Bit-level Scheduling of Heterogeneous Behavioural Specifications*

M.C. Molina, J.M. Mendías, R. Hermida
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid
Avda. Complutense s/n, 28040 Madrid (SPAIN)
{cmolinap, mendias, rhermida}@dacya.ucm.es

Abstract

This paper presents a heuristic scheduling algorithm for *heterogeneous specifications*, those formed by operations of different types and widths. The algorithm extracts the common operative kernel of the operations, and binds afterwards operations to cycles with the aim of distributing uniformly the number of bits calculated per cycle. In consequence, operations may be fragmented and executed during a set of non-necessarily consecutive cycles, and over a set of several linked simple hardware resources. The proposed algorithm, in combination with allocation algorithms able to guarantee bit-level reuse of hardware resources, obtains considerably smaller datapaths than the ones proposed by conventional synthesis algorithms. 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) algorithms try to obtain RT–level circuits with a trade-off between their *latency* (number of cycles needed to perform a computation), and the *cost* of their hardware (HW) resources (related to the maximum number of operations executed in a cycle). The synthesis is basically performed in two steps: *scheduling* (sets the cycle in which every operation starts its execution) and *allocation* (selects the HW resources in which every operation is executed and every argument is stored).

In order to obtain small circuits, conventional HLS algorithms try to balance the number of operations executed per cycle, and keep HW resources busy during most cycles. Nevertheless, in almost every cycle some *HW waste* (percentage of idle hardware resources) appears due to two main factors: the *operation mobility* (range of cycles in which every operation may start its execution, subject to data dependencies and given time constraints) and the *specification heterogeneity* (number of different operation types and widths occurring in the specification).

The *operation mobility* influences the HW waste because a limited mobility makes perfect distributions of operations among cycles impossible to reach. Even in the hypothetical case of specifications without data dependencies, some waste appears when the latency is not a divisor of the number of operations.

The *specification heterogeneity* influences the HW waste because HLS algorithms usually treat separately operations with different types or widths, preventing them from sharing the same HW resource. In consequence, a particular *mobility dependent HW waste* appears for every different (type, width) pair that occurs in the specification. This waste even appears when more efficient HLS algorithms are used. For example, many algorithms are able to allocate operations of different widths to the same functional unit (FU), but if an operation is executed over a wider FU (extending its arguments), the FU is partially wasted because some bits of the result are computed but not really needed.

On one hand, the *mobility dependent HW waste* could be reduced by balancing the number of bits calculated per cycle instead of the number of operations. In order to distribute uniformly the number of bits calculated, the operations should be fragmented.

On the other hand, the *heterogeneity dependent HW waste* could be reduced by synthesizing jointly 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 that in addition to the previous capabilities also fragment *compatible operations* into their common operative kernel plus some glue logic.

In both cases, each operation fragment inherits the mobility of the original operation and is scheduled separately. In consequence, one original operation may be executed during a set of non-necessarily consecutive cycles, starting in the earliest

* Supported by Spanish Government Grant CICYT TIC-99 0474

0-7803-7607-2/02/$17.00 ©2002 IEEE

one from the least significant bits, and over a set of linked HW resources. In the datapaths designed following these strategies the number, type, and width of the HW resources are in general independent of the number, type, and width of the specification operations.

To most directly present these design strategies, the example illustrated in Fig. 1 is used. Fig. 1a) shows a fragment of a data flow graph. Figs. 1b) and 1c) show the scheduling and the FU cost of the implementation proposed by respectively, a conventional algorithm and a more efficient one (proposed by our approach). Fig. 1d) shows, for the second scheduling, the set of FUs selected to execute every operation scheduled in the first cycle. Note that every multiplication is executed over a set of chained adders, and some of the additions over a set of adders linked to propagate the carry signals. For example, the 10×6 bit multiplication (P=L×M) is executed during the first and third cycles; in the first cycle the 10 bits addition and the first addition of this multiplication are chained, the partial result produced is stored until the third cycle in which the execution of the multiplication resumes.

In the next sections we present a scheduling algorithm able to balance the number of bits calculated per cycle, and that includes a pre-processing phase to extract the common operative kernel of specification operations. In combination with allocation algorithms able to allocate operations over several linked FUs [1], our approach obtains smaller circuits than the ones proposed by conventional algorithms.

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:

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 [2].

