A Technique for FPGA Synthesis Driven by Automatic Source Code Analysis and Transformations

Beniamino Di Martino[1], Nicola Mazzocca[1], Giacinto Paolo Saggese[2], and Antonio G.M. Strollo[2]

[1] Department Of Information Engineering,
Second University of Naples, Via Roma 29, 81031 Aversa (CE), Italy
[2] Department of Electronics and Telecommunications Engineering,
University of Naples "Federico II", Via Claudio 21, 80125 Napoli, Italy
{beniamino.dimartino, n.mazzocca, saggese, astrollo}@unina.it

Abstract. This paper presents a technique for automatic synthesis of high-performance FPGA-based computing machines from C language source code. It exploits data-parallelism present in source code, and its approach is based on hardware application of techniques for automatic loop transformations, mainly designed in the area of optimizing compilers for parallel and vector computers. Performance aspects are considered in early stage of design, before low-level synthesis process, through a transformation-intensive branch-and-bound approach, that searches design space exploring area-performance tradeoffs. Furthermore optimizations are applied at architectural level, thus achieving higher benefits with respect to gate-level optimizations, also by means of a library of hardware blocks implementing arithmetic and functional primitives.
Application of the technique to partial and complete unrolling of a Successive Over-Relaxation code is presented, with results in terms of effectiveness of area-delay estimation, and speed-up for the generated circuit, ranging from 5 and 30 on a Virtex-E 2000-6 with respect to a Intel Pentium 3 1GHz.

1 Introduction

In the last years reconfigurable computing systems, based on FPGAs or other programmable hardware, is gaining more and more interest. This is due to their capability of mapping algorithm execution of compute-intensive calculations to reconfigurable hardware, thus achieving speed-ups up to 3 order of magnitude faster than commercial processors, and performance typical of an ASIC implementation. FPGA Custom Computing Machines (FCCMs) have demonstrated the potential for achieving extremely high performance for many data-intensive tasks, such as data encryption/decryption [9], automatic target recognition [3], string pattern matching and morphological skeletonization [2]. Although significant performance speed-ups for many applications are achieved, main issues with FCCMs are the complexity of their design and the difficulty of integrating use of

M. Glesner, P. Zipf, and M. Renovell (Eds.): FPL 2002, LNCS 2438, pp. 47–58, 2002.

reconfigurable co-processors into traditional PC-based systems. Manual circuit description is a powerful method for the production of high-quality circuit designs, but it requires time effort, expertise in digital systems design and a deep knowledge of the particular reconfigurable system employed. Furthermore it is often hard to highlight and exploit all parallelism present in target applications. We devise in these problems a possible limiting factor for adoption of reconfigurable hardware. In this paper we present the *SUNthesis* (Second University of Naples synthesis) project. Main goal of SUNthesis project is to design and develop an automatic compilation framework, that accepts C language programs and automatically outputs various synthesizable VHDL description of the system, optimized for a FPGA architecture, mainly in terms of throughput. From this description it is straightforward to generate the configuration bitstream for target FPGA, by using commercial synthesis tools. The framework shall in addition automatically interface reconfigurable hardware with PCs, by means of suitable wrapping C functions that take care of communication with the target reconfigurable processor. In this way it is possible to provide the user with hardware acceleration capability, transparently and without any knowledge of the reconfigurable platform employed.

SUNthesis framework exploits results of software optimizing compiler techniques, and is based on a transformation-intensive branch-and-bound methodology, that permits to highlight design trade-offs between area and throughput. SUNthesis VHDL code generator extracts an RTL description that exploits FPGA logic blocks target architecture (for instance multiplexers with many inputs, that are area-consuming, are inferred by means of tristate buffers). Hardware block library contains different implementations of arithmetic operators, described as bit-length parametrized VHDL blocks: for example adders with many operands, are synthesized using carry-save technique, and are specified for actual number and width of operands. SUNthesis can provide high-performance designs by using optimizations at architectural level, usually not employed by gate-level synthesizer: for instance, operators are arranged when possible in tree-like structure (instead of series of blocks) and signals arriving last are assigned to fastest propagation block inputs, by using information derived from critical-paths analysis.

