Optimal Unroll Factor for Reconfigurable Architectures

Ozana Silvia Dragomir, Elena Moscu-Panainte, Koen Bertels, and Stephan Wong

Computer Engineering, EEMCS, Delft University of Technology,
Mekelweg 4, 2628 CD Delft, The Netherlands
{O.S.Dragomir, E.Moscu-Panainte, K.L.M.Bertels, J.S.S.M.Wong}@tudelft.nl

Abstract. Loops are an important source of optimization. In this paper, we address such optimizations for those cases when loops contain kernels mapped on reconfigurable fabric. We assume the Molen machine organization and Molen programming paradigm as our framework. The proposed algorithm computes the optimal unroll factor u for a loop that contains a hardware kernel K such that u instances of K run in parallel on the reconfigurable hardware, and the targeted balance between performance and resource usage is achieved. The parameters of the algorithm consist of profiling information about the execution times for running K in both hardware and software, the memory transfers and the utilized area. In the experimental part, we illustrate this method by applying it to a loop nest from a real-life application (MPEG2), containing the DCT kernel.

1 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. As loops represent an important source of performance improvement, we investigate how existing loop optimizations can be applied when hardware kernels exist in the loop body. Assuming the Molen machine organization [1] as our framework, we focus our research in the direction of parallelizing applications by executing multiple instances of the kernel in parallel on the reconfigurable hardware.

Optimal is defined in this paper as the largest feasible unroll factor, given area constraints, performance requirements and memory access constraints, taking into account also that multiple kernels may be mapped on the area. The contributions of this paper are: a) an algorithm to automatically determine the optimal unroll factor, based on profile information about memory transfers, available area, and software/hardware execution times; b) experimental results for a well known–kernel – DCT (Discrete Cosine Transformation), showing that the optimal unroll factor is 6, with a speedup of 9.55x and utilized area of 72%.

The paper is organized as follows. Section 2 introduces the background and related work. In Section 3, 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 4 and show the results for a specific application in Section 5. Finally, concluding remarks and future work are presented in Section 6.

2 Background and related work

The work presented in this paper is related to the Delft WorkBench (DWB) project. The 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 [1]. More specifically, the DWB supports the entire design process, as follows. The kernels are identified in the first stage of profiling and cost estimation. Next, the Molen compiler [2] generates the executable file, replacing 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 [3]) is used to transform the selected kernels into VHDL code targeting the Molen platform.

Several approaches – [4], [5], [6], [7], [8], [9] are focused on accelerating kernel loops in hardware. Our approach is different, as we do not aggressively optimize the kernel implementation, but we focus on the optimization of the application for any hardware implementation, by executing multiple kernel instances in parallel.

PARLGRAN [10] is an 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 in several ways: (i) they target task chains and make a decision on the parallelism granularity of each task, while we target loops (loop nests) with kernels inside them and make a decision on the unroll factor; (ii) in their case, the task instances have identical area requirements but different workloads, which translates into different executions time (a task is split into several sub-tasks); in our algorithm, all instances have the same characteristics in both area consumption and execution time; (iii) their algorithm takes into consideration the physical (placement) constraints and reconfiguration overhead at run-time, but without taking into account the memory bottleneck problem; we present a compile-time algorithm, which considers that there is no latency due to configuration of the kernels (static configurations), but takes into account the memory transfers.

3 Problem statement

Loops represent an important source of optimization, and there are a number of loop transformations (such as loop unrolling, software pipelining, loop shifting, loop distribution, loop merging, loop tiling, etc) that are used to maximize the parallelism inside the loop and improve the performance. The applications we target in our work have loops that contain kernels inside them. One challenge we address is to improve the performance for such loops, by applying standard loop transformations such as the ones mentioned above.

In this paper, we focus on loop unrolling and present the assumptions and parameters for our model. Loop unrolling is a transformation that replicates the loop body and reduces the iteration number. Traditionally it is used to eliminate the loop overhead, thus improving the cache hit rate and reducing branching, while in reconfigurable computing it is used to expose parallelism.

