# Loop Parallelization for Reconfigurable Architectures

Ozana Silvia Dragomir, Elena Moscu Panainte, Koen Bertels
{O.S.Dragomir, E.Moscu-Panainte, K.L.M.Bertels}@tudelft.nl

Computer Engineering, EEMCS,
Delft University of Technology,
Mekelweg 4, 2628 CD Delft,
The Netherlands

*Abstract*—**Reconfigurable Computing (RC) is one of the research directions that focuses on accelerating applications. In the presented approach we assume the Molen machine organization and the Molen programming paradigm as our framework. Molen combines a general purpose processor (GPP) and a Field Programmable Gate Array (FPGA), having the advantages of both speed of hardware and flexibility of software execution. In this paper we present a method that allows complete automation of efficient code generation with the Molen compiler for reconfigurable architectures in Delft WorkBench project. The proposed algorithm computes the optimal degree of parallelism for a kernel $K$ called from inside a loop or loop nest, in order to achieve the maximum performance, taking into consideration the resource constraints. The input data for the algorithm consists of profiling information about the execution times for running $K$ in both hardware and software, the memory transfers and the occupied area.**

## I. Introduction

Reconfigurable Computing (RC) is becoming increasingly popular and the common solution for obtaining a significant performance increase is to identify the application kernels and accelerate them on hardware. Usually these kernels consist of loop nests, and there are a number of approaches that use different loop optimizations (such as loop unrolling, software pipelining, loop shifting, loop distribution, loop merging, loop tiling, etc) in order to maximize the parallelism inside the loop and to obtain in this way a significant speedup ([1], [2], [3], [4], [5], [6], [7]).

In this paper we propose a method to determine the optimal unroll factor that determines the parallelism granularity. The paper is organized as follows. Section II introduces the background and related work. In Section III we give a general definition of the problem and present the target architecture and application. We propose a method for solving the problem in Section IV. Finally, concluding remarks are presented in Section V.

## II. Background and related work

One challenge for loop optimizations is the selection, parametrization and order of the applied transformations. In the context of reconfigurable hardware, additional decision factors such as area usage and parallel execution have to be taken into account. The most popular loop optimizations that are used in reconfigurable computing are *loop unrolling* and *software pipelining*.

Loop unrolling is a technique that expands the loop body, so that a new iteration consists of 2 or more of the initial iterations. The number of times that the loop is expanded is called unroll factor ($u$) and the iteration step will be $u$ instead of 1. If there are no dependencies between iterations, than the unrolled body can be executed in parallel. A main problem in the domain of loop optimizations is that of finding the optimal unroll factor ([8], [5])

Software pipelining is used to achieve higher instruction level parallelism, by moving operations across iteration boundaries. To be more specific, the operations in a loop iteration are broken into $s$ stages. A single iteration executes stage 1 from iteration $i$, stage 2 from iteration $i-1$ and so on. The method we propose for computing the optimal degree of parallelism uses only loop unrolling, but we take into account that when loop unrolling and software pipelining are used together, the performance gain may overcome their separate contributions [9], [10].

The work presented in this paper is related to Delft WorkBench project. DWB is a semi-automatic toolchain platform for integrated hardware-software co-design in the context of Custom Computing Machines (CCM) which targets the Molen polymorphic machine organization [11] and supports the Molen programming paradigm [12]. The general workflow is shown in Fig. 1. The kernels are identified in the first stage of profiling and cost estimation. Next, the Molen compiler [13] generates the executable file, re-

placing function calls to the kernels implemented in hardware with specific instructions for hardware reconfiguration and execution, according to the Molen programming paradigm. An automatic tool for hardware generation (DWARV [14]) is used to transform the selected kernels into VHDL code targeting the Molen platform.

