LOOP UNROLLING AND SHIFTING FOR RECONFIGURABLE ARCHITECTURES

Ozana Silvia Dragomir, Todor Stefanov, and Koen Bertels

Email: O.S.Dragomir@tudelft.nl
Computer Engineering, EEMCS, TU Delft, The Netherlands

ABSTRACT

Loops are an important source of optimization. In this paper, we propose a new technique for optimizing loops that contain kernels mapped on a reconfigurable fabric. We assume the Molen machine organization and programming paradigm as our framework. The method we propose extends our previous work on loop unrolling for reconfigurable architectures by combining unrolling with shifting to relocate the function calls contained in the loop body such that in every iteration of the transformed loop, software functions (running on GPP) execute in parallel with multiple instances of the kernel (running on FPGA). The algorithm is based on profiling information about the kernel's execution times on GPP and FPGA, memory transfers and area utilization. In the experimental part, we apply this method to a loop nest extracted from MPEG2 encoder containing the DCT kernel. The achieved speedup is 19.65x over software execution and 1.8x over loop unrolling.

1. INTRODUCTION

Loops are an important source for optimization improvement, as many of the kernels in today's applications (audio, video, image processing, compression, etc) perform computations inside loops. The case we address in our research is when hardware-mapped kernels exist in the loop body. Assuming the Molen machine organization [1] as our framework, we focus on applying existing loop optimizations to such loops, with the purpose of parallelizing applications such that multiple kernel instances run in parallel on the reconfigurable hardware, while there is also the possibility of concurrently executing code on the GPP.

The contributions of this paper are: a) a method to automatically determine the unroll factor which gives the best performance for a loop transformed with shifting and unrolling, based on profile information about area consumption, memory transfers, and execution times; b) experimental results for the DCT (Discrete Cosine Transformation) kernel showing that the best performance is achieved when the method is applied with unroll factor 8, which leads to a speedup of 19.56x and improvement of 1.8x over loop unrolling alone (see [11]), while the area consumption is 96%.

The rest of this paper is organized as follows. Section 2 introduces the background and related work. In Section 3, we present the problem statement and the target application. In Section 4, we propose our algorithm and prove that it gives better results than our previous work. We illustrate this method and show 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)[1] project. The DWB is a semi-automatic toolchain platform which targets the Molen polymorphic machine organization [1], supporting the entire design process. In the first stage, profiling and cost estimation are performed and kernels are identified. After performing the appropriate transformations by collapsing the identified kernels on `set/execute` nodes, the compiler [2] generates the executable file, replacing and scheduling function calls to the kernels implemented in hardware with specific instructions for hardware reconfiguration and execution, according to the Molen programming paradigm. The DWARV automatic hardware generator [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. They use different loop transformations (unrolling, pipelining, etc) to exploit parallelism and speedup the kernel. Our approach is different, as we do not aggressively optimize the kernel implementation, but focus on the optimization of the application for any hardware implementation, by executing multiple kernel instances in parallel. In [11], we presented an algorithm that automatically determines the optimal unroll factor for a loop containing a hardware mapped kernel inside. [5] and [10] use shifting to expose loop parallelism and then to compact the loop by scheduling multiple operations to execute in parallel. We use the same basic idea, but at a functional level, in order to be able to parallelize the execution of software and hardware functions.

This work is supported by the FP6 EU project hArtes, with integrated project number 035143

[1] http://ce.et.tudelft.nl/DWB/

978-1-4244-1961-6/08/$25.00 ©2008 IEEE.

3. PROBLEM STATEMENT

In many real life applications, most of the execution time is spent in loops. Traditional loop transformations (such as loop unrolling, software pipelining, loop shifting, loop fusion, etc) can be applied successfully for maximizing the parallelism inside the loop and improving the performance. The applications we target in our work contain kernels inside loops. One challenge we address is to use the above mentioned loop transformations to improve the performance of such loops.

In this paper, we extend the work presented in [11] by combining loop unrolling with loop shifting. In our research, **loop shifting** means moving a function from the beginning of the loop body to the end. We use loop unrolling to expose the parallelism at hardware level (e.g., run multiple kernels in parallel) and loop shifting to eliminate the data dependencies between software and hardware functions, allowing concurrent execution on the GPP and FPGA (as illustrated in Fig. 3).

The problem statement is: find automatically the unroll factor u for a loop (loop nest) containing both code for GPP and one or more kernels to be executed in hardware, such that u identical instances of the kernel(s) run in parallel while the rest of the code in the loop body executes on the GPP, leading to the best performance allowed by the area and memory constraints. The method proposed in this paper addresses this problem, given a C implementation of the target application and a VHDL implementation of the kernel. The performance of this method is compared to that presented in [11], where only loop unrolling is used.

The target architecture is Molen [1], which allows running multiple kernels/applications at the same time on the reconfigurable hardware. As in our previous work, the unroll factor is computed (at compile time) taking into consideration profiling information about memory transfers, execution times for the kernel in hardware and in software (in GPP cycles), area requirements for the kernel, and also memory bandwidth and available area (other kernels may be configured at the same time). Thus, the hardware configuration at a certain time influences the algorithm's output. Note that we consider that the execution time in hardware is constant for all kernel instances, independent on the input data.