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*

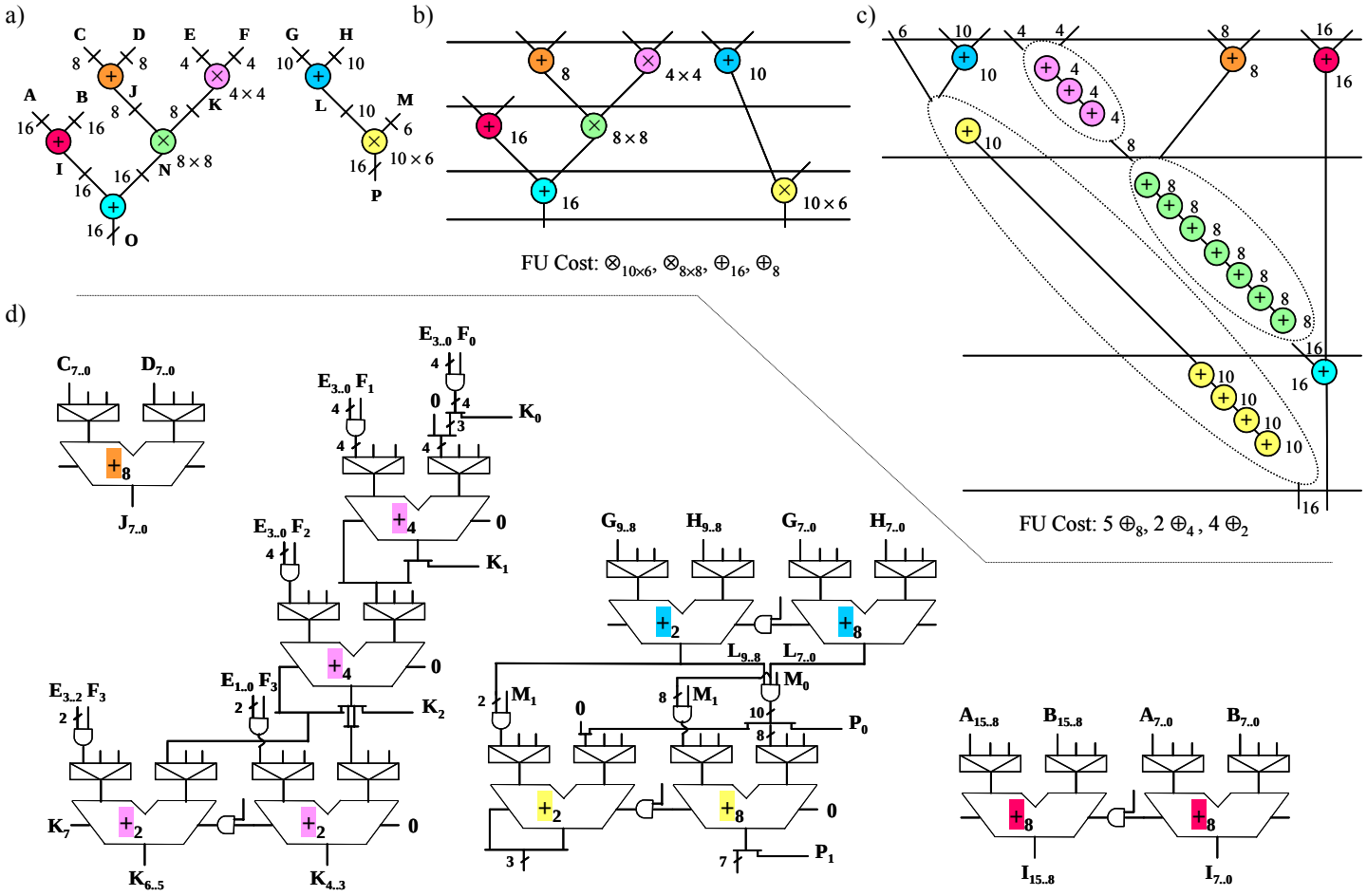

Fig. 1. a) Data flow graph, b) scheduling proposed by a conventional algorithm, c) more efficient scheduling, d) datapath obtained from the 2nd scheduling (allocation of the operations executed in the 1st cycle).

