Area Constraint Propagation in High Level Synthesis

R. Nane, V.M. Sima, K. Bertels
Computer Engineering Lab
Delft University of Technology
Delft, The Netherlands
Email: ⟨r.nane; v.m.sima; k.l.m.bertels⟩@tudelft.nl

***Abstract*—Hardware compilers which generate hardware descriptions from high-level languages are rapidly gaining in popularity. These generated descriptions are used to obtain fast implementations of software/hardware solutions in heterogeneous computing platforms. However, to obtain optimal solutions under certain platform constraints, we need intelligent hardware compilers that choose proper values for the different design parameters automatically. In this paper, we present a two-step algorithm to optimize the performance for different area constraints. The design parameters under investigation are the maximum unroll factor and the optimal allocation of resource types. Experimental results show that generated solutions are mapped into the available area at an occupancy rate between 74% and 99%. Furthermore, these solutions provide the best execution time when compared to the other solutions that satisfy the same area constraint. Finally, a reduction in design time of 42x on average can be achieved when these parameters are chosen by the compiler compared to manually selecting them.**

***Keywords*-FPGA, DWARV 2.0, HLS, unroll, area constraint**

I. INTRODUCTION

Heterogeneous multicore architectures are a direct consequence of the end of Moore's law. In many cases, this heterogeneity is being implemented by means of FPGA based custom computing units. The Xilinx Zynq at the embedded side and the Convey HC-1 on the supercomputing side are just a couple of the more telling examples. The FPGA blades allow to provide application specific hardware support which can even be modified at runtime and thus provide a tailored support for different application domains. The gain that can be obtained by combining a traditional processor with a Reconfigurable Architecture (RA) can be tremendous (e.g. between 20x and 100x [1]). However, before the potential of this technology can be fully exploited, a number of challenges have to be addressed. One of the challenges is the automatic generation of the hardware units through e.g. C-to-VHDL generation, while a second important challenge is to have an efficient way to explore the design space. This paper primarily focuses on the second challenge.

The strength of RAs is that they offer much more design freedom than a General Purpose Processor (GPP). In this work we rely on such architectures to maximize the application performance by automatically exploiting the available parallelism subject to area constraints. In particular, we look at application loops in more detail as these constructs provide a greater source of performance improvement, also in hardware synthesis. Considering the scenario where Hardware Description Language (HDL) code is automatically generated, two important parameters have to be explored namely, the degree of parallelism (i.e. the loop unrolling factor) and the number of functional modules used to implement the source High Level Language (HLL) code. Determining without any human intervention these parameters is a key factor in building efficient HLL-to-HDL compilers and implicitly any Design Space Exploration (DSE) tools.

This paper presents an optimization algorithm to compute the above parameters automatically. This optimization is added as an extension to the DWARV 2.0 hardware compiler [2] which generates synthesizable VHDL on the basis of C-code. The contributions of this paper are:

- The automatic determination of the maximum unroll factor to achieve the highest degree of parallelism subject to the available area and the function characteristics.
- The automatic computation of the number of functional units instantiated by the compiler, to optimize the performance given the previously identified unroll factor and respecting the given design constraints.
- The validation of the algorithm through an implementation on an operational platform.

The rest of the paper is organized as follows. Section II presents the background and related research. In Section III the details of the algorithm are presented, while in Section IV the experimental results are discussed. Finally, Section V draws the conclusions and highlights future research activities.