Although synthesis techniques based on repeated and extensive application of data-flow graph transformations were demonstrated in the past to be very time consuming, our synthesis technique is rather fast, because estimation of area-time performances is conducted at an early stage of the design process, before any time consuming process of low level synthesis; this is possible by means of the adoption of a strict interaction with a hardware component library, characterized by propagation delay and silicon area requirement. Pipeline vectorization [2] shares with SUNthesis project some aspects: it uses C language, it copes with automatic parallelization of loops, but considers only inner loops that can be completely unrolled for parallel execution in hardware, in order to build deeply pipelined circuits. Outer loops may not have different iterations executing simultaneously. Any loop re-ordering transformation is left to designer.

In proposed framework of [2] is well-faced also the automatic software/hardware partitioning problem. Cameron project [13] is the closest to our approach, but it uses a language specifically designed for hardware description, named SA-C. The adoption of a dedicated language makes difficult real co-design methodology.

The paper proceeds as follows. In the following section the synthesis technique that exploits application of unrolling and tiling transformations is described. Section 3 shows the application of the technique to a case study, a Successive OverRelaxation code excerpt. In section 5 numerical results are resported and conclusions on the validity of the technique are drawn.

2 The Synthesis Technique

In this paper we focus attention on managing of data-parallelism included in loop nests. It is well known in literature that, in scientific and engineering programs, major fraction of execution time is usually spent in very small parts of the code corresponding to loops. Furthermore loops, especially regular and iterative computations, are likely to achieve high speed-up when implemented on an hardware processors. So our technique is focused on loop analysis and optimisation, and uses as basic building block loop nests. Central problem regards the hardware implementation of a loop nest, using a given number of hardware resources (in FPGA case, we intend CLBs as hardware resources), with the objective of minimizing loop execution time. Proposed technique can cope with perfect loop nest (Fig. 1), which loop bounds can be expressed as linear function of outermost indexes, and which dependence vectors can be expressed as distance vectors[4][7]. These hypothesis are sufficiently general to cover a number of actual programs.

1. $for\ (i_1 = L_1; i_1 < U_1; i_1 + +)$
2. $\quad for\ (i_2 = L_2; i_2 < U_2; i_2 + +)\ \cdots$
3. $\quad\quad for\ (i_N = L_N; i_N < U_N; i_N + +)\ \{$
4. $\quad\quad\quad S_I :\ a[f_1(i_1, \ldots, i_N), \ldots, f_M(i_1, , i_N)] = \cdots;$
5. $\quad\quad\quad S_J :\ \cdots = a[g_1(i_1, \ldots, i_N), \ldots, g_M(i_1, \ldots, i_N)];\ \}$

Fig. 1. Perfect N-nested loop model in pseudo-C code.

The choice of transformations to apply, after their legality verification, is driven by purpose of maximizing some cost-function, that is in our case maximize performances, guaranteeing that synthesized circuit can be hosted on target FPGA.

In literature regarding optimizing compiler for multiprocessors, different approaches are presented: some use heuristic search, others as [1] propose a unified framework for unimodular transformations. We combine both, using transformations to expose all available parallelism, and then evaluating effects of loop transformations that exploit parallelism (tiling, unrolling), in order to occupy

available area and maximize throughput. Enough accurate estimate of performance parameters, before any gate-level synthesis process, is possible for two reasons: the former is the internal representation of the code that directly corresponds to the synthesized data-path, the latter is the utilization of synthesizable off-the-shelf blocks available into hardware characterized in terms of area and propagation delay.

We intend to present an algorithm to exploit most profitable one between coarse and fine grain parallelism. We use a method similar to [1] in order to detect sequence of unimodular transformations that enable and make useful tiling and wavefront, partial and complete unrolling. By using skewing, it is possible to transform an N-nested loop with lexicographically positive distance vectors in a fully permutable loop (Th. 3.1 in [1]). A nest of N fully permutable loops can be transformed to code containing at least N-1 degrees of parallelism [10], when no dependences are carried by these loops, it can be exposed a degree of parallelism N, otherwise, N-1 parallel loops can be obtained. In the following we consider application of transformations in order to expose finest or coarsest granularity of parallelism making reference to simple algebraic case, and then generalizing.