The problem definition is: find the optimal unroll factor u of the loop (loop nest) with a kernel K, such that u identical instances of K run in parallel on the reconfigurable hardware, leading to the best balance between the performance and area utilization. The method proposed in this paper addresses this problem, given a C implementation of the target application and a VHDL implementation of the kernel.

We target the Molen framework, which allows multiple applications to run simultaneously on the reconfigurable hardware. The algorithm computes (at compile time) the optimal unroll factor, taking into consideration the memory transfers, the execution times in software and hardware, the area requirements for the kernel, and the available area (without physical details, P&R, etc). Thus, because of the reconfigurable hardware's flexibility, the algorithm's output depends on the hardware configuration at a certain time.

The main benefits of this algorithm are that it can be integrated in an automatic toolchain and it can use any hardware implementation of the kernel. Thus, performance can be improved even when the kernel is already optimized. Our assumptions regarding the application and the framework are presented below.

Target architecture. The Molen machine organization has been implemented on Vitex-II Pro XC2VP30 device, utilizing less than 2% of the available resources of the FPGA. The memory design uses the available on-chip memory blocks of the FPGA.

The factors taken into consideration by the proposed method are:

- area utilized by one kernel running on FPGA;
- available area (other kernels may be configured in the same time);
- execution time of the kernel in software and in hardware (in GPP cycles);
- the number of cycles for memory transfer operations for one kernel instance running in hardware;
- available memory bandwidth.

General and Molen-specific assumptions for the model:

1. There are no data dependencies between different iterations of the analyzed loop. Practical applications that satisfy this assumption exist, for example MPEG2 multimedia benchmark.
2. The loop bounds are known at compile time.
3. Inside the kernel, all memory reads are performed in the beginning and memory writes in the end. This does not reduce the generality of the problem for most applications.

4. The placement of the specific reconfiguration instructions is decided by a scheduling algorithm which runs after our algorithm, such that the configuration latency is hidden.
5. Only on-chip memory is used for program data. This memory is shared by the GPP and the CCUs.
6. All data that are necessary for all kernel instances are available in the shared memory.
7. The PowerPC and the CCU run at the same clock.
8. All transfers to/from the shared memory are performed sequentially.
9. Kernel's local variables/arrays are stored in the FPGA's local memory, such that all computations not involving the shared memory transfers can be parallelized.
10. As far as running multiple applications on the reconfigurable hardware is concerned, for now we take into consideration only the area constraints.
11. The area constraints do not include the shape of the design.
12. Additional area needed for interconnecting Molen with the kernels grows linearly with the number of kernels.

Motivational example. Throughout the paper, we will use the motivational example in Fig. 1(a). It consists of a loop with two functions – one function is executed always in software ($CPar$), and the other is the application kernel (K) – implicitly, the execution time for $CPar$ is much smaller than for K. In each iteration i, data dependencies between $CPar$ and K exist, as $CPar$ computes the parameters for the kernel instance to be executed in the same iteration. The example has been extracted from the MPEG2 encoder multimedia benchmark, where the kernel K is DCT.

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

(a)Loop containing a kernel call

```
for (i = 0; i < N; i += u) {
    CPar (i + 0, blocks);
    ...
    CPar (i + u − 1, blocks);
    /*u instances of K() in parallel*/
    #pragma parallel
        K (blocks[i + 0]);
        ...
        K (blocks[i + u − 1]);
    #end parallel
}
```

(b)Loop unrolled with a factor u

Fig. 1. Motivational example

4 Proposed methodology

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) area usage for one instance of the kernel. We illustrate the method by unrolling with factor u the code from Fig. 1(a). Figure 1(b) presents a simplified case when $N \; mod \, u = 0$. Each iteration 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).