II. BACKGROUND AND RELATED WORK

The MOLEN Machine Organization [5] is an architecture developed at TU DELFT that facilitates Hardware/Software (HW/SW) co-design. It includes three main components, a GPP, Custom Computing Unit (CCU) used as an accelerator and a shared memory between them. The CCUs are implemented on a reconfigurable (FPGA based) platform. In order to create an accelerator for this platform, we use DWARV 2.0, a C-to-VHDL compiler, that generates for an application kernel a CCU. More information about DWARV 2.0 will be given below. The CCU generated complies with a simple

978-1-4673-2845-6/12/$31.00

interface that contains ports to enable the exchange of data and control information. This allows changing the CCU while the system is running, without modifying the hardware design, thus allowing multiple applications and CCUs to execute at the same time. Given enough resources, multiple CCUs can be executed in parallel, taking advantage of inherent application parallelism. To manage the reconfigurable area we divide it (logically) into slots, in which one CCU can be mapped. Each slot can be used by a different application. However, these slots can be combined to allow different sized kernels to be mapped corresponding to different design goals. For example, possible layouts include 5 CCUs that each use an equal area or 2 CCU, having an area ratio of 3/2. Having only one slot using all the available area is another possible scenario.

DWARV 2.0 is a C-to-VHDL hardware compiler [2] built with CoSy Compilers Framework [8]. Compilers built with CoSy are composed of a set of *engines* which work on the Intermediate Representation (IR) generated based on the input program. The initial IR is generated by the front-end. To generate VHDL from C code, DWARV 2.0 performs standard and custom transformations to the combined Control Data Flow Graph (CDFG). The ROCCC project [4] aims at the parallelization and acceleration of loops. Catapult-C [9] and CtoS [10] are commercial high-level synthesis tools that take as input ANSI C/C++ and SystemC inputs and generate register transfer level (RTL) code. However, these compilers are complex and require extensive designer input in both the applied optimizations and the actual mapping process, making it less viable for a software designer that does not have in-depth hardware knowledge. Both Altium's C to Hardware (CHC) [11] and LegUp [3] compilers are intended to compile annotated functions that belong to the application's computational intensive parts in a HW/SW co-design environment.

However, none of these compilers, including DWARV 2.0, currently possesses any capabilities to generate hardware that satisfies a particular area constraint, while maximizing performance. More precisely, requiring that a function's generated HDL takes no more than a given area is not possible. Neither the unroll factor, nor the number of functional units used are determined taking into account the execution time or the area. Performing this optimization automatically would enable high-level tool-chains to analyse different application mappings in a shorter amount of time. For example, the algorithm presented in [7], where the best mapping of CCUs is selected given a fixed number of resources and fixed kernel implementations, would be improved if the implementations would be generated according to some determined area constraint.

III. AREA CONSTRAINED HARDWARE GENERATION

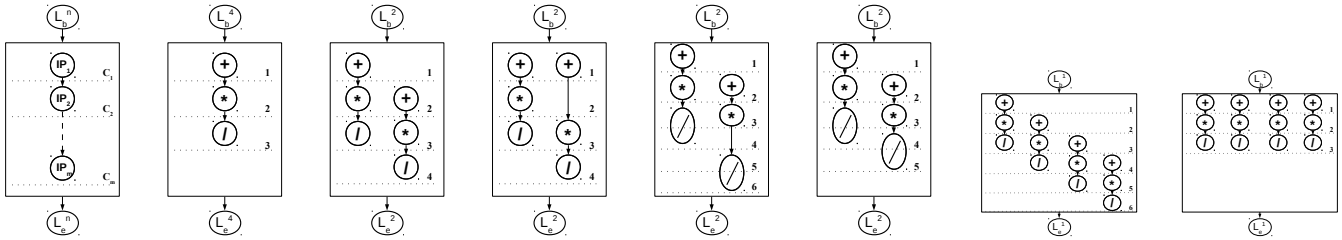
In this section, we present the model that allows the compiler to decide the unroll factor and the number of resources. In the first part we describe and define the problem, while in the second part we elaborate on the details of the algorithm. Finally, we conclude the section by showing how this model has been integrated in the DWARV 2.0 compiler.