(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 operations whose widths allow the maximum bit-level reuse of HW resources [3][4].

c) *HLS of DSP algorithms.* Conventional scheduling algorithms synthesize *heterogeneous specifications* by balancing the number of operations of every different type executed per cycle, and binding operations to FUs of the same width [5][6]. More efficient algorithms allow the execution of operations over wider FUs [7][8]. And in order to obtain smaller datapaths some authors propose algorithms which perform the scheduling and allocation phases at the same time. In [8] the proposed algorithm involves an iterative refinement of operator word-length information, leading to a scheduled and bound data-flow graph. Some authors admit that these trivial solutions are not good enough and suggest (but not implement) other alternatives, like for example the execution of one operation during several cycles [7].

3. Proposed Algorithm

Our scheduling algorithm takes into account the following two features in order to obtain the maximum bit-level reuse of datapath HW resources:

- complex operations may be executed over simpler linked HW resources,
- any operation can be executed during a set of non-necessarily consecutive cycles (saving the partial results and carries produced), if data dependencies and given time constraints allow it.

The scheduling is performed in three phases: first the common operative kernel of *compatible specification operations* is extracted, next the algorithm performs a virtual scheduling of these new operations trying to balance the number of bits calculated per cycle, and finally a compaction phase produces the definitive operations to be allocated. Fig. 2 shows a schema of the algorithm.

3.1 Kernel Extraction

As it has been shown in the introduction, some operations have a common operative kernel whose extraction may improve HW reuse. Adders are the most versatile HW module, and most operations may be executed over a set of adders linked with some glue logic. This is why in this version of the algorithm the

```
BEGIN
AdditiveOperations2Additions(OP);
Unscheduled = OP;
Bound = PerfectDistribution(OP, λ);
REPEAT
   F = CalculateForces(Unscheduled);
   BestOC = SelectMinForce(F);
   IF Bound < (BitsSched(BestOC.cycle)+Width(BestOC.op))
   THEN
      Fragment(BestOC.op,  op1, op2);
      Schedule(op1, BestOC.cycle);
      Add(Unscheduled, op2);
      FOR ope IN (Suc(BestOC.op) ∪ Pred(BestOC.op)) DO
            Fragment(ope, op1, op2);
            Remove(Unscheduled, ope);
            Add(Unscheduled, op1);
            Add(Unscheduled, op2);
      END FOR;
   ELSE Schedule(BestOC.op, BestOC.cycle);
   END IF;
   Remove(Unscheduled, BestOC.op);
   UpdateDependencies(Unscheduled);
   IF UnreachableSolution(Bound,Unscheduled,BestOC.cycle);
   THEN
       Bound = Bound + Correct(Unscheduled, BestOC.cycle);
   END IF;
UNTIL Unscheduled = Ø;
CompactScheduling;
END;
```

Fig. 2. Algorithm to perform the scheduling of *heterogeneous specifications*.

pre-processing phase extracts the *additive* kernel of the specification operations. The operation types considered in this version are: multiplication, addition, subtraction, comparison, maximum, minimum, absolute value, code converters, and data limiters, for both two complement signed and unsigned operands.

For example, one comparison may be transformed into one subtraction, and one subtraction into one addition, thus one comparison may be transformed into one addition. These operation transformations are shown in Fig. 3.

An unsigned $m \times n$ bits multiplication (being $m \geq n$) is transformed into n-1 chained additions of m bits, as it is shown in Fig. 4. A two complement signed $m \times n$ bits multiplication (being $m \geq n$) is transformed into n-2 additions of m-1 bits, one addition of m bits, and one addition of n+1 bits, all of them chained as it is shown in Fig. 5. This structure corresponds to a Baugh and Wooley multiplier [9] with its partial results reorganized. We have selected this multiplier because its uniform structure, constructed entirely with conventional full adders, has a bigger additive kernel (and in consequence less glue logic) than other proposed implementations.

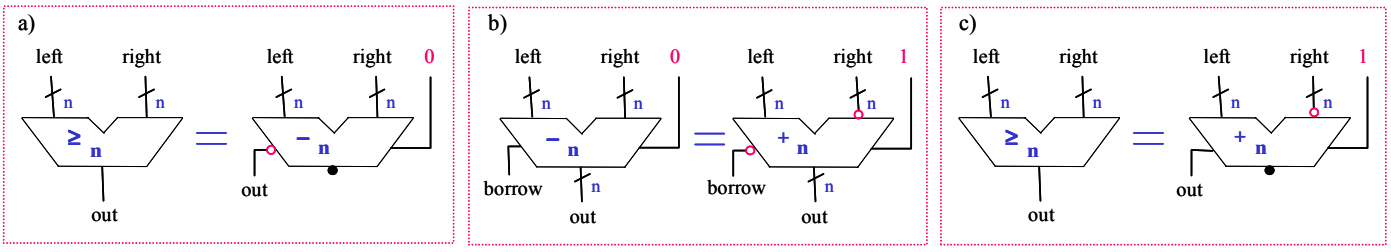

Fig. 3. a) Transformation of a comparison into a subtraction, b) subtraction into addition, c) comparison into addition.

