to be fragmented is formed by either one $m \times n$ or one $k \times l$ multiplication per cycle. In those cycles in which there are operations of both widths scheduled, only one multiplication of the biggest width is selected.

Once selected the set of operations to be fragmented and the desired fragment width, there are 8 different ways of fragmentation (shown in Fig. 5). The algorithm selects one of them, following the next criteria:

- The best fragmentations are those which obtain in addition to one multiplication fragment of the desired width, other multiplication fragments of the same width as any of the yet unallocated multiplications.
- Among those fragmentations with identical multiplication fragments, the one that requires the smallest adder cost is preferable.

After fragmenting the selected operations the algorithm executes again steps one and two to instance, if possible, any multiplier of the same width as any of the new multiplications (resulting from the fragmentation).

At the end of this phase, if there still are unallocated multiplications (because due to the given scheduling it is not possible to instance a new multiplier without exceeding the MIWA parameter), these are transformed into additions as Fig. 2 shows.

3.2. Adders selection and binding

This phase performs jointly the adder selection and binding, guaranteeing the maximum bit-level reuse of these FUs. The allocated additions may come from three different sources: additions in the original specification, additive operation transformations, and multiplication fragmentations. The algorithm used to perform this phase is detailed in [10].

The datapaths obtained at the end of this phase have the following features:

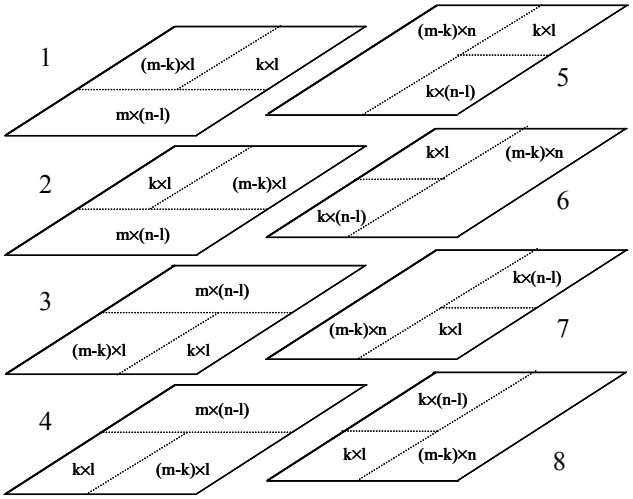

Fig. 5. Fragmentations of an $m \times n$ multiplication obtaining a $k \times l$ multiplication fragment.

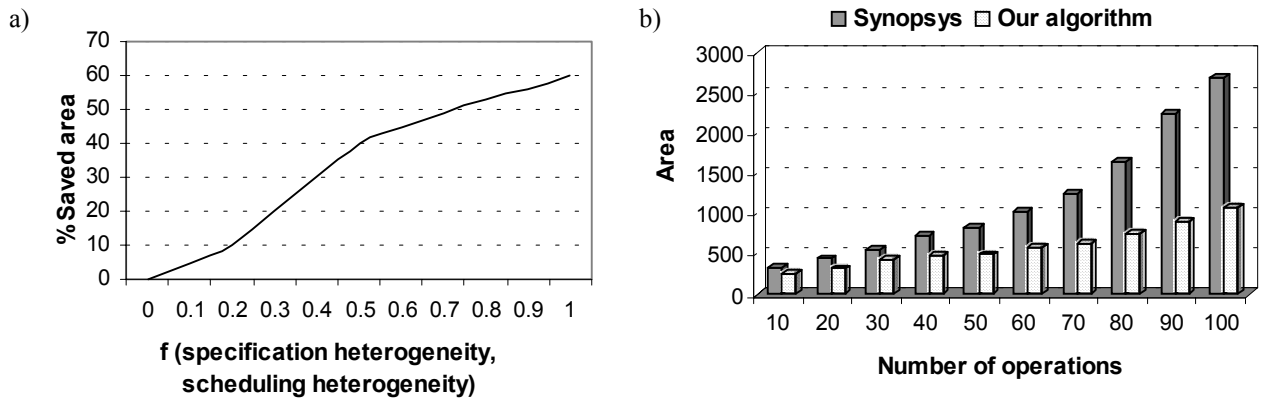

Fig. 6. a) Percentage of area saved by our algorithm in comparison with Synopsys Behavioral Compiler, b) average area of some implementations proposed by Synopsys and our algorithm.