Area. Taking into account only the area constraints and not the shape of the design, an upper bound for the unroll factor is set by

$$\mathrm{u_a} = \left\lfloor \frac{Area_{(available)}}{Area_{(K)} + Area_{(interconnect)}} \right\rfloor \text{, where:}$$

- $Area_{(available)}$ is the available area, taking into account the resources utilized by Molen and by other configurations;
- $Area_{(interconnect)}$ is the area necessary to connect one kernel with the rest of the hardware design (we made the assumption that the overall interconnect area grows linearly with the number of kernels);
- $Area_{(K)}$ is the area utilized by one instance of the kernel, including the storage space for the values read from the shared memory. All kernel instances have identical area requirements.

Memory accesses. In the ideal case, all data are available immediately and the degree of parallelism is bounded only by the area availability. However, for many applications, the memory bandwidth is an important bottleneck in achieving the ideal parallelism. We consider that T_r, T_w, respectively T_c are the times for memory read, write, and computation on hardware for kernel K. Then, the total time for running K in hardware is $T_\mathrm{r} + T_\mathrm{w} + T_\mathrm{c}$. Without reducing the generality of the problem for most applications, we assume that the memory reads are performed at the beginning and memory writes in the end. Then, as illustrated in Fig. 2, where we consider $\mathrm{T_w} \leq \mathrm{T_r} < \mathrm{T_c}$ and $\mathrm{T_r} + \mathrm{T_w} > \mathrm{T_c}$, a new instance of K can start only after a time T_r (we denote kernel instances by $K^{(1)}$, $K^{(2)}$, etc)[1].

The condition that memory access requests from different kernel instances do not overlap sets another bound ($\mathrm{u_m}$) for the degree of unrolling on the reconfigurable hardware[1]:

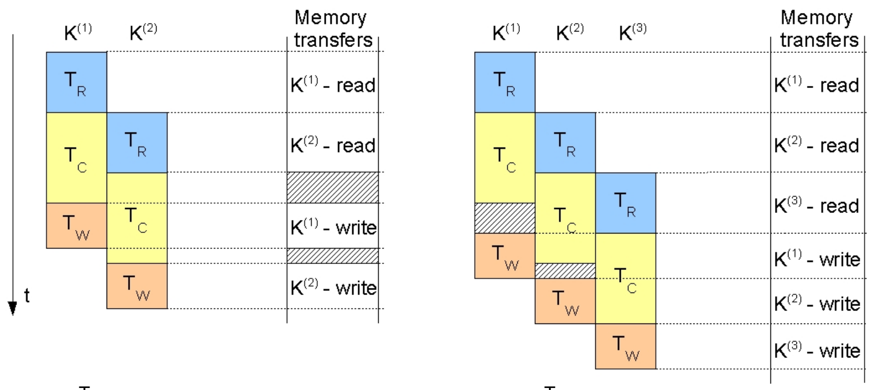
$$u \cdot \min(T_\mathrm{r}, T_\mathrm{w}) \leq \min(T_\mathrm{r}, T_\mathrm{w}) + T_\mathrm{c} \quad \Rightarrow \quad u \leq \mathrm{u_m} = \left\lfloor \frac{T_\mathrm{c}}{\min(T_\mathrm{r}, T_\mathrm{w})} \right\rfloor + 1 \tag{1}$$

The time for running u instances of K on the reconfigurable hardware is[1]:

$$T_{K(\mathrm{hw})}(u) = \begin{cases} T_\mathrm{c} + T_\mathrm{r} + T_\mathrm{w} + (u-1) \cdot \max(T_\mathrm{r}, T_\mathrm{w}) & \text{if } u < \mathrm{u_m} \\ u \cdot (T_\mathrm{r} + T_\mathrm{w}) & \text{if } u \geq \mathrm{u_m} \end{cases} \tag{2}$$

Since we are interested only in the case $u < \mathrm{u_m}$, from (2) we derive the time for u instances of K in hardware as:

$$T_{K(\mathrm{hw})}(u) = T_\mathrm{C} + \min(T_\mathrm{r}, T_\mathrm{W}) + u \cdot \max(T_\mathrm{r}, T_\mathrm{W}) \tag{3}$$