A. Motivational Example and Problem Definition

To describe the problem, we make use of a synthetic case study. We consider a simple function that transforms each element of an input array based on simple arithmetic operations with predefined weights. The function has no loop-carried dependencies as each array element is processed independently of the others. Figure 1(b) shows this graphically, where L_b marks the beginning of the loop (i.e., compute the new value for each of the array elements) and the number four in the superscript represents the total number of loop iterations (i.e., we use an input array size of four elements). The body of the loop is delimited by the rectangle box. Furthermore, we see that in this body three operations are performed, each taking one cycle for a total function execution time of 12 cycle time (C_T). Unrolling once and given there are no loop-carried dependencies, we can speed up the application by a factor of two if we would double the resources. However, the overall speedup will depend on the available hardware resources and is thus constrained by this. For instance, if we use only one resource of each type as in the initial case, the speed-up would be less than the maximum possible ($C_T = 8$ in Figure 1(c)). The L_e marks the end of the four loop iterations.

Doubling only one resource type, as for example the addition unit (Figure 1(d)), and keeping the initial assumption that each of the three computations takes the same number of clock cycles (i.e., one clock cycle), does not decrease the execution time while increasing the area. This is a suboptimal allocation of resources for the achieved execution time. It is important to note that this scenario can falsely allow one to draw the conclusion that for such loop body types, the number of resources that should be used for each type is the number corresponding to the resource with the minimum count in the loop. However, this is true only for the case where all resources compute the output in the same number of cycles. However, if this is not the case which is a fair assumption for real world applications, having different counts of resources is possible without obtaining a suboptimal resource allocation. This is illustrated in Figures 1(e) and 1(f). In the first illustration, we have the scenario when one resource of each type is used, while in the second only the number of division units is doubled. This leads to a decrease in C_T from 12 to 10 because the number of cycles for one loop iteration is decreased by 1 due to the availability of a second division unit that can be used before the first division finished execution.

Finally, fully unrolling the loop and using only one resource of each type achieves yet a better execution time. This is illustrated in Figure 1(g) where C_T has been reduced to 6 cycles. Nevertheless, the best execution time (C_T = 3) is achieved when fully unrolling the loop and using the maximum possible units for each operation as shown in Figure 1(h). However, this can increase the area past the available area for the function. This is specially important in our scenario, where the reconfigurable area is divided among differently sized slots. For example, in the current implementation of the MOLEN machine organisation, we have available a maximum



(a) $C_T = C_m n$ (b) $C_T = 12$ (c) $C_T = 8$ (d) $C_T = 8$ (e) $C_T = 12$ (f) $C_T = 10$ (g) $C_T = 6$ (h) $C_T = 3$

Fig. 1. Motivational Examples: a) Formal Representation; b) No Unroll and 1+, 1*, 1/ units; c) 2 Unroll and 1+, 1*, 1/ units; d) 2 Unroll and 2+, 1*, 1/ units; e) 2 Unroll and 1+, 1*, 1/ units; f) 2 Unroll and 1+, 1*, 2/ units; g) 4 Unroll and 1+, 1*, 1/ units; h) 4 Unroll and 4+, 4*, 4/ units;

of 5 slots. Given runtime and partial reconfigurability, the slots can be merged or split depending on the application's requirements. This would lead to different area requirements for one kernel. Therefore, it is necessary to have hardware compilers that generate automatically hardware designs that map onto the available area given as an input constraint. This would avoid iterating over the design choices to find the optimal unroll factor and the maximum number of resources of each type, thus reducing the design space exploration time.

Summarizing, there is a trade-off between the number of unrolls one can do and the number of resources used for each unroll. The goal of our model is to compute the best combination given performance as the main design objective. Given the general form of applications with loop body types as shown in Figure 1(a), we define the problem as follows: be the loop delimited by L_b^n and L_e^n iterating n times, performing operations that use m different hardware modules, IP_1 to IP_m, for which we know the sizes and latencies c_1 to c_m respectively, determine the *unroll factor* and the maximum number of modules for each of the *m* IP types such that the input area constraint is satisfied while the performance is maximized.