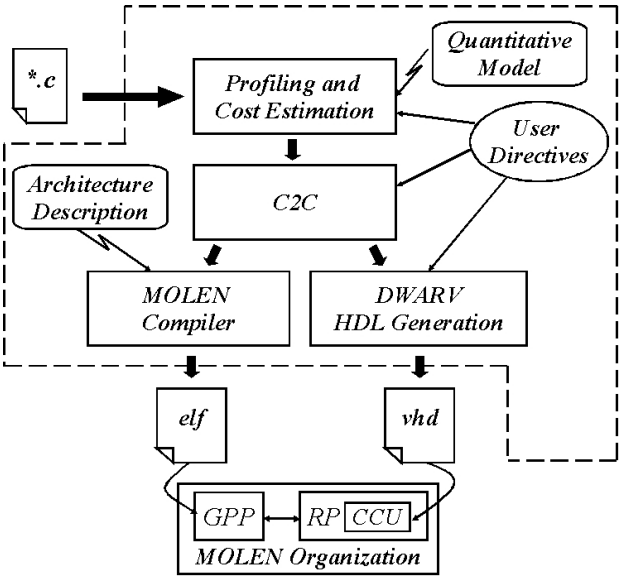



Fig. 1. Delft WorkBench Workflow

Assuming the Molen machine organization [11] and the Molen programming paradigm [12] as our framework, we focus our research in the direction of parallelizing a multimedia application by executing multiple instances of the kernel in parallel on the reconfigurable hardware.

Several other research projects develop C to VHDL frameworks, trying to exploit as much as possible the advantages of reconfigurable systems by maximizing the parallelism in targeted applications.

ROCCC is a C to hardware compilation project whose objective is the FPGA-based acceleration of frequently executed code segments (loop nests). The compiler applies loop unrolling, fusion and strip mining and creates pipelines for the unrolled loops in order to efficiently use the available area and memory bandwidth of the reconfigurable device [7].

Cardoso and Diniz [5] use unroll-and-jam (unrolling one or more nested loops in the iteration space and fusing inner loop bodies together) to expose operator parallelism. The design space algorithm evaluates a set of possible unroll factors for multiple loops in the loop nest, searching for the one that leads to a high-performance, balanced, efficient design.

Weinhardt and Luk [2] present pipeline vectorization, a method for synthesizing hardware pipelines based on software vectorizing compilers. In their approach, full loop unrolling is used to increase basic block size and extend the scope of local optimizations.

Liao et al. [10] propose a model for hardware realization of kernel loops. The compiler is used to extract certain key parameters of the analyzed loop. From these parameters, taken into account the resource constraints, the user is informed about the performance that can be obtained by unrolling the loop or applying loop unrolling together with software pipelining. The algorithm also suggests the optimal unroll to be used, but the main difference between our approaches is that their method does not consider parallel executions.

DRESC compiler performs software pipelining [4] in order to parallelize the kernels identified in the profiling/partitioning step, using the architecture representation as input. Traditional ILP scheduling techniques are applied to discover the available moderate parallelism for the non-kernel code. The speed-up reported for a MPEG2-decoder mapped over an 8-issue VLIW is about 12 times for kernels and 5 times for the entire application.

PARLGRAN [15] is to our knowledge the only approach that tries to maximize performance on reconfigurable architectures by selecting the parallelism granularity for each individual data-parallel task. However, this approach is different than ours by taking into consideration the physical (placement) constraints and reconfiguration overhead, without addressing the memory bottleneck problem.

## III. Problem overview

We denote by kernel a function that represents more than a threshold percentage of execution time from the total sequential program execution time. The targeted applications have kernels inside loops or nested loops, without data dependencies between different loop iterations.

The problem addressed in this paper is to find the optimal unroll factor $u$ of the loop (loop nest) with a kernel $K$, so that $u$ instances of $K$ run in parallel on the reconfigurable hardware, leading to the maximum possible speedup, taking into consideration the total available area and the area requirements for the kernel. More specifically, we do not aggressively optimize the kernel implementation, but we focus on the optimization of the application for any hardware implementation of the kernel.

The factors taken into consideration are:

- area occupied by one kernel running on FPGA
- available area (it is possible that not all area is available, because other kernels may be configured in the same time)
- execution time of the kernel in software and in hardware (always in GPP cycles)
- number of memory transfer operations in one kernel instance
- FPGA throughput (the amount of data - in bits - that can be accessed in one cycle)

We assume there is no latency due to configuration of the kernel on the hardware (it can be statically configured in advance, so that it is hidden by the software execution of the program).