(a) $u < \frac{T_C}{T_W} + 1 \rightarrow T_{k(hw)}(u) = T_C + T_W + u \cdot T_r$ (b) $u \geq \frac{T_C}{T_W} + 1 \rightarrow T_{k(hw)}(u) = u \cdot (T_r + T_W)$

Fig. 2. Parallelism on reconfigurable hardware

Note that when applied to the example in Fig. 2, the case $u < \mathrm{u_m}$ corresponds to Fig. 2(a) and the case $u \geq \mathrm{u_m}$ corresponds to Fig. 2(b). In our example, $T_\mathrm{W} \leq T_\mathrm{r}$, thus $\max(T_\mathrm{r}, T_\mathrm{W}) = T_\mathrm{r}$ and $\min(T_\mathrm{r}, T_\mathrm{W}) = T_\mathrm{W}$. In Fig. 2(a), the time for running in parallel two kernel instances ($K^{(1)}$ and $K^{(2)}$) is given by the time for $K^{(1)}$ ($T_\mathrm{C} + T_\mathrm{r} + T_\mathrm{W}$) plus the necessary delay for $K^{(2)}$ to start (T_r). In Fig. 2(b), $K^{(1)}$ writing to memory is delayed because of $K^{(3)}$ reading from memory; in this case, the actual kernel computation is hidden by the memory transfers and the hardware execution time depends only on the memory access times ($u \cdot (T_\mathrm{r} + T_\mathrm{W})$).

Speedup. In order to compute the optimal unroll factor, we use the following notations:

- N- initial number of iterations (before unrolling);
- $N(u)$ - number of iterations after unrolling with factor u: $N(u) = \lceil N/u \rceil$;
- T_SW - number of cycles for that part of the loop body that is always executed by the GPP (in our example, the *CPar* function);
- $T_{K(\mathrm{sw})}$ / $T_{K(\mathrm{hw})}$ - number of cycles for one instance of K() running in software/hardware;

[1] (1) and (2) are derived from an exhaustive analysis of all possible cases with different relations between T_C, T_r, and T_W. Fig. 2 represents only one of the 8 possible cases ($T_\mathrm{W} \leq T_\mathrm{r} < T_\mathrm{C}$ and $T_\mathrm{W} + T_\mathrm{r} > T_\mathrm{C}$).

- $T_{\text{loop(sw)}}$ / $T_{\text{loop(hw)}}$ - number of cycles for the loop nest with K() running in software/hardware (considering that the unroll factor satisfies the condition $u < \mathrm{u_m}$):

$$T_{\text{loop(sw)}} = (T_{\text{sw}} + T_{K(\text{sw})}) \cdot N \tag{4}$$

$$T_{\text{loop(hw)}} = (T_{\text{sw}} + \max(T_{\text{r}}, T_{\text{w}})) \cdot N + (T_{\text{c}} + \min(T_{\text{r}}, T_{\text{w}})) \cdot N(u) \tag{5}$$

The speedup at loop nest level is: $$S_{\text{loop}}(u) = \frac{T_{\text{loop(sw)}}}{T_{\text{loop(hw)}}} \tag{6}$$

For $u < \mathrm{u_m}$, $T_{\text{loop(hw)}}$ is a monotonic decreasing function; as $T_{\text{loop(sw)}}$ is constant, it means that $S_{\text{loop}}(u)$ is a monotonic increasing function for $u < \mathrm{u_m}$.