3.2 Scheduling

This scheduling phase is a variant of the popular and classical *force-directed scheduling algorithm* [10]. The intent of the original method is to minimize HW cost subject to a given time constraint (λ) by balancing the number of operations executed per cycle. For every different type of operations the algorithm successively selects, among all operations and all execution cycles, an (operation, cycle) pair according to an estimation of the circuit cost called *force*.

By contrast, the intent of our variant is to minimize HW cost by balancing the number of bits calculated per cycle. Our method successively selects, among all operations (additions after *kernel extraction* phase) and all execution cycles, a pair formed by an operation (or an operation fragment) and an execution cycle, according to a new *force* definition that takes into account the operations widths.

In order to ease the understanding of this phase some definitions are introduced first:

- OP: set of specification operations.
- EOP_c (*estimated operations* in cycle c): set of operations whose mobility makes their scheduling possible in cycle c.

$$EOP_c = \left\{ op \in OP \,\middle|\, c \in \mu_{op} \right\}$$

where the mobility of the operation op is defined by:

$$\mu_{op} = \left\{ c \in N \,\middle|\, \sigma_{ASAP}(op) \le c \le \sigma_{ALAP}(op) \right\}$$

- SOP_c: set of operations scheduled in cycle c.

$$SOP_c = \left\{ op \in OP \,\middle|\, \sigma(op) = c \right\}$$

initially, $\bigcup_{c=1}^{\lambda} SOP_c = \phi$

and, at the end of the algorithm, $\forall c \in \{1,..,\lambda\}\ SOP_c = EOP_c$

- UOP_c: set of unscheduled operations whose mobility makes their scheduling possible in cycle c.

$$UOP_c = EOP_c - SOP_c$$

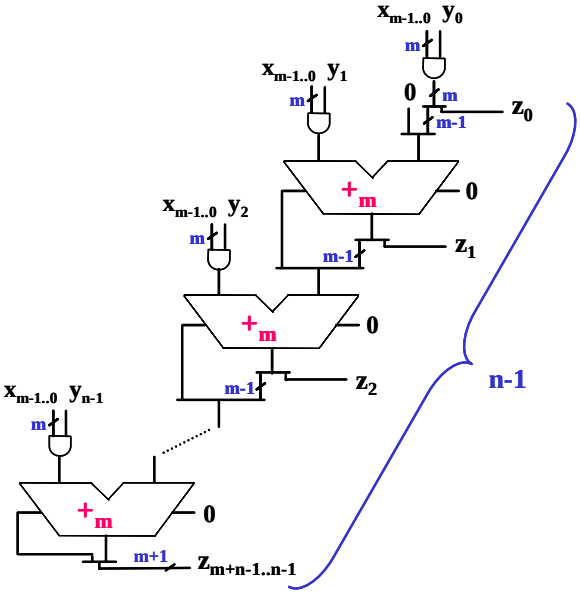

Fig. 4. Transformation of a *m*×*n* bits unsigned multiplication into additions.

initially, $\forall c \in \{1,..,\lambda\}\ UOP_c = EOP_c$

and, at the end of the algorithm, $\bigcup_{c=1}^{\lambda} UOP_c = \phi$

- P(*op*, *c*): probability of scheduling operation *op* in cycle *c* (assuming that all bindings to feasible execution cycles have equal probability). Operation *op* covers cycle *c* with probability P(*op*, *c*).

$$P(op,c) = \begin{cases} \dfrac{1}{\|\mu_{op}\|} & \text{if } c \in \mu_{op} \\ 0 & \text{otherwise} \end{cases}$$

- EOC(*c*) (*estimated operation cost* in cycle *c*): average value of the number of operation widths covering execution cycle *c*:

$$EOC(c) = \sum_{op \in EOP_c} \left(\text{width}(op) \cdot P(op,c) \right)$$

- BS(c): number of bits scheduled in cycle *c*.