Finest Granularity of parallelism: finest granularity of parallelism corresponds to a situation where innermost loops are parallel, while outermost loops are constrained to be executed sequentially because of loop-carried dependences [4]. From a fully permutable loop, skewing the innermost loop of the fully permutable nest by each of the remaining loops and moving it to the outermost position (Th. B.2 in [1]), it is possible to place the maximum number of parallelizable loops in the innermost loops, maximizing fine-grain parallelism. This situation involves in hardware presence of counters spanning dimensions of iteration space associated to outermost loops, and possibility to make a complete unrolling (or complete tiling if legal [1]) of innermost loops, allocating multiple active instances of loop body. Supposing that available hardware resources are A_{avail} (number of CLBs), the number of outermost sequential loops are K, area required for one instance of loop body is A_{body} and T_{body} is its propagation delay, then we can calculate approximatively A_{tot}, total area required and T_{elab} total execution time as:

$$\begin{cases} T_{elab} = (U_1 - L_1) \cdot (U_2 - L_2) \cdot \ldots \cdot (U_K - L_K) \cdot (T_{body} + T_Q + T_{set-up}) \\ A_{tot} = (U_{K+1} - L_{K+1}) \cdot \ldots \cdot (U_N - L_N) \cdot A_{body} \end{cases}$$

where T_Q and T_{set-up} are respectively delay and set-up time of registers, needed when results calculated by the loop body are stored in registers. So it is possible to do a *complete* unrolling, exploiting parallelization and exhausting available parallelism, only when $A_{tot} \leq A_{avail}$. In this case, if necessary, it is possible to use other resources to partial unroll sequential loops, at the cost of remarkable increase of area, but with quite performance improvement, by reducing impact of $T_Q + T_{set-up}$ on T_{elab}. Note that we have referred simple case in which loop bounds are constant, and only in this case it is possible use simple expression of T_{elab} reported. However if the loop bounds are linear (as we supposed) it is possible to resort symbolic sum technique [12] or use efficient numerical tech-

nique like [11]. If resources required for a complete unrolling exceeds available resources, a *partial* unrolling, or a partial tiling if legal, can be applied. Problem in this case is choice of unrolling/tiling factors $f_{K+1} \dots f_N$ for parallel loops, and can be formalized in following minimization problem if loop bounds are constant:

$$\begin{cases} Minimize\ T_{elab} = (U_1 - L_1) \cdot \ldots \cdot (U_K - L_K) \cdot \lceil (U_{K+1} - L_{K+1})/f_{K+1} \rceil \cdot \\ \qquad\qquad \ldots \cdot \lceil (U_N - L_N)/f_N \rceil \cdot (T_{body} + T_Q + T_{set-up}) \\ A_{tot} = f_{K+1} \cdot \ldots \cdot f_N \cdot A_{body} \leq A_{avail} \\ 1 \leq f_{K+1} < (U_{K+1} - L_{K+1}), 1 \leq f_N < (U_N - L_N) \end{cases}$$

In general situation when number of clock ticks are computed with [12] or [11], T_{elab} is function of $f_{K+1} \dots f_N$, and this problem can be solved using an exhaustive enumeration technique with cuts.

Coarsest Granularity of parallelism: coarsest granularity of parallelism corresponds to a situation where outermost loops are parallel, while innermost loops are constrained to be executed accordingly to loop-carried dependences. Theorem and algorithm B.4 in [1] describes an effective method to detect a series of transformations producing the maximal number of parallel loop as outermost loops. The problem can be modeled in a similar way of previous case, and we will describe method to obtain system to solve with an example. In Fig. 2, a 3-nested perfect loop is reported:

1. $for(i = 0; j < N; i++)$
2. $\quad for(j = 0; j < P; j++)$
3. $\quad\quad for(k = 0; k < M; k++)\ \ a[i][j][k] = f(a[i][j][k-1]);$

Fig. 2. A simple example of application of proposed technique for choice of unrolling factor.