### A. Target architecture

We assume the Molen machine organization [11] and the Molen programming paradigm [12] as our framework. The Molen machine organization has been implemented on Virtex-II Pro XC2VP30 device [16]. The memory design uses the available on-chip memory blocks of the FPGA, 64KB for program data and 64KB for program instructions. The communication between the general purpose processor (GPP) and the reconfigurable processor (passing of parameters and result) is performed through the so-called *exchange registers (XREGs)*, which are implemented also in BRAM. The total resources consumed by the Molen implementation on the XC2VP30 chip are less than 2% [17] .

### B. Target application

We chose as a case study a loop where the loop body contains two functions: the first one ($CPar$) computes the parameters for the second one, which is the application kernel ($K$). The sample loop is shown in Example 1.

Example 1 Loop containing a kernel call

```
for (i = 0; i < N; i++ ) {
  /* Compute parameters for K() */
  CPar (i, blocks);
  /* Kernel function */
  K (blocks[i]);
}
```

**Assumptions:**

1. Access time for on-chip memory is 3 cycles for reading and storing the value into a register and 1 cycle for writing a value;
2. All on-chip memory transfers are performed sequentially (also because the VHDL code is not optimized);
3. All local variables/arrays are stored in the FPGA's local memory, so they can be accessed in parallel.

## IV. Proposed method

In this section we present a method to determine the optimal unroll factor having as input the profiling information (execution time and number of memory transfers) and area usage for one instance of the kernel. We illustrate the method by unrolling with factor $u$ the code from Example 1. Assuming that $N \ div \ u = 0$, the result looks like in Example 2.

Example 2 Loop after unrolling with factor $u$

```
for (i = 0; i < N; i += u) {
  CPar (i + 0, blocks);
  CPar (i + 1, blocks);
  ...
  CPar (i + u − 1, blocks);

/* Execute u instances of K() in parallel */
  #pragma parallel
   K (blocks[i + 0]);
   K (blocks[i + 1]);
   ...
   K (blocks[i + u − 1]);
  #end parallel
}
```

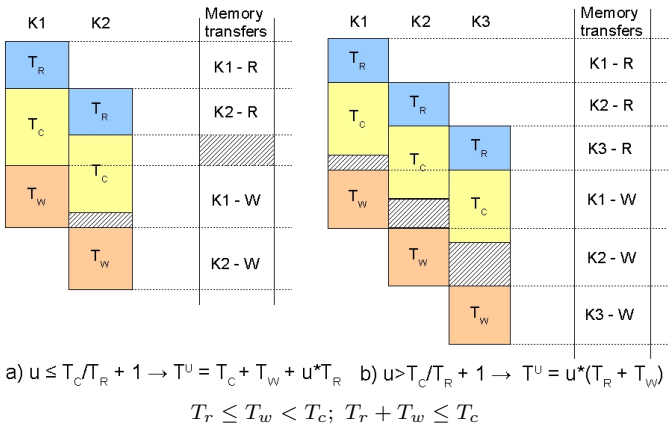
### A. Memory bottleneck

Ideally, the degree of parallelism is bounded only by the area availability. However, the FPGA throughput is an important bottleneck in achieving the ideal parallelism. Considering $T_r$ and $T_w$ the times for memory read and write accesses and $T_c$ the computation time on the hardware, the total time for running a kernel $K$ in hardware is $T_r + T_w + T_c$. Without losing the generality of the problem, we assume that the memory reads are all done in the beginning and memory writes in the end. Then, a new instance of $K$ can start only after a time $T_r$, as illustrated in Fig. 2.

One bound for the degree of parallelism on the reconfigurable hardware is set by the condition that memory access requests from different kernel instances do not overlap:

$$u \times T_m \leq T_m + T_c \ \Rightarrow \ u \leq U_m = \frac{T_c}{T_m} + 1 \qquad (1)$$

where $T_m = min(T_r, T_w)$.




$T_r \leq T_w < T_c;\ T_r + T_w \leq T_c$

Fig. 2. Parallelism on reconfigurable hardware

The time for running $u$ instances of $K()$ on the reconfigurable hardware is:

$$T^u = \begin{cases} T_c + u \cdot T_r & if \quad u \leq T_c/T_m + 1 \\ u \cdot (T_r + T_w) & if \quad u > T_c/T_m + 1 \end{cases} \tag{2}$$

*B. Algorithm*

Each iteration in Example 2 consists of $u$ sequential executions of the function $CPar()$ followed by the parallel execution of $u$ kernel instances (there is an implicit synchronization point at the end of the parallel region).