$$BS(c) = \sum_{op \in SOP_c} \text{width}(op)$$

Once introduced these definitions, we are ready to define the *force* associated with an operation to cycle binding. It measures how desirable the binding is, considering that fixing the cycle in which an operation starts its execution possibly affects the mobility of data dependent operations, resulting in additional changes in the EOP_c set for all cycles. So the *force* associated with a certain operation to cycle binding is the addition of the *self force* of the binding and the *affected force* of all the successors and predecessors of the operation. The overall formula is:

$$F(op,c) = SF(op,c) + \sum_{op' \in Suc(op)} AF(\mu_{op'}^{new}) + \sum_{op' \in Pre(op)} AF(\mu_{op'}^{new})$$

where, $$SF(op,c) = EOC(c) - \sum_{i \in \mu_{op}} \frac{EOC(i)}{\|\mu_{op}\|}$$

and, $$AF(\mu_{op}^{new}) = \sum_{i \in \mu_{op}^{new}} \frac{EOC(i)}{\|\mu_{op}^{new}\|} - \sum_{i \in \mu_{op}^{init}} \frac{EOC(i)}{\|\mu_{op}^{init}\|}$$

being μ_{op}^{new} and μ_{op}^{init} the new and initial mobility of the affected operation.

The smaller or more negative the *force* expression is, the more desirable an operation to cycle binding becomes.

Additionally, our algorithm binds operations (or operation fragments) to cycles subject to an *adaptable bound*, whose initial value equals to:

$$\frac{\sum_{op \in OP} \text{width}(op)}{\lambda}$$

This value corresponds to the most uniform operation bits distribution among cycles, not including data dependencies. Because this perfect distribution is not always reachable, the *bound* is updated during the scheduling in the way explained below.

The scheduling algorithm consists in a loop which finishes when all the operations and all the operation fragments have been

scheduled. In every step a pair (*op*, *c*) formed by an operation and one of its mobility cycles is selected using the *force* measure. If the number of bits already scheduled in the selected cycle plus the width of the selected operation does not reach the *bound*, that is:

$$\mathrm{BS}(c) + \mathrm{width}(op) < bound$$

then, the operation is scheduled in the selected cycle. Otherwise the operation is fragmented into two new operations whose widths are:

$$width1 = bound - \mathrm{BS}(c)$$
$$width2 = \mathrm{width}(op) - width1$$

The first fragment is scheduled in the selected cycle, and the other one remains unscheduled with its original mobility. In order to avoid any reduction in the mobility of the predecessors and successors of the fragmented operation, they are also fragmented. These fragmentations divide the computation path into two new independent ones, in which the two fragments of a same operation may have different mobility. For example, Figs. 6a) and 6b) show respectively, a computation path and the mobility of its operations, and Figs. 6c) and 6d) show respectively, the resulting computation paths and the mobility of the new operations after scheduling an *x* bit fragment of operation *B* in cycle *i*.

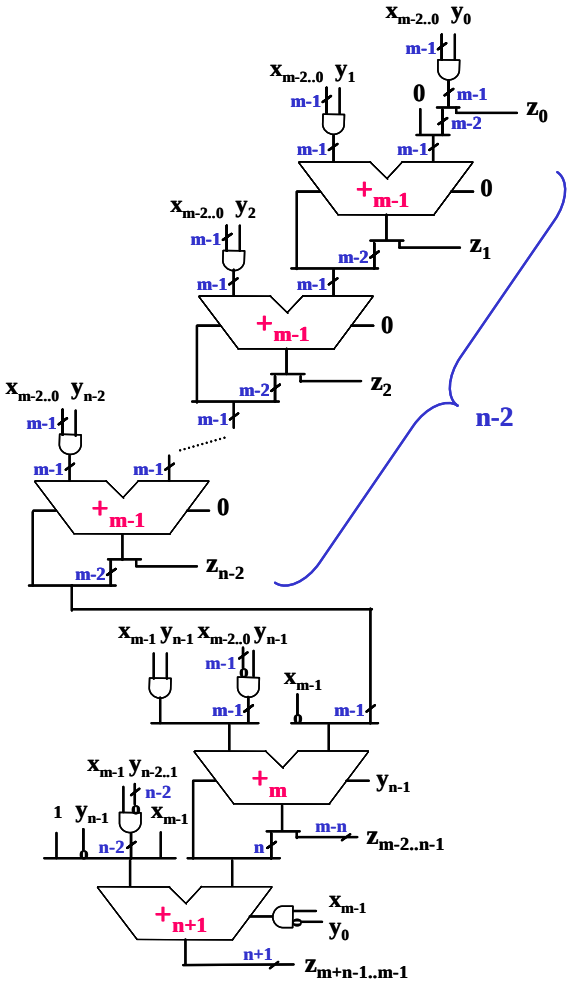

Fig. 5. Transformation of a *m*×*n* two complement signed multiplication into additions.