Distance vector is (0,0,1), and so two outermost loops are parallel, while innermost loop has to be serially executed considering loop-carried dependence on loop k with level equal to 1. It is possible to (completely or partially) unroll and parallelize outermost loops, and unroll innermost loop. When considering unrolling of loop body, propagation delay is difficult to analytically estimate, because of the composition of critical paths. In this case it is essential to exploit a critical path analysis of data-flow graph resulting from unrolled loop. In case study section, we prove that our technique of area-time estimation, based on analysis of data-flow graph of loop body, is effective and gives results enough close to actual results obtained through synthesis process. Note that only when unrolling factor is greater than level of loop-carried dependences, dependences are moved into unrolled body of the loop, requiring a data-dependence analysis to be correctly treated (see parallel version of case study). In the case of Fig. 2 applying unrolling (or tiling that is legal in this case) to outermost loops, and unrolling innermost, we obtain the subsequent optimization problem:

$$\begin{cases} Minimize\ T_{elab} = \lceil N/f_1 \rceil \cdot \lceil P/f_2 \rceil \cdot (T_{body}(f_3) + T_Q + T_{set-up}) \\ A_{tot} = f_1 \cdot f_2 \cdot A_{body}(f_3) \leq A_{avail} \\ 1 \leq f_1 < N,\ 1 \leq f_2 < P,\ 1 \leq f_3 < K \end{cases}$$

Values of A_{body} and T_{body} are function of unrolling factor f_3 and can be considered as upper bounds respectively $f_3 \cdot A_f$ and $f_3 \cdot t_f$, where A_f and t_f are area and delay of block implementing function f. These inequalities can become equalities when no circuit optimization is possible on data-flow graph obtained from unrolling. If input code corresponds to one reported to Fig. 3 implementing matrix multiplication algorithm between $\underline{A}$ and $\underline{B}$, effects of possible optimization on unrolled data-flow graph is clear, by means of the application of transformation exploiting associativity and commutativity properties of addition. In Fig. 4 we

1. $for(i = 0; j < N; i++)\ \ for(j = 0; j < P; j++)\{$
2. $\quad temp = 0;$
3. $\quad for(k = 0; k < M; k++)\ \ S:\ temp+ = a[i][k] \cdot b[k][j];$
4. $\quad c[i][j] = temp;$

Fig. 3. C-code for Matrix multiplication.

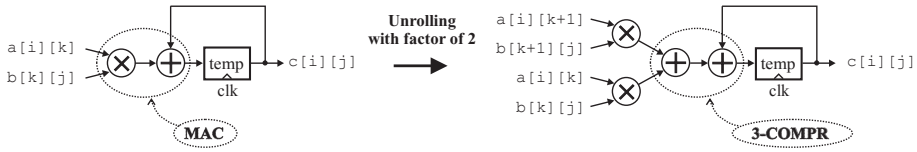

Fig. 4. Data-flow graph generation after application of unrolling with factor 2 on the loop body of matrix multiplication algorithm.

report data-flow graph for loop body S of code in Fig. 3. Using library of hardware components it is possible to implement multiplication and addition with a MAC (multiply-and-accumulate) block. Furthermore, when loop on k is unrolled twice, few other architectural level optimisations can be developed (in Fig. 4 use of 3-compressor instead of cascade of two adders), and so analysis of data-flow graph is essential in order to estimate area requirement and critical path.