When multiple kernels are mapped on the reconfigurable hardware, the goal is to determine the optimal unroll factor for each kernel, which would lead to the maximum performance improvement for the application. For this purpose, we introduce a new parameter to the model: the calibration factor F, a positive number decided by the application designer, which determines a limitation of the unroll factor according to the targeted trade-off. (For example, you would not want to increase the unrolling if the gain in speedup would be with a factor of 0.5%, but the area usage would increase with 15%.) The simplest relation to be satisfied between the speedup and necessary area is:

$$\Delta S(u+1, u) > \Delta A(u+1, u) \cdot F$$

where $\Delta S(u+1, u)$ is the relative speedup increase between unroll factors u and $u+1$:

$$\Delta S(u+1, u) = \frac{S(u+1) - S(u)}{S(u)} \cdot 100\ [\%] \tag{7}$$

and $\Delta A(u+1, u)$ is the area increase. Since all kernel instances are identical, the total area grows linearly with the number of kernels and $\Delta A(u+1, u)$ is always equal to the area utilized by one kernel instance ($Area_{(K)})[\%]$).

Thus, F is a threshold value which sets the speedup bound for the unroll factor ($\mathrm{u_s}$). The speedup bound is defined as:

$$\mathrm{u_s} = \min(u) \quad \text{such that} \quad \Delta S(u+1, u) < F \cdot Area_{(K)}.$$

Note that when the analyzed kernel is the only one running in hardware, it might make sense to unroll as much as possible, given the area and memory bounds ($\mathrm{u_a}$ and $\mathrm{u_m}$), as long as there is no performance degradation. In this case, we set $F = 0$ and $\mathrm{u_s} = \mathrm{u_m}$.

Local optimal values for the unroll factor u may appear when u is not a divisor of N, but $u+1$ is. To avoid this situation, as S is a monotonic increasing function for $u < \mathrm{u_m}$, we add another condition for $\mathrm{u_s}$: $\Delta S(\mathrm{u_s}+2, \mathrm{u_s}+1) < F \cdot Area_{(K)}$.

By using (4) and (5) in (6) and the notations

$$x = \frac{T_{\text{c}} + \min(T_{\text{r}}, T_{\text{w}})}{(\max(T_{\text{r}}, T_{\text{w}}) + T_{\text{sw}}) \cdot N} \quad \text{and} \quad y = \frac{T_{\text{sw}} + T_{K(\text{sw})}}{\max(T_{\text{r}}, T_{\text{w}}) + T_{\text{sw}}},$$

the total speedup is computed by: $$S_{\text{loop}}(u) = \frac{y}{1 + x \cdot N(u)} \qquad (8)$$

On the base of (8) and given the fact that the maximum unrolling factor is known – being the number of iterations N –, binary search can be used to compute in $O(\log N)$ time the value of u_s that satisfies the conditions $\Delta S(\text{u}_\text{s}+1, \text{u}_\text{s}) < F \cdot Area_{(K)}$ and $\Delta S(\text{u}_\text{s}+2, \text{u}_\text{s}+1) < F \cdot Area_{(K)}$.

Integrated constraints. In the end, speedup, area consumption and memory accesses need to be combined in order to find the feasible unroll factor, given all constraints. If $\text{u}_\text{s} < \min(\text{u}_\text{a}, \text{u}_\text{m})$, then the optimal unroll factor is $\min(u)$ such that $\text{u}_\text{s} < u \leq \min(\text{u}_\text{a}, \text{u}_\text{m})$; else, we choose it as $\max(u)$ such that $u \leq \min(\text{u}_\text{a}, \text{u}_\text{m})$.

5 Experimental results

The purpose of this section is to illustrate the presented method which computes automatically the unroll factor taking into account the area constraints and profiling information. The result and also the performance depend on the kernel implementation. The order of magnitude of the achieved speedup in hardware compared to software is not relevant for the algorithm, although it influences its output. Instead, we analyze the relative speedup obtained by running multiple instances of the kernel in parallel, compared to running a single one.