Once an operation (or a fragment) is scheduled in a cycle *c*, the algorithm checks if the distribution of operations among cycles, defined by the actual value of the *bound*, is still reachable. Otherwise the value of the *bound* is updated with the next reachable most uniform distribution. This occurs when:

a) The number of bits scheduled in cycle *c* does not reach the *bound* and there are no more operations which could be scheduled in it, either because the operations which originally could be scheduled in cycle *c* are already scheduled, or their mobility has changed, that is:

$$\left(\mathrm{BS}(c) < bound\right) \wedge \left(\mathrm{UOP}_c = \phi\right)$$

The new *bound* value is the old one plus the number of bits needed to reach the bound in cycle *c* divided by the number of *open cycles* (those included in the mobility of the unscheduled operations):

$$\frac{bound - \mathrm{BS}(c)}{\left\| \mathrm{OC} \right\|}$$
$$\text{where, } \mathrm{OC} = \left\{ c \in \mathrm{N} \mid \mathrm{UOP_c} \neq \phi \right\}$$

b) The number of bits scheduled in cycle *c* is equal to the *bound* and there is at least one unscheduled operation whose mobility includes cycle *c*, but even fragmented cannot be scheduled in its mobility cycles, that is:

$$\mathrm{BS(c)} = bound \wedge$$
$$\exists op \in \mathrm{UOP}_C \Big| \sum_{c \in \mu op} (bound - \mathrm{BS(c)}) < \mathrm{width}(op)$$

The new *bound* value is the old one plus, for every operation satisfying the above condition, the number of bits which cannot be scheduled divided by the number of cycles of its mobility:

$$\frac{\mathrm{width}(op) - \sum_{c \in \mu op} \left(bound - \mathrm{BS}(c)\right)}{\left\| \mu_{op} \right\|}$$

The proposed scheduling algorithm supports both chaining and multi-cycle features. Chaining (execution of several data-dependent operations in the same cycle) is implemented by extending the mobility of fast operations into the previous and/or next cycles (being the total propagation delay less than the clock cycle). In this case, in order to calculate the earliest/latest cycle in which an operation may start its execution, it is necessary to take into account the addition of the propagation delays of all the predecessor/successor operations that could be scheduled in the same cycle. Although multi-cycle feature (execution of an operation during a set of consecutive cycles) is rarely implemented with adders, it is included in the algorithm by extending the mobility of operations that require several cycles for execution. This extension requires a new method for calculating the contributions of multi-cycle operations to the *estimated operation cost*, EOC(*c*). For its calculation, every stage of multi-cycle operations is treated as an individual operation dependent on the other stages.

3.3 Compacting Operations

This phase produces the definitive set of operations to be allocated by joining those fragments of the same original operation scheduled in the same cycle.

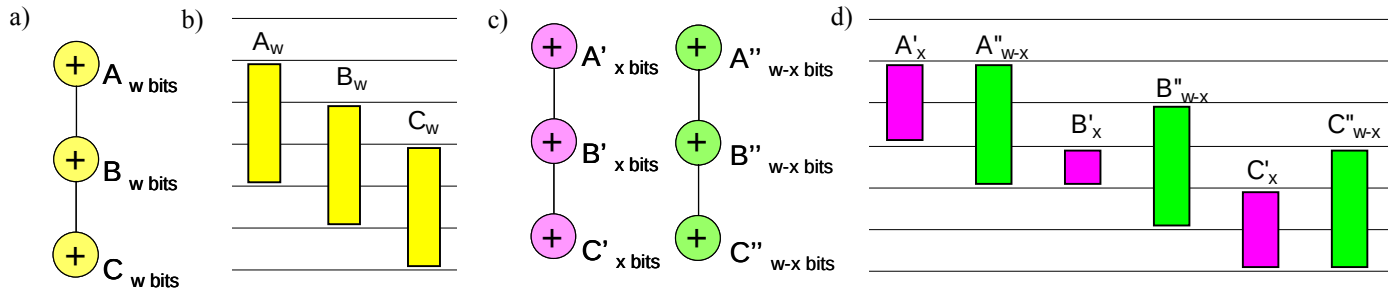

Fig. 6. a) Computation path, b) mobility of the operations, c) new independent computation paths after scheduling a fragment of operation *B*, d) mobility of the new operations.