3 Application of the Technique to a Case Study

In this section we describe the unrolling transformation and its effects on a case study, in order to prove effectiveness of our technique of area-delay evaluation from data-flow graph. As case study we propose the algorithm known as Successive Over Relaxation (SOR), because, although this is a not straightforward

parallelizable algorithm [5], it is a typical example where proper choice and application of sequences of unimodular loop transformations can provide optimal speedups on different target architectures [6]. SOR is a method for the numerical resolution of systems of linear equations, i.e. finding a solution to the vector equation $\widetilde{A} \cdot \underline{x} = \underline{b}$, where $\widetilde{A}$ is the $N \times N$ matrix of linear equation coefficients, and $\underline{x}$ and $\underline{b}$ are N-dimensional vectors. In non-simulaneous methods, value of the generic j-th variable at the (k+1)-th iteration $x_j(k+1)$ is updated in sequence, using the newly obtained j-1 values of the preceding variables for step k+1 and the "old" N-j values of the remaining variables from step k. One of these methods is Successive Over Relaxation:

$$x_j^{(k\ 1)} \quad \frac{}{a_{j,j}} \quad b_j \quad {}^{j\ 1}_{i\ 1} a_{j,i} x_i^{(k\ 1)} \quad {}^{N}_{i\ j\ 1} a_{j,i} x_i^{(k\ 1)} \quad (\quad 1) x_j^{(k)}$$

When the matrix $\widetilde{A}$ is a $(2 \cdot s + 1)$-diagonal matrix $(0 \leq s \leq N)$, computational complexity can be improved by modifying the update expression to:

$$x_j^{(k\ 1)} \quad \frac{}{a_{j,j}} \quad b_j \quad {}^{1}_{i\ \ s} a_{j,(j\ i)} x_{j\ i}^{(k\ 1)} \quad {}^{s}_{i\ 1} a_{j,(j\ i)} x_{j\ i}^{(k\ 1)}$$

This paradigm of computation is called a stencil computation, and s is the stencil size, which can vary from N (full range interaction) to 1 (nearest neighbours interaction). The C sequential code to compute reported expressions, assumes the generalized form of the loop nest in Fig. 5. The outer loop on index n represents the iteration for convergence, the intermediate loop on j represents the scanning of the elements of $\underline{x}$, while the inner loop on i represents the update of the jth element of x, based on the value of neighbour elements belonging to the stencil. The + sign in the updating statement can be substituted by any associative operator, while $f_{j,i}$ and g_j are generic functions, which could depend on i and j.

Loop carried dependence relation can be taken in account as set of direction vectors $\{(1,0,<),(0,0,>),(1,0,0)\}$: with reference to conditions enabling parallelizations [1], neither the outermost loop (n) nor the intermediate loop (j) can be parallelized. The innermost loop (i) can be unrolled and it is possible to use a single block that sums together all the inputs, exploiting the associativity and commutativity of the addition. SUNthesis resorts to a tree-organization of adders, reducing the propagation delay of statement 3-4 of Fig. 5 that are just the core of the computation. Loop on i derives from a lack of a summatory operator in C and a lack of correspondent functional block in conventional processors, while in a hardware synthesis process, loop can be substituted by a

```
1.  for (n = 0; n < Niter; n++)  for (j = 0; j < N; j++){
2.    temp = 0;
3.    for (i = −s; i <= s; i++)
4.      if ((i + j >= 0)&&(i + j <= N − 1))  temp += f_{j,i}(x[i + j]);
5.      x[j] = g_j(temp);}
```

Fig. 5. Sequential SOR algorithm in C. It is worth point out that perfect loop model can be adopted.

block directly implementing this arithmetic primitive with a significant gain of performance.

We describe the algorithm for systematic and partial unrolling, making reference to loop j of SOR code in Fig. 5, where, without loss of generality, we assume N = 8, S = 1. Furthermore we suppose that operators $f_{j,i}$ and g_j are multiplication operations respectively by bidimensional array a, and by vector b, including dependences of operator f_j,i and g_j from subscripts j and i. The specified code is reported in Fig. 6.. Function $f_{j,i}$ is substituted with a[j][i+1]

```
1.  int n, i, j; signed_int8 x[8], a[3][8], b[8]; int temp;
2.    for (n = 0; n < Niter; n++) #pragma clk
3.      for (j = 0; j < 8; j++) { #pragma unrolling
4.        for (i = −1; i ≤ 1; i++) #pragma unrolling, partial eval, const folding
5.          if ((i + j >= 0)&&(i + j <= 7)) temp += a[j][i + 1] ∗ x[i + j];
6.          x[j] = b[j] ∗ temp;
```