- One addition may be executed over a wider adder.
- One addition may be executed over a set of narrower adders linked to propagate the carry signal.
- The sum of the adder widths is equal to the maximum number of bits added simultaneously in a cycle.
- The number and width of the adders are independent of the number and width of the additions allocated.

4. Experimental results

In order to measure the quality of the implementations obtained by our algorithm, they have been compared to those proposed by Synopsys Behavioral Compiler. We have synthesized a wide collection of *heterogeneous specifications* formed by multiplications and additive operations of different widths. Specifications sizes ranged from 10 to 100 operations (about 50% were multiplications), latencies varied from 5 to 50 cycles, and MIWA values up to 15%.

Results show that the areas of the implementations obtained by our approach are always smaller than the ones of the circuits proposed by the commercial tool. For the circuits synthesized, the average area saved by our approach is about 40%. The amount of saved area grows in general with both the *specification heterogeneity* (number of different widths of every different operation type in the specification divided by the number of operations) and the *scheduling heterogeneity* (measures the uniformity of the operations to cycles distribution) as it is shown in Fig. 6 a). Fig. 6 b) shows the average area of some implementations obtained by both Synopsys and our algorithm, grouped by the number of specification operations.

5. Conclusion

This paper presents a novel allocation algorithm especially suited for heterogeneous specifications and macro-cells technologies (due to the structured designs obtained). In order to reduce the HW waste produced by conventional algorithms, some operations are fragmented and executed over a set of linked FUs.

The implementations obtained are multiple-precision datapaths, where the number, type, and width of the HW resources used are independent of the circuit specification.

Experimental results show that the circuits synthesized using this allocation algorithm have smaller area than the implementations offered by commercial tools. The amount of area saved by our approach grows in general with both the specification and the scheduling heterogeneities.

References

[1] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens. "A methodology and design environment for DSP ASIC fixed point refinement". *Proc. DATE, 1999.*

[2] Y.N. Chang, and K.K. Parhi, "High-Performance Digit-Serial Complex-Number Multiplier-Accumulator". *Proc. ICCD, 1998*.

[3] H. Lee, and G.E. Sobelman. "FPGA-Based FIR Filters Using Digit-Serial Arithmetic". *Proc. Int. ASIC Conf., 1997.*

[4] C. Huang, Y. Chen, Y. Lin, and Y. Hsu. "Data path allocation based on bipartite weighted matching". *Proc. DAC, 1990.*

[5] K. Küçükçakar, and A. Parker. "Data Path tradeoffs using MABAL". *Proc. DAC, 1990.*

[6] M. Ercegovac, D. Kirovski, and M. Potkonjak. "Low-power behavioural synthesis optimization using multiple precision arithmetic". *Proc. DAC, 1999.*

[7] G.A. Constantinides, P.Y.K. Cheung, and W.Luk. "Heuristic datapath allocation for multiple wordlength systems". *Proc. DATE, 2001.*

[8] M.C. Molina, J.M. Mendías, and R. Hermida. "Bit-level Scheduling of Heterogeneous Behavioural Specifications". *Proc. ICCAD, 2002*.

[9] M.C. Molina, J.M. Mendías, and R. Hermida. "High-Level Synthesis of Multiple-Precision Circuits Independent of Data-Objects Length". *Proc. DAC, 2002*.

[10] M.C. Molina, J.M. Mendías, and R. Hermida. "Multiple-Precision Circuits Allocation Independent of Data-Objects Length". *Proc. DATE, 2002*.