We use the following notations:

- $N = N^1_{iter}$ - initial number of iterations (before unrolling)
- $T^1_{par}$ - number of cycles for $CPar()$ in software
- $T^1_{K(sw)}$ / $T^1_{K(hw)}$ - number of cycles for one instance of K() running in software/hardware;
- $T^1_{iter(sw)}$ / $T^1_{iter(hw)}$ - number of cycles for one loop iteration with K() running in software/hardware
- $T_{loop(sw)}$ / $T_{loop(hw)}$ - number of cycles for the loop nest with K() running in software/hardware

Using these data, we compute the following:

- $N^u_{iter}$ - number of iterations in the loop nest, depending on the unrolling factor $u$:

$$N^u_{iter} = \lfloor N/u \rfloor, where \lfloor x \rfloor \leq x < \lfloor x \rfloor + 1 \tag{3}$$

- $T^u_{K(sw)}$ - number of cycles for $u$ calls (one iteration) of K() in software:

$$T^u_{K(sw)} = T^1_{K(sw)} \cdot u \tag{4}$$

- $T^u_{K(hw)}$ - number of cycles for $u$ kernel instances running in parallel on FPGA, computed using formula 2 (taking into consideration Assumption 3):

$$T^u_{K(hw)} = T_c + u \cdot T_m \tag{5}$$

The number of cycles for one iteration for unroll factor $u$ is:

$$T^u_{iter(sw)} = T^1_{K(sw)} \cdot u + T^1_{par} \cdot u \tag{6}$$

$$T^u_{iter(hw)} = T_c + T_m \cdot u + T^1_{par} \cdot u \tag{7}$$

For the general case ($N$ *div* $u$ may be different than 0), the formulas for computing the total number of cycles for the loop nest are:

$$T_{loop(sw)} = T^1_{iter(sw)} \cdot N = constant \tag{8}$$

$$T^u_{loop(hw)} = (T^1_{par} + T^1_m) \cdot N + T_c \cdot \lceil N/u \rceil, \tag{9}$$

where $x \leq \lceil x \rceil < x + 1$

If $u + 1$ is not a divisor of $N$, then

$$\lceil N/u \rceil = \lceil N/(u+1) \rceil \tag{10}$$

meaning that $T^u_{loop(hw)} = T^{u+1}_{loop(hw)}$. In this case, $u + 1$ is not a valid choice because it generates the same execution time as the unroll factor $u$, but it occupies more area. For this reason, we take into consideration only the unroll factors which are divisors of $N$.

We can rewrite $T^u_{loop(sw)}$ and $T^u_{loop(hw)}$ as:

$$T^u_{loop(sw)} = T^u_{iter(sw)} \cdot N^u_{iter} \tag{11}$$

$$T^u_{loop(hw)} = T^u_{iter(hw)} \cdot N^u_{iter} \tag{12}$$

The speedups for the execution of kernels at iteration level and at loop nest level are:

$$S^u_K = \frac{T^u_{K(sw)}}{T^u_{K(hw)}}; \quad S^u_{loop} = \frac{T_{loop(sw)}}{T_{loop(hw)}} \tag{13}$$

The optimal unroll factor from the speedup point of view is the maximum value of $u$ for which the difference between two consecutive speedup factors is less than a certain threshold value $F$, chosen in the beginning (it can depend on the area occupied by one instance of the kernel or it can be just the value 1, etc.). Using the notations above, the problem of finding the optimal unroll factor becomes:

Find $min(u)$ such that $\Delta S = S^{u+1}_{loop} - S^u_{loop} < F$.

$$S^u_{loop} = \frac{T_{loop(sw)}}{T_{loop(hw)}} = \frac{T^u_{iter(sw)} \cdot N^u_{iter}}{T^u_{iter(hw)} \cdot N^u_{iter}} \Rightarrow$$

$$S^u_{loop} = \frac{T^u_{iter(sw)}}{T^u_{iter(hw)}} = \frac{(T^1_{iter(sw)}) \cdot u}{T_c + (T_m + T^1_{par}) \cdot u}$$

Using the notations:

$$x = \frac{T_c}{T_m + T^1_{par}} \quad and \quad y = \frac{T^1_{iter(sw)}}{T_m + T^1_{par}} \tag{14}$$