Fig. 6. SOR algorithm in C, specified with described assumptions, with declaration of type variable. The pragma directive clk allows to specify which loop corresponds to the clock, i.e. execution of which loop of the nest is completed in a clock tick.

in order to guarantee that subscripts falls in $[0, 2 \cdot S]$ in according to C syntax. Examples of C code in the following are enhanced with some pragma directives that mean application of transformations by SUNthesis, helping to understand effects of transformations on internal graph structures. The pragma *unrolling* means that unrolling is complete, i.e. all instances of the loop referred by unrolling directive are concurrently active. The pragma directive *clk* implicates that all the hardware blocks corresponding to statements executed in the loop n, must be accommodated in a clock tick, or an iteration of loop n would be executed in a clock period. After application of complete unrolling on loop i, it is possible to apply partial evaluation, constant evaluation (and if necessary redundancies removing), that can simplify the if conditions. In Fig. 7 we report resulting code

During unrolling of j loop the signal temp has been replicated one for each loop instance (*temp_j* where j = 0 . . . 7), j and i values have been propagated

4. $\#pragma\ partial\ eval,\ const\ folding$
5. $if\ ((-1+j \geq 0)\&\&(-1+j \leq 7))\ temp+= a[j][-1+1] * x[-1+j];\ //i=-1$
6. $if\ ((j \geq 0)\&\&(j \leq 7))\ temp+= a[j][1] * x[j];\ //i=0$
7. $if\ ((1+j \geq 0)\&\&(1+j \leq 7)\ temp+= a[j][1+1] * x[1+j];\ //i=1$
8. $x[j] = b[j] * temp;$

Fig. 7. Loop i of the SOR algorithm in C after complete unrolling and constant propagation.

through the code. All the if statements with condition not verified are clearly considered dead-code, and hence are discarded with corresponding hardware blocks. A simple data-dependence analysis [4] can point out which values can be forwarded, because they are calculated in the same clock cycle in which other hardware blocks need them. For instance, in our example, updated value of x[j] element is loaded into the register at the end of any clock tick, but it is needed, if j = 1 ... N-1, to calculate new value of x[j+1], so the block corresponding to instances of statement 5 of Fig. 6, can exploit x_new[i]. In Fig. 8 we report the resulting code:

1. $for(n=0; n<Niter; n++)\ \{\#pragmaclk$
2. $\quad temp\_0 = 0;\ temp\_0+= a[0][1] * x[0];\ //j=0, i=0$
3. $\quad temp\_0+= a[0][2] * x[1];\ //i=1$
4. $\quad x\_new[0] = b[0] * temp\_0;$
5. $\quad \ldots$

$\ldots$ $\quad temp\_7 = 0;\ temp\_7+= a[7][0] * x\_new[6]; //j=1, i=0 \ldots$

Fig. 8. SOR algorithm in C, completely unrolled with respect to j loop and i loop, after partial evaluation, constant propagation of i and j, and dead code elimination.

The algorithm obtained with the above-mentioned sequence of transformations can be translated in a VHDL description with a direct inspection of the annotated graph, and in example of complete unrolled description of Fig. 8, SUNthesis produces the circuit of Fig. 9.

Unrolling of loop i - serial implementation of SOR algorithm: Serial version of SOR algorithm of Fig. 5 (with the same assumptions made in the completely unrolled example) is derived by the code of Fig 6 leaving pragma unrolling of line 3 and substituting it with a pragma clk. It corresponds to datapath depicted in Fig. 10. In this case a decreasing of throughput corresponds to a decrease of area consumed. It is possible to obtain other intermediate implementations of SOR algorithm using a partial unrolling, providing a trade-off between performances and area required.