B. Optimisation Algorithm

The algorithm consists of two parts, one for each parameter that needs to be determined. In the first step, we determine the unroll factor based on the available FPGA area as well as the area increase due to wiring when more parallelism is available after an unroll is performed. To obtain the unroll factor (uf) we solve inequality (1) for the maximum value of uf:

$$A_i + uf * wi <= A_t \tag{1}$$

,where A_i, A_t and wi represent the initial area of the kernel, the total area available for the kernel on the target FPGA and the wiring increase per unroll, respectively. Furthermore, we have $uf \in N$ and have obtained both A_i and wi after a complete compilation and estimation of the kernel's generated HDL. The initial kernel refers to the code 'just as is', i.e., the code without any unroll optimization applied and using the minimum number of resources for each of the required IPs necessary to implement its functionality. The compiler is executed once without activating any specific optimization regarding the number of IPs and unroll. The compiler is executed a second time with the loop unrolled once. Then, the wi is the difference between the estimated area of these two generated kernels. The estimation is based on [6].

In the second step, we determine the component counts necessary to fit the hardware design onto the FPGA area available. This step assumes the IP sizes are available and can be loaded from an external library. However, if these are not available, the netlist of the IP should be synthesized for the specific device and the number of slices required for it should be saved in the external database. Furthermore, the available parallelism for each IP has been fixed by the previous step which unrolled the loop body. That means that no more than some value n of IPs of type m can execute in parallel. The second constraint according to the problem definition involves the area of the IPs itself, which leads to the constraint that the sum of all IP sizes multiplied by the number of their instantiations should not exceed the total area given as a parameter. Finally, the objective is to minimize the time it takes to compute each level in the graph by instantiating as many resources of that type possible. This is expressed as minimizing the number of cycles the slowest level in the CDFG takes. Furthermore, the candidate solutions are selected from those which satisfy the area constraint. Note that minimizing the number of cycles only for a level in the CDFG would be ineffective if the other levels with different operations are not minimized as well. The complete set of equations is shown in (2):

$$\begin{gathered} min : MAX\{countIP_i/x_i + countIP_i\%x_i?1:0\} \\ \sum_{i=1}^{m} x_i * area\{IP_i\} <= A_t \\ x_i <= t_{IP_i} * uf \quad (2) \\ countIP_i = MAX\{t_{IP_i} * uf\} \\ x_i \in N, \forall i \in \{1, ..., m\} \end{gathered}$$

,where *area* is the area of the component accounting for both the number of slices and dsps it requires. x_is are the variables which represent how many IPs of type *i* can be used inside the total area available (A_t). Furthermore, t_{IP_i} represents how many instances of type IP_i are used in the initial version of the code when no unroll has been performed.

C. Integration in DWARV 2.0

Figure 2 shows how the compiler flow has been extended to compute the predefined algorithm values and how the

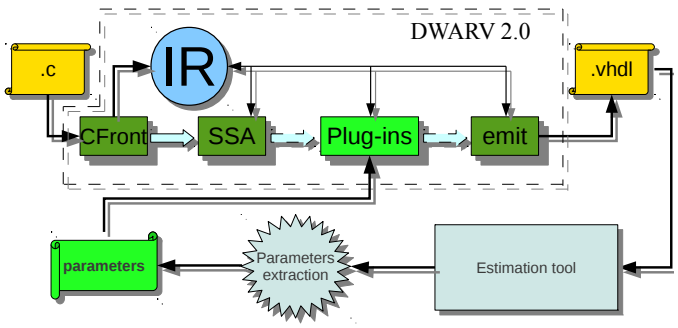


Fig. 2. Algorithm Integration with DWARV 2.0 Compiler.

algorithm has been integrated in DWARV 2.0. In the upper stripped box we see the internals of the DWARV 2.0 compiler as it was described in section II. This is composed of a series of loosely integrated engines, executing different optimizations and transformations on the globally available IR. The small engine box denoted *Plug-ins* represents the implementation of our algorithm, which performs no action if there are no input parameters, i.e. for the first two runs to obtain the parameters for the initial area and the area of the kernel with the loop unrolled once. The difference between these estimated areas gives us the wiring increase for unrolling.