The main benefits of this algorithm are that it can be integrated in an automatic toolchain and use any available hardware implementation of the kernel. The performance of the application can thus be improved also when using already optimized kernels. Also, the performance achieved with unrolling and shifting is better than when applying only unrolling. The same method can be easily extended to multiple kernels inside the loop – if they have similar input sets – by taking into account the maximum execution time from the different hardware kernels. The assumptions regarding the application and framework are summarized in Table 1.

Table 1. General and Molen-specific assumptions

Loop nest ⋆ no data dependencies between different iterations; ⋆ loop bounds are known at compile time; ⋆ loops are perfectly nested;
Memory accesses ⋆ memory reads in the beginning, memory writes in the end; ⋆ on-chip memory shared by the GPP and the Custom Computing Units (CCUs) is used for program data; ⋆ all necessary data are available in the shared memory; ⋆ all transactions on shared memory are performed sequentially; ⋆ kernel's local data are stored in the FPGA's local memory, not in the shared memory;
Area & placement ⋆ shape of design is not considered; ⋆ placement is decided by a scheduling algorithm such that the configuration latency is hidden; ⋆ interconnection area needed for CCUs grows linearly with the number of kernels.

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

Fig. 1. Loop containing a kernel call

Motivational example. Throughout the paper, we will use the sample code from Fig. 1. It consists of a loop with two functions – one function ($CPar$) is executed always on the GPP, and the other is the application kernel (K). The execution time for $CPar$ is much smaller than the execution time for K on the GPP. Data dependencies exist between $CPar(i)$ and $K(i)$ in each iteration i, but **not** between $CPar(i)$ and $K(j)$ or $K(i)$ and $K(j)$ for any iterations i and j, $i \neq j$. The example has been extracted from the MPEG2 encoder multimedia benchmark, where the kernel K is DCT and $CPar$ is some code which computes the parameters for the kernel to be executed.

4. PROPOSED METHODOLOGY

In this section, we present the loop optimization based on unrolling and shifting, which uses our previous work presented in [11]. The inputs for our method are the profiling information (execution time and number of memory transfers) and area usage for one instance of the kernel. We will also demonstrate that combining unrolling and shifting gives better results than only unrolling. Figure 2 presents a simplified case of applying the unrolling method, when $N \textit{ mod } u = 0$.

This method is extended in Fig. 3 by shifting the software part of the loop to the end of the loop body, such that in each iteration u sequential executions of the function $CPar()$ are executed in parallel with u identical kernel instances. The loop has one iteration less than in the case when only unrolling in applied, as the first u calls of $CPar()$

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

Fig. 2. Loop unrolled with a factor u

```
CPar (0, blocks); ... CPar (u-1, blocks);
for (i=u; i<N; i+=u) {
    #pragma parallel /*u instances of K || with the GPP*/
     K (blocks[i-u]);
      ...
     K (blocks[i-1]);
     {/* sequential execution in software */
      CPar (i+0, blocks); ... CPar (i+u-1, blocks);
     }
    #end parallel
}
#pragma parallel
 K (blocks[N-u]);
  ...
 K (blocks[N-1]);
#end parallel
```

Fig. 3. Loop unrolled and shifted with a factor u

are executed before the loop (prologue) and the last u kernel instances are executed after the loop (epilogue).

Note that this optimization may be applied as presented in this paper, taking into account that there are data dependencies between $CPar(i)$ and $K(i)$, but there are **no** data dependencies between $CPar(i)$ and $K(j)$ or $K(i)$ and $K(j)$ $(i \neq j)$.

Next we show briefly how the final selection of the unroll factor depends on area and memory constraints (consult [11] for more details).

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:

$$\mathrm{u_a} = \left\lfloor \frac{Area_{(available)}}{Area_{(K)} + Area_{(interconnect)}} \right\rfloor \quad (1)$$

where: (i) $Area_{(available)}$ is the available area;
(ii) $Area_{(interconnect)}$ is the area necessary to connect one kernel with the rest of the hardware design;
(iii) $Area_{(K)}$ is the area utilized by one kernel instance.

Memory accesses. For many applications, the memory bandwidth is an important bottleneck in achieving the theoretical maximum parallelism. Considering that T_{r}, T_{w} and T_{c} are the times for memory read, write, and computation on hardware for kernel K, as determined from the profiling information, the total time for running K in hardware is $T_{\mathrm{r}} + T_{\mathrm{w}} + T_{\mathrm{c}}$. The following notations are used:

$$T_{\mathrm{min(r,w)}} = \min(T_{\mathrm{r}}, T_{\mathrm{w}});\; T_{\mathrm{max(r,w)}} = \max(T_{\mathrm{r}}, T_{\mathrm{w}}) \quad (2)$$