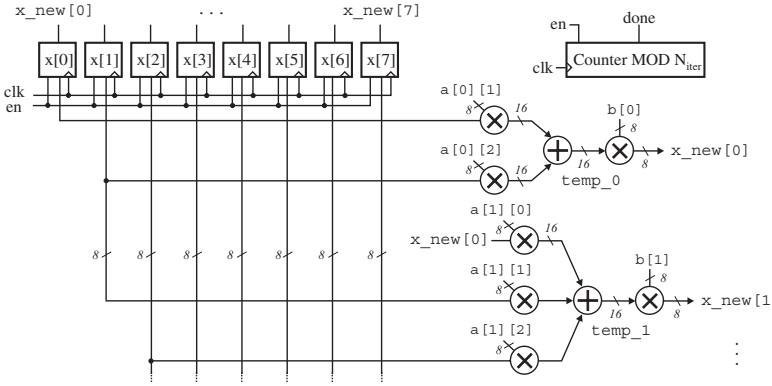

Fig. 9. Circuit corresponding to parallel description of C code of Fig. 8. Counter stores value of number n of iterations, and issues done signal when circuit has completed elaboration, while en started the execution.

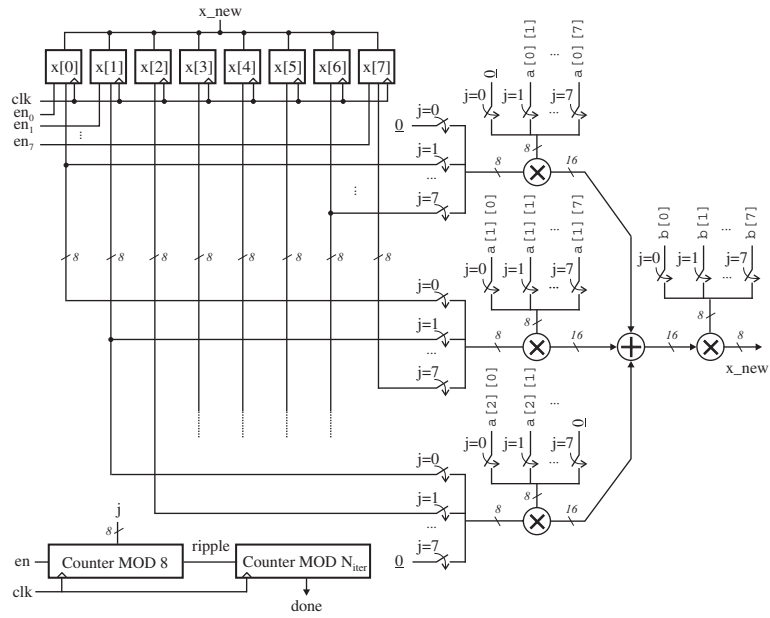

Fig. 10. Serial implementation of SOR. Please note the extended use of tristate buffer, represented as controlled switches, and enable signals for flip-flop, to implement respectively multiplexers and demultiplexers.

Unrolling Factor	Area [slices]	Tristate buffers	Minimum T_{Clock} [ns]	Execution Time Sw [s]	Execution Time Hw [s]	Speedup
U = 1	795 (4%)	1984	46,6	0,373	2,03	5,44
U = 2	1162 (6%)	1984	64,5	0,259	2,03	7,86
U = 4	2048 (11%)	1984	108,3	0,217	2,03	9,37
U = 8	3623 (19%)	0	131,3	0,131	2,03	15,5

Table 1. Results for the case N = 8, S = 7, exploring with different unrolling factor U the design space (U=1 is serial case, and U=8 is parallel). Execution times are referred to 1 million iterations of variables update. Occupation as number of slices used and tristate buffer are also reported.

Unrolling Factor	Area [slices]	Tristate buffers	Minimum T_{Clock} [ns]	Execution Time Sw [s]	Execution Time Hw [s]	Speedup
U = 1	2257 (12%)	8064	56,5	0,904	8,1	8,96
U = 16	9172 (48%)	0	267,5	0,267	8,1	30,3

Table 2. Results for the case N = 16, S = 15, where U = 1 is serial case, while U = 16 parallel

	Xilinx after place&route			SUNthesis estimate		Xilinx estimate after synthesis
Unrolling Factor	Area [slices]	T_{Clock} [ns]	Logic Levels	Area [slices]	T_{Clock} [ns]	T_{Clock} [ns]
U = 1	795	46,6	18	823(+3,5%)	31,4(-32%)	31,5(-32%)
U = 2	1162	64,5	27	1213(+4,4%)	53,8(-17%)	40,4(-37%)
U = 4	2048	108,3	37	2037(-0,5%)	90,8(-16%)	64,1(-40%)
U = 8	3623	131,3	55	3617(-0,2%)	164(+25%)	94,5(-28%)