After these preliminary compilations (maximum two), the algorithm can be applied. Using the computed wiring increase and the initial area obtained, the unroll factor can be computed by solving inequality (1). The solution is feed into the unroll engine which obtains a new intermediary version of the code with an increased parallelism. Note that the unroll engine is encompassed by the "Plug-ins" engine which was built around it, i.e., it is composed actually of two smaller engines corresponding to the two steps. One it is run before the unroll engine, the other after. However, due to space reasons we show this chaining as one engine in Figure 2. Finally, based on the determined parallelism, the inequalities in (2) are solved.

IV. EXPERIMENTAL RESULTS

In this section, we describe the experimental environment, the test cases and provide a discussion of the obtained solutions for the different design options available.

A. Experimental Environment

The environment used for the experiments is composed of three main parts: i) The C-to-VHDL DWARV 2.0 compiler [2], extended with an optimization engine applying the two-step algorithm described in the previous section, ii) the Xilinx ISE 12.2 synthesis tools, and iii) the Xilinx Virtex5 ML510 development board. This board contains a Virtex 5 xc5vfx130t FPGA consisting of 20,480 slices and 2 PowerPC processors. From these, 9600 slices are allocated to the reconfigurable part of the MOLEN design as presented in section II, and constitute the maximum area that DWARV 2.0 generated designs target. More precisely, we use in the experiments 1920, 2880, 4800 and 9600 slices corresponding to 20%, 30%, 50% and respectively the full area of the reconfigurable part to test the capability of the algorithm to generate designs that during synthesis will fit within these pre-defined area constraints.

B. Test Cases

To validate the correctness and usefulness of the algorithm, we used three real case studies. These are a simple vector summation of 128 elements, a 10x2 with a 2x10 matrix multiplication and a 5-tap-FIR filter computing 27 values. The vector summation contains 64 additions in parallel, the matrix multiplication iterates 100 times to compute each element of the 10x10 resulting matrix by doing two parallel multiplications followed by an addition, whereas the FIR test case consists of 5 parallel multiplications and 2 parallel additions for each computed element. All the arithmetic operations are done on floating points numbers, therefore the IPs of interest in the experiments are floating point adders and multipliers. However, the general approach described in the previous section can apply to any IP block, not just floating point adders and multipliers.

C. Discussion

To gain insight in the results and understand why the design process benefits from such an optimization, we look at the generated hardware under various area constraints, different unroll factors and number of IP cores. That is, we analyse the Pareto points obtained for different configurations. Due to space reasons, we discuss only the matrix multiplication example and give for the other two only the final results. This function consists of a loop iterating 100 times to compute each of the elements of the resulting 10x10 matrix. Each iteration contains one addition and two parallel multiplications. Fully unrolling this code would lead to a maximum of 100 additions and 200 multiplications to execute in parallel. Clearly this can easily lead to a HDL implementation that would not fit into the available area. Therefore, we perform an analysis with the DWARV 2.0 compiler extended with the optimisation engine and investigate its capability to generate CCUs that fit into different sized slots available in the current MOLEN design.

We begin by constraining the available area for the CCU to the smallest size slot that has 1920 slices accounting for 20% of the available reconfigurable FPGA area. Figure 3 shows different points corresponding to different configurations of the matrix function. To explain the points in the graph, we define the tuple $\langle x, y*, z+\rangle$ notation which represents the solution point with the loop body unrolled x times, instantiating y multipliers and z adders. In the graphs, the first element of the tuple is represented by a different shape and color. For example, the most left side point in the figure is $\langle 1, 2*, 1+\rangle$ denoting the implementation of the initial code (i.e., no unroll) and using two multiplication and one addition units. This implementation executes in 20030 ns and occupies 299 slices. Note that the implementation using the minimum number of resources, i.e., $\langle 1, 1*, 1+\rangle$ is slower (21030 ns) and occupies 29 slices more. The execution time is bigger because using only one multiplication core we do not take full advantage of the available parallelism, whereas the increase in area is due