$\Delta S$ becomes:

$$\Delta S = \frac{(u+1)\cdot y}{(u+1)+x} - \frac{u\cdot y}{u+x} = \frac{x\cdot y}{u^2+u\cdot(2x+1)+(x^2+x)}$$

$$\Delta S < F \Rightarrow u^2+u\cdot(2x+1)+(x^2+x)-x\cdot y/F > 0$$

The positive root of the equation is:

$$u_+ = \frac{-(1+2x)+\sqrt{1+4xy/F}}{2} = \sqrt{\frac{xy}{F}+\frac{1}{4}}-x-\frac{1}{2} \quad (15)$$

Taking into account also the area constraints, another upper bound for the parallelism degree is set by

$$U_a = \frac{Area_{(available)}}{Area_{(K)}}.$$

where:

- $Area_{(available)}$ - is the available area, taking into account the resources occupied by Molen and by other configurations;
- $Area_{(K)}$ - is the area occupied by one instance of $K$.

If $u_+ < min(U_a, U_m)$, then the optimal unroll factor is $min(u)$ such that $u$ is a divisor of $N$ and $u_+ < u \leq min(U_a, U_m)$; else, we choose it as $max(u)$ such that $u$ is a divisor of $N$ and $u \leq min(U_a, U_m)$.

## V. Conclusion

In this paper we presented a method to automatically compute the optimal number of instances of a kernel $K$ to be run in parallel on reconfigurable hardware by applying loop unrolling. The algorithm uses only the profiling information about memory transfers and execution times in software and hardware and information about area usage for one kernel instance. Its implementation in the compiler decreases the time for design-space exploration and makes use efficiently of the hardware resources.

Using the proposed method, we achieved a theoretical speedup for an automatically generated VHDL implementation of DCT with a factor of 10. As our approach demonstrates the potential for significant performance improvement, in future work we

## References


[1] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution", *1993 Workshop on Languages and Compilers for Parallel Computing*, vol. 768, pp. 301–320, 1993.

[2] M. Weinhardt and W. Luk, "Pipeline vectorization", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[3] A. Fraboulet, K. Kodary, and A. Mignotte, "Loop fusion for memory space optimization", *ISSS '01: Proceedings of the 14th International Symposium on Systems Synthesis*, pp. 95–100, 2001.

[4] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling", *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, 2003.

[5] J. M. P. Cardoso and P. C. Diniz, "Modeling loop unrolling: approaches and open issues", *the 4th International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS'04)*, pp. 224–233, 2004.

[6] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Loop shifting and compaction for the high-level synthesis of designs with complex control flow", *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, 2004.

[7] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, "Optimized generation of data-path from C codes for FPGAs", *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 112–117, 2005.

[8] A. Koseki, H. Komastu, and Y. Fukazawa, "A method for estimating optimal unrolling times for nested loops", *ISPAN '97: Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97)*, p. 376, 1997.

[9] S. Novack and A. Nicolau, "Resource directed loop pipelining: exposing just enough parallelism", *The Computer Journal*, vol. 40, pp. 311–321, 1997.

[10] J. Liao, W.-F. Wong, and T. Mitra, "A model for hardware realization of kernel loops", *13th International Conference on Field-Programmable Logic and Applications (FPL'03)*, 2003.

[11] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor", *IEEE Transactions on computers*, November 2004.

[12] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. M. Panainte, "The Molen programming paradigm", *the 3rd International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03)*, July 2003.

[13] E. M. Panainte, K. Bertels, and S. Vassiliadis, "The PowerPC backend Molen compiler", *14th International Conference on Field-Programmable Logic and Applications (FPL'04)*, 2004.

[14] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, J. Lu, and S. Vassiliadis, "DWARV: DelftWorkbench automated reconfigurable VHDL generator", *the 17th International Conference on Field Programmable Logic and Applications (FPL'07)*, 2007.

[15] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "PARLGRAN: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures", in *Design Automation, 2006. Asia and South Pacific Conference on*, p. 6pp., 24-27 Jan. 2006.

[16] http://www.xilinx.com/bvdocs/publications/ds083.pdf.

[17] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "The Virtex II Pro MOLEN processor", *the 4th International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS'04)*, 2004.