Table 3. Comparison among estimation of area-time performance for N = 8, S = 7. Values of area-time performance after place&route process, estimation of area-time with SUNthesis (without synthesis), and finally estimation of clock period after synthesis pass. Area estimation is very close to actual results, while minimum operating frequency estimation is less close to actual performances because of route delays.

4 Numerical Results and Conclusions

In this paper, for synthesis of VHDL RTL description, we have targeted a Xilinx Virtex-E 2000-6bg560. Xilinx XCV2000E presents 19200 slices and 19520 tris-

tate buffers. All the synthesis were been realized using Xilinx ISE 4. In Tab. 1 we compared performances of circuits produced by SUNthesis with execution time of an Intel Pentium 3 processor running at 1 GHz. Hardware execution is between 5 and 15 times faster than a P3 1GHz. It is clear that if N and S grow, the speed-up can be higher. This is proven by results reported in Tab. 2 referring to the case N = 16, S = 15, in which parallel version can run 30 times faster than software version. Finally in Tab. 3 we report results of comparison of area-time estimation done by SUNthesis with respect to results of synthesis process. It is worth point out that early SUNthesis estimates are in very good accordance to actual results. Note also that after synthesis pass ISE values worst than SUNthesis minimum period, although it has information about gate netlist not available to SUNthesis, proving effectiveness of our approach. Reported experiments proved validity of its early estimation of impact of transformations on area requirements and performances.

References

1. M.E. Wolf and M.S. Lam: "A loop transformation theory and an algorithm to maximize parallelism", IEEE Trans. Parallel. Distrib. Syst. Vol.2, pp. 452-471, Oct 1991.
2. M. Weinhardt and W. Luk: "Pipeline Vectorization", IEEE Trans. On CAD of Integrated Circuits and Systems, Vol.20, No. 2, Feb 2001.
3. M. Rencher, B. L. Hutchings, "Automated Target Recognition on SPLASH2", IEEE Symposium on Field-Programmable Custom Computing Machines, 1997.
4. H. Zima and B. Chapman: "Supercompilers for Parallel and ", Reading MA: Addison-Wesley, 1991.
5. B. Di Martino, G. Iannello, "Parallelization of Nonsimultaneous...", in: *Parallel Processing*, Lecture Notes in Computer Science n. 854, pp. 253-262, Springer-Verlag, 1994.
6. B. Di Martino, "Algorithmic Concept Recognition Support for Automatic ...", *Journ. of Information Science and Engineering*, Vol. 14, n. 1, pp. 191-203, March 1998.
7. M. J. Wolfe: Optimizing Supercompilers for Supercomputers. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Mass.
8. M. Girkar and C.D. Polychronopoulos : "Automatic Extraction of Functional Parallelism...", IEEE Trans. On Par. and Distr. Syst. Vol.3, No.2, March 1992, pp. 166-178.
9. A. J. Elbirt, C. Paar, "An FPGA Implementation and Performance Evaluation ...", ACM/SIGDA International Symposium on FPGAs, pp. 33-40, 2000.
10. F. Irigoin and R. Triolet, "Supernode partitioning", In Proc. 15th Annual ACM SIGACT-SIGPLAN Synp Principles Programming Languages, Jan. 1988, pp. 319-329.
11. N. Tawbi : "Estimation of nested loops execution time by integer arithmetic in convex polyhedra", Parallel Processing Symposium, 1994. Proceedings., 1994, Page(s): 217-221
12. D.E. Knuth, "The Art of Computer Programming", Vol. I, Addison-Wesley, 1973
13. R. Rinker et al.: "An Automated Process for Compiling Dataflow Graphs into ...", IEEE Trans. On VLSI Systems, Vol.9, No.1, Feb. 2001, pp. 130-139.