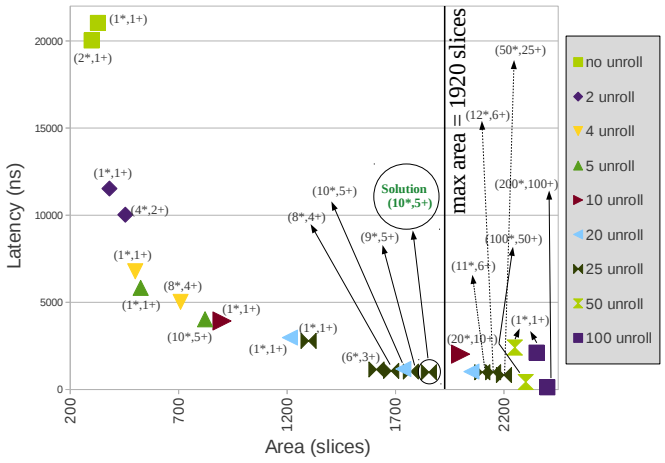


Fig. 3. Matrix multiplication: 20% area design constraint.

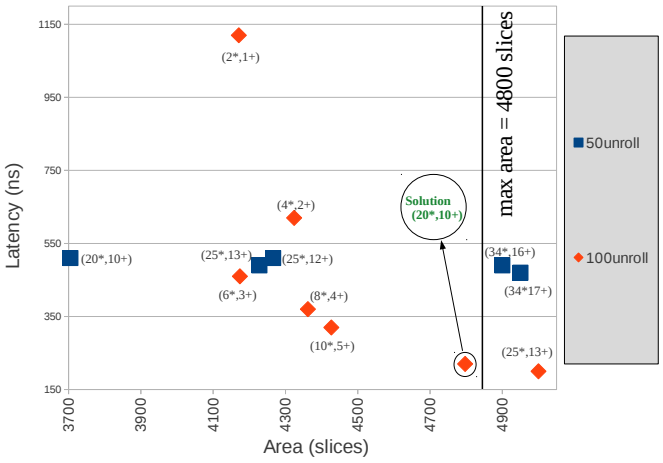


Fig. 5. Matrix multiplication: 50% area design constraint.

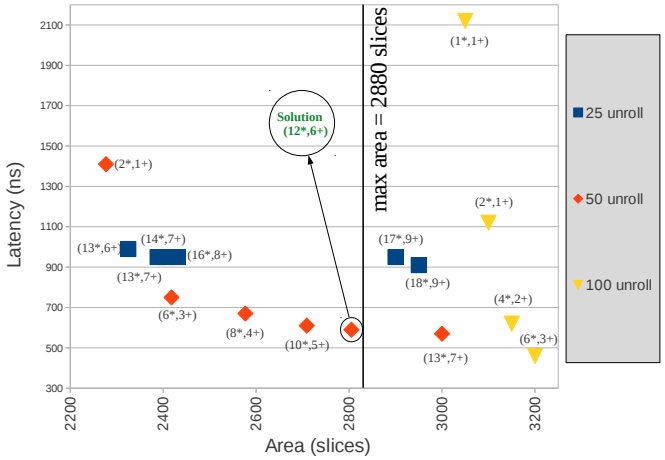


Fig. 4. Matrix multiplication: 30% area design constraint.

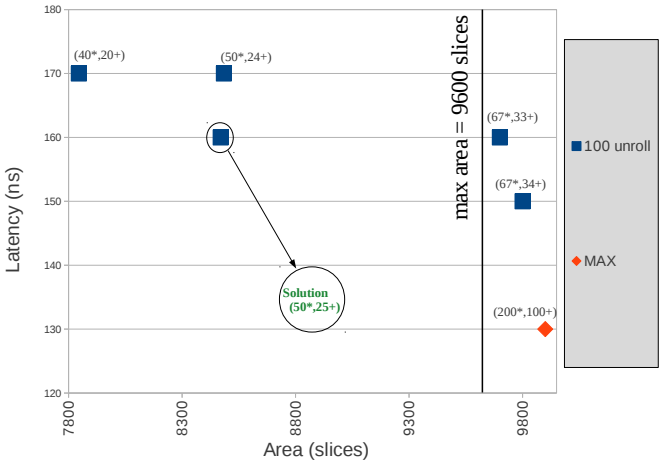


Fig. 6. Matrix multiplication: 100% area design constraint.

to the more slices required to route multiple inputs to one unit compared to the increase obtained by duplication.