The loop from Fig. 3, containing the DCT kernel (2-D integer implementation), was extracted from MPEG2 encoder multimedia benchmark and executed on the VirtexII Pro board. We used the following parameters: $width = 64$, $height = 64$, $block\_count = 6$ (the picture size is 64×64, leading to $N = 96$ iterations).

```
for ($j = 0, k = 0$; $j$ <height; $j$=$j$+16) {
    for ($i = 0$; $i$ <width; $i$=$i$+16 ) {
        for ($n = 0$; $n$ <block_count; $n$++ ) {
            /*Compute parameters for K()*/
            CPar ($n$, $i$, $j$, $k$, $blocks$);
            /*Kernel function*/
            DCT ($blocks$[$k$*block_count+$n$]);
        }
    }
}
```

Fig. 3. MPEG2 loop with DCT kernel.

The DCT implementation operates on 8×8 memory blocks, therefore one kernel performs 64 memory reads and 64 memory writes. The memory blocks in different iterations do not overlap, thus there are no data dependencies and the first assumption in Section 3 holds.

The VHDL code was **automatically** generated with DWARV [3] tool and synthesized with Xilinx XST tool of ISE 8.1; it is not optimized, each memory access and each loop are synchronization points. One instance of the DCT kernel uses 12% of the total available area on VirtexII Pro. The execution times measured using the PowerPC timer registers presented in Table 1 are for one DCT instance (including the parameter transfer using exchange registers).

Table 1. Initial execution time (cycles)

	Hardware	Software	Percent	Speedup
T_K	37 278	106 626	34.96%	2.86
T_{par}	5 292	5 292	100%	-
T_{loop}	4 093 308	10 751 868	38.07%	2.63

We used the following notations: (i) T_K - the number of cycles for one instance of the DCT kernel; (ii) T_{par} - the number of cycles for $CPar()$; (iii) T_{loop} - the number of cycles for the loop nest.

The experiment was performed with one instance of the kernel running on the FPGA. We extrapolated these results for all possible unroll factors, computing the number of cycles for software and hardware execution of the kernel, and also for the loop nest. We observe that the theoretical (computed) execution time (in cycles) for the loop nest (T_{loop}) with the kernel executed in software does not depend on the unroll factor. Comparing with the measured execution time, there is an error of approx 0.072%, due to not taking into account the loop overhead; this error is negligible. Next, we compute the unroll factor applying the method described in Section 4.

Area. The upper bound that satisfies the area constraints is:

$$\mathrm{u_a} = \left\lfloor \frac{Area_{(total)} - Area_{(Molen)}}{Area_{(DCT)} + Area_{(interconnect)}} \right\rfloor = 8.$$

Memory accesses. For the considered implementation, the shared memory has an access time of 3 cycles for reading and storing the value into a register and 1 cycle for writing a value to memory; since there are 64 memory reads and 64 memory writes, $\min(T_{\mathrm{r}}, T_{\mathrm{w}}) = 64$ cycles. The computation time is $T_{\mathrm{c}} = T_{K(\mathrm{hw})} - (T_{\mathrm{r}} + T_{\mathrm{w}}) = 37\,022$ cycles. Using these values in (1), $\Rightarrow \mathrm{u_m} = 579$.

Speedup. To compute the speedup limit $\mathrm{u_s}$, we use the data from Table 1. Thus, $T_{\text{par}} = 5\,292$, $T_{K(\text{sw})} = 106\,626$, $T_{\mathrm{c}} = 37\,022$, $\max(T_{\mathrm{r}}, T_{\mathrm{w}}) = 192$, $N = 96$, then $x \approx 0.07$ and $y \approx 20.9$. According to (6), $S_{\text{loop}}(u) \approx \dfrac{20.9}{1 + N(u) \cdot 0.07}$.

Figure 4 presents the speedup for different unroll factors. One is the speedup at kernel level, and the second at loop level.