When, as a result of the scheduling phase, an operation is executed in several cycles, the result is calculated by starting in the earliest cycle from the least significant bits, and storing its partial results until its latest execution cycle. In order to perform correct computations the partial carry signals of these operations need to be stored.

3.4 Allocation Requirements

In order to take advantage of those schedulings obtained by the proposed algorithm, we also need allocation algorithms able to implement the following design strategies: the execution of one addition over a wider adder, or over a set of narrower adders linked by some glue logic. One allocation algorithm including these features is proposed in [1].

4. Experimental Results

In this section we present the factors that influence the implementations obtained both by conventional algorithms and by our approach, and the experimental results carried out in order to measure the quality of the implementations proposed.

4.1 Implementations Quality Influences

The main difference between conventional synthesis algorithms and our approach is the number of factors that could influence the quality of the implementations obtained.

The implementations proposed by conventional algorithms depend on the specification size, the operation mobility, and the *specification heterogeneity*. Otherwise, our algorithm gets implementations totally independent from the *specification heterogeneity*, i.e. independent from the type, width and number of the operations used to describe behaviours.

To illustrate the influences presented above we have synthesized, using our scheduling algorithm in combination with the allocation one presented in [1], and Synopsys Behavioral Compiler, different descriptions of a specification, originally formed by 50 operations (20 of them were multiplications), and a latency equal to 10 cycles. Fig. 7 a) shows the amount of area saved by our approach in function of the *specification heterogeneity* (ratio of the number of different widths of every different operation type in the specification to the number of operations).

4.2 Experimental Saved Area

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 randomly generated *heterogeneous specifications*, formed by *compatible operations* (operations with an additive kernel listed in section 3.1) of different widths. Specifications sizes ranged from 10 to 100 operations (about 40% were multiplications), and latencies varied from 4 to 30 cycles. 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 65%. Fig. 7 b) shows the average area of the implementations obtained both by Synopsys and our algorithm, grouped by the number of specification operations.

5. Limitations and Future Work

The solution proposed is well suited for *cell based technologies* (FPGAs, standard cells, etc) but could not be appropriate for *macro-cells* ones, because the exhaustive transformation of operations into additions could waste the benefits of a structured design (regularity, locality, etc). In these cases, selective transformations are needed, and a certain HW waste should be tolerated. For example, if it is possible to obtain a distribution of operations to cycles in which an equal number of operations of a same type and width is scheduled in most cycles, then those operations should not be transformed into additions.

In order to obtain structured datapaths, the scheduling of operations of every different type and width should be performed separately, except for additions. So there should be as many scheduling processes as the number of different operation types and widths in the specification.

Because complex operations may be transformed into a set of simpler ones, the scheduling should be performed beginning with the most complex operations and finishing with the simplest ones (additions). And also, for every operation type the wider ones should be scheduled first, because wider operations may be transformed into a set of narrower ones.

If after any of the scheduling processes a *good* distribution (some HW waste may be tolerated) of operations to cycles were not reached, then those operations which unbalance the