The vertical line on the right of Figure 3, denoted by *max area*, represents the threshold after which the generated configurations will not fit into the requested area constraint. The fastest solution obtained by fully unrolling the loop and using the maximum number of cores for both operations, i.e. $\langle 100, 200*, 100+\rangle$, does not meet the design constraint of 1920 slices being situated on the right of the threshold line. The rest of the points show representative design configurations. That is, for each possible unroll factor we show the minimum (i.e., using one core of each unit) and the maximum implementations. However, for each unroll we can have more configurations than the minimum and maximum, for example the $\langle 20, 10*, 5+\rangle$ and $\langle 25, 8*, 4+\rangle$ solutions, with the second being faster and smaller.

Analysing all these solutions manually is very time consuming, therefore the benefit of having the compiler perform this automatically has tremendous effects in terms of saved design time. Considering the 20% area constrained matrix example and assuming a binary search through the design space, we would need to evaluate at least seven designs manually to obtain the $\langle 25, 10*, 5+\rangle$ optimal solution. Assuming 30 minutes to obtain the bit stream for each implementation, we would need at least 210 minutes to arrive at the optimal solution. The automation of this search and running the compiler along with the estimator takes on average five minutes. This leads to an average speed-up in design time of 42x.

Figure 4 illustrates the design points for the 30% area constraint. For this experiment, the figure shows only configurations after the loop body was unrolled by at least a factor of 25 because from the previous experiment with the smaller area constraint we know that the design choices up to 25 unroll factor cannot be the optimal solution. Therefore, we highlight several possible implementations for the 25, 50 and full unroll factors. Analysing the performance for these implementation we observe that unrolling has a bigger impact on performance than using more parallelism with a smaller unroll factor. For example, the $\langle 50, 6*, 3+\rangle$ kernel implementation is faster and occupies less area than the one

TABLE I
EXPERIMENTAL RESULTS OF THE TEST CASES AND THEIR CORRESPONDING SOLUTIONS FOR DIFFERENT AREA DESIGN CONSTRAINTS.

Function	Case	Uf	Max IPs	Solution	Area (slices)	Occupancy (%)	Latency (ns)	Freq. (MHz)	Power (mW)
VectorSum	20%	N/A	64+	10+	1888	98	925	213	106
	30%		64+	22+	2817	97	885	240	126
	50%		64+	64+	4526	94	865	338	294
	100%		64+	64+	5581	78	865	338	294
Matrix	20%	25	50*, 25+	10*, 5+	1854	96	990	300	95
	30%	50	100*, 50+	12*, 6+	2805	97	590	283	140
	50%	100	200*,100+	20*,10+	4797	99	220	280	324
	100%	100	200*,100+	50*,25+	8470	88	160	280	403
FIR	20%	14	70*,28+	8*, 3+	1897	98	1020	250	106
	30%	14	20*,28+	14*, 7+	2738	95	900	325	155
	50%	28	140*,56+	28*,12+	4738	98	500	339	245
	100%	28	140*,56+	47*,19+	7174	74	480	327	293

represented by the $\langle 25, 18*, 9+\rangle$ point. However, if we unroll too much we might not obtain any valid solutions as is the case with all the 100 unroll points. Therefore, finding the optimum unroll factor is a key step in the area constraint driven HDL generation. The optimal solution obtained for the 30% area constraint is the $\langle 50, 12*, 6+\rangle$ implementation point.