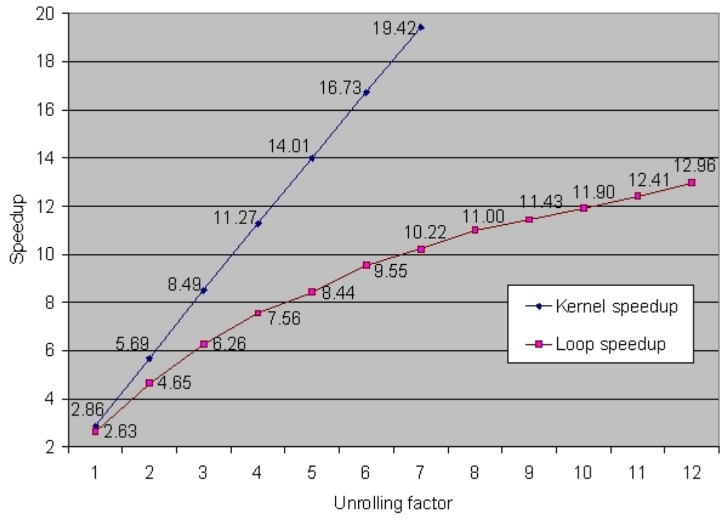


Fig. 4. Speedup obtained with loop unrolling

Assuming that we are interested in a relative speedup increase greater than the area increase ($\Delta S(u+1,u) > Area_{(K)}$ and $\Delta S(u+2,u+1) > Area_{(K)}$) for two consecutive unroll factors, $\Rightarrow \mathrm{u_s} = 6$.

Integrated constraints. The condition $\mathrm{u_s} < min(\mathrm{u_a}, \mathrm{u_m})$ is satisfied, meaning that $u = 6$, leading to a loop speedup of 9.55 and 72% area utilization of the VirtexII Pro total area.

6 Conclusion and future work

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

One of the main benefits of this algorithm is that it can be used to improve performance even when given an already optimized VHDL implementation of the kernel, if there are enough resources available (for instance, when moved to a different platform). Different results will be obtained for different kernel implementations, depending on how much optimized they are.

The presented method takes into account the area constraints when running multiple applications on the reconfigurable hardware, but not the memory constraints for this case. This will be addressed in future work. However, as our approach demonstrates the potential for significant performance improvement (experimental results for DCT show a speedup with a factor of 9.55, for an automatically generated VHDL implementation of the kernel), we plan to extend it by combining loop unrolling with pipelining and considering also transfers from a slow memory (DRAM).

References

1. Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K., Kuzmanov, G., Panainte, E.M.: The MOLEN Polymorphic Processor. IEEE Transactions on Computers (October 2004) 1363–1375
2. Panainte, E.M., Bertels, K., Vassiliadis, S.: The PowerPC Backend Molen Compiler. the 14th International Conference on Field-Programmable Logic and Applications (FPL'04) (August 2004) 434–443
3. Yankova, Y.D., Kuzmanov, G., Bertels, K., Gaydadjiev, G., Lu, J., Vassiliadis, S.: DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator. the 17th International Conference on Field Programmable Logic and Applications (FPL'07) (August 2007) 697–701
4. Guo, Z., Buyukkurt, B., Najjar, W., Vissers, K.: Optimized Generation of datapath from C codes for FPGAs. DATE '05: Proceedings of the conference on Design, Automation and Test in Europe (March 2005) 112–117
5. Gupta, S., Dutt, N., Gupta, R., Nicolau, A.: 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 (February 2004) 114–119
6. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: 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 (March 2003) 296–301
7. Cardoso, J.M.P., Diniz, P.C.: Modeling loop unrolling: approaches and open issues. the 4th International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS'04) (July 2004) 224–233
8. Weinhardt, M., Luk, W.: Pipeline vectorization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (February 2001) 234–248
9. Liao, J., Wong, W.F., Mitra, T.: A model for hardware realization of kernel loops. the 13th International Conference on Field-Programmable Logic and Applications (FPL'03) (September 2003) 334–344
10. Banerjee, S., Bozorgzadeh, E., Dutt, N.: PARLGRAN: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures. In: ASP-DAC'06: Proceedings of the 2006 conference on Asia South Pacific design automation. (January 2006) 491–496