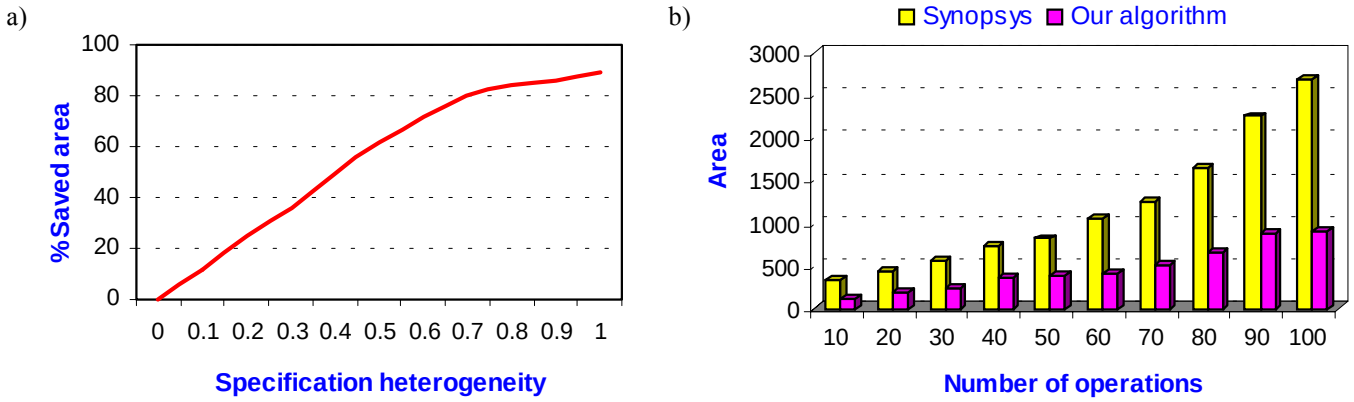

Fig. 7. a) Percentage of area saved by our algorithm in comparison with Synopsys Behavioral Compiler, b) average area of some implementations proposed by Synopsys and our algorithm.

distribution should be transformed into a set of simpler ones to be scheduled afterwards.

For example, an $m \times n$ bits two complement signed multiplication (being $m \geq n$) could be transformed into one $(m\text{-}1)\times(n\text{-}1)$ bit unsigned multiplication and 2 additions of m and $n+1$ bits, as it is shown in Fig. 8. And an $m \times n$ bit unsigned multiplication (being $m \geq n$) could be transformed into r unsigned multiplications whose dimensions were: $m \times k_1$, $m \times k_2$, …, and $m \times k_r$ bits (being $k_1 + k_2 + \ldots + k_r = n$), and r-1 additions of $m+k_1$, $m+k_2$,…, $m+k_{r-1}$ bits.

6. Conclusion

This paper presents a scheduling algorithm especially suited for *heterogeneous specifications* and *cell based technologies*. In order to reduce the HW waste produced by conventional algorithms, it takes advantage of the common operative kernel extraction of *compatible operations*. And in order to increase the bit-level reuse of HW resources, it balances the number of bits calculated per cycle by fragmenting specification operations.

Experimental results show that circuits synthesized using this scheduling algorithm in combination with allocation algorithms which satisfy the requirements presented in subsection 3.4, have smaller area than the implementations offered by commercial tools. The amount of area saved by our approach grows in general with the *specification heterogeneity*.

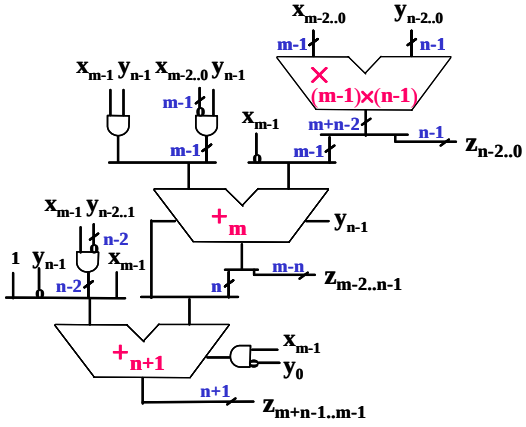

Fig. 8. Transformation of a $m \times n$ two complement signed multiplication into one multiplication and two additions.

7. References

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

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

[3] Y.N. Chang, and K.K. Parhi, "High-Performance Digit-Serial Complex-Number Multiplier-Accumulator". In Proceedings of ICCD, 1998.

[4] H. Lee, and G.E. Sobelman. "FPGA-Based FIR Filters Using Digit-Serial Arithmetic". In Proceedings of the International ASIC Conference, 1997.

[5] C. Huang, Y. Chen, Y. Lin, and Y. Hsu. "Data path allocation based on bipartite weighted matching". In Proceedings of DAC, 1990.

[6] K. Küçükçakar, and A. Parker. "Data Path tradeoffs using MABAL". In Proceedings of DAC, 1990.

[7] M. Ercegovac, D. Kirovski, and M. Potkonjak. "Low-power behavioural synthesis optimization using multiple precision arithmetic". In Proceedings of DAC, 1999.

[8] G.A. Constantinides, P.Y.K. Cheung, and W.Luk. "Heuristic datapath allocation for multiple wordlength systems". In Proceedings of DATE, 2001.

[9] C.R. Baugh, and B.A. Wooley. "A two's Complement Parallel Array Multiplication Algorithm". IEEE Transactions on Computers, 1973.

[10] P.G. Paulin, and J.P. Knight. "Force-Directed Scheduling for the Behavioral Synthesis of ASICS". IEEE Transactions on CAD, 1989.