Finally, Figures 5 and 6 illustrate design points for the 50% and 100% area constraint, respectively. The solutions obtained are close to the minimum latency achieved when all 200 multiplications and 100 additions are used in the fully unrolled version. However, because of the high number of arithmetic units required by this implementation, and given the maximum number of slices available in the current setup, this best solution cannot be achieved. Therefore, a compiler that cannot handle area constraints and always fully unroll loops to achieve the highest code parallelism using the maximum possible resources, will always fail to give a valid solution.

The same experiments were subsequently performed for the VectorSum and FIR case studies as well. The results are summarized in Table I. The third column shows the unroll factor obtained in the first step of the algorithm, while the next column gives the maximum operations of each type that could be executed in parallel for the previous unroll factor. The solution of the second step of the algorithm for how many instances to use for each operation is shown in the fifth column. The next two columns highlight the number of slices for the obtained solution as well as the occupancy rate for the given area constraint (column two). Kernel latency and frequency information is shown in columns eight and nine. Last, power metrics for the solutions are given. The dynamic power consumed by the unoptimized, base kernel implementations is 81 mW, 8 mW, and 38mW for the vector summation, the matrix and the FIR function respectively.

V. CONCLUSION AND FUTURE RESEARCH

In this paper, we presented an optimization algorithm to compute the optimal unroll factor and the optimum allocation of resources during the HLL-to-HDL generation process when this is subject to area constraints. The described algorithm was added to an existing C-to-VHDL hardware compiler and three case studies were used to validate the optimisation. The experiments showed that the generated solutions are mapped into the available area at an occupancy rate between 74% and 99%. Furthermore, these solutions provide the best execution time when compared to the other solutions that satisfy the same area constraint. Furthermore, a reduction in design time of 42x on average can be achieved when these parameters are chosen automatically by the compiler.

Future research includes analysing other applications and investigating how different graph characteristics influence the optimisation presented. Another model extension involves dealing with variable loop bounds. In addition, more accurate prediction models for the wiring increase as well as for the power consumption are needed.

ACKNOWLEDGEMENT

This research is partially supported by the Artemisia iFEST project (grant 100203), the Artemisia SMECY project (grant 100230), and the FP7 Reflect project (grant 248976).

REFERENCES

[1] Z. Guo, W. Najjar, F. Vahid and K. Vissers. *A quantitative analysis of the speedup factors of FPGAs over processors*. Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays (FPGA '04): 162 - 170 .

[2] R. Nane, V.M. Sima, B. Olivier, R. Meeuws, Y. Yankova and K. Bertels. *DWARV 2.0: A CoSy-based C-to-VHDL Hardware Compiler*. To Appear in Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL '12).

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown and T. Czajkowski. *LegUp: high-level synthesis for FPGA-based processor/accelerator systems*. Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11): 33–36.

[4] Z. Guo, B. Buyukkurt, W. Najjar and K. Vissers. *Optimized Generation of Data-Path from C Codes for FPGAs*, Proceedings of the conference on Design, Automation and Test in Europe - Volume 1 (DATE '05): 112–117.

[5] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov and E. M. Panainte. *The molen polymorphic processor*. In IEEE Transactions on Computers(November 2004). pages: 1363-1375.

[6] R. J. Meeuws, C. Galuzzi and K.L.M. Bertels. *High Level Quantitative Hardware Prediction Modelling using Statistical methods*. In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Models, and Simulations (SAMOS '11): 140-149.

[7] V.M. Sima, E.M. Panainte, K. Bertels. *Resource allocation algorithm and OpenMP extensions for parallel execution on a heterogeneous reconfigurable platform*, Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL '08): 651-654.

[8] Associated Compiler Experts ACE: *CoSy compiler platform*. [Online]. Available: www.ace.nl

[9] Catapult C Synthesis Overview. [Online]. Available: http://www.mentor.com/esl/catapult/overview

[10] C-to-Silicon Compiler. [Online]. Available: http://www.cadence.com/Community/tags/CTOS/default.aspx

[11] Altium Designer 10. [Online]. Available: http://www.altium.com/