The performance increases until the computation is fully overlapped by the memory transfers performed by the kernel instances running in parallel, and we denote by $\mathrm{u_m}$ the unroll factor where this case happens. Then, $\mathrm{u_m}$ sets another bound for the degree of unrolling on the reconfigurable hardware. Further increase of the unroll factor gives a converse effect when the computation stalls occur due to waiting for the memory transfers to finish. Without reducing the generality of the problem for most of the applications, we assume that the memory reads are performed in the beginning and memory writes in the end.[2] Then, the execution time for u instances of K in hardware can be expressed as:

Table 2. Notations

N	initial number of iterations (before unrolling);
T_{p}	number of cycles for one instance of the software function (the function that is always executed by the GPP - in our example, the $CPar$ function);
$T_{K(\mathrm{hw})}(u)$	number of cycles for u instances of K() running in hardware, defined in (3) (only the case $u \leq \mathrm{u_m}$);
$T_{\mathrm{loop(sw)}}$	number of cycles for the loop nest executed completely in software;
$T_{\mathrm{shift}}(u)$	number of cycles for the transformed loop nest with u instances of K() running in hardware;
$S_{\mathrm{shift}}(u)$	the speedup at loop level.

$$T_{K(\mathrm{hw})}(u) = \begin{cases} T_{\mathrm{c}} + T_{\mathrm{min(r,w)}} + u \cdot T_{\mathrm{max(r,w)}}, & \text{if } u \leq \mathrm{u_m} \\ u \cdot (T_{\mathrm{min(r,w)}} + T_{\mathrm{max(r,w)}}), & \text{if } u > \mathrm{u_m} \end{cases} \quad (3)$$

This shows that for $u > \mathrm{u_m}$, the speedup at kernel level $S_K = \frac{T_{K(\mathrm{hw})}(u)}{u \cdot T_{K(\mathrm{hw})}(1)}$ is constant, thus it is not worth to unroll more. The performance increases with the unroll factor while the following condition is satisfied:

$$T_{\mathrm{c}} + T_{\mathrm{min(r,w)}} + u \cdot T_{\mathrm{max(r,w)}} < u \cdot (T_{\mathrm{min(r,w)}} + T_{\mathrm{max(r,w)}}) \quad (4)$$

Thus, the memory bound is derived:

$$u \leq \mathrm{u_m} = \left\lfloor \frac{T_{\mathrm{c}}}{T_{\mathrm{min(r,w)}}} \right\rfloor + 1 \quad (5)$$

Speedup. We use the notations presented in Table 2. Note that the case $u > u_m$ is not considered because there is no speedup increase for the hardware kernels.

$T_{\mathrm{loop(sw)}}$ does not depend on the unroll factor:

$$T_{\mathrm{loop(sw)}} = \left(T_{\mathrm{p}} + T_{K(\mathrm{sw})}\right) \cdot N \quad (6)$$

The speedup at loop nest level is:

$$S_{\mathrm{shift}}(u) = \frac{T_{\mathrm{loop(sw)}}}{T_{\mathrm{shift}}(u)}, \quad (7)$$

where the total execution time ($T_{\mathrm{shift}}(u)$) for a loop transformed with unrolling and shifting can be expressed like:

$$T_{\mathrm{shift}}(u) = T_{\mathrm{prolog}}(u) + T_{\mathrm{body}}(u) + T_{\mathrm{epilog}}(u), \quad (8)$$

[2] Assuming that memory reads are performed in the beginning and memory writes in the end is actually the worst case. Depending on the hardware implementation, the real threshold value for the unroll factor regarding memory transfers might be more permissive.

We use the notations:
(i) $T_{\text{prolog}}(u)$ is the time for the loop prologue:

$$T_{\text{prolog}}(u) = u \cdot T_{\text{p}} \tag{9}$$

(ii) $T_{\text{body}}(u)$ is the time for the transformed loop body, consisting of parallel hardware and software execution:

$$T_{\text{body}}(u) = (\lfloor N/u \rfloor - 1) \cdot \max\left(u \cdot T_{\text{p}},\ T_{K(\text{hw})}(u)\right) \tag{10}$$

(iii) $T_{\text{epilog}}(u)$ is the time for the loop epilogue.

For the simplified case in Fig. 3, the epilogue consists of the hardware parallel execution of u kernel instances:

$$T_{\text{epilog}}(u) = T_{K(\text{hw})}(u) \tag{11}$$

For the general case where u is not a divisor of N, it contains also the execution of the remainder instances of the software function and hardware kernel. We denote by R the remainder of the division of N by u: $R = N - u * \lfloor N/u \rfloor, 0 \leq R < u$. We define $T_{K(\text{hw})}(R)$ as:

$$T_{K(\text{hw})}(R) = \begin{cases} 0, & R = 0 \\ T_c + T_{\text{min(r,w)}} + R \cdot T_{\text{max(r,w)}}, & R > 0 \end{cases} \tag{12}$$

Then,

$$T_{\text{epilog}}(u) = \max\left(R \cdot T_{\text{p}},\ T_{K(\text{hw})}(u)\right) + T_{K(\text{hw})}(R) \tag{13}$$

In order to compute $T_{\text{body}}(u)$ from (10) and $T_{\text{epilog}}(u)$ from (13), there are different cases depending on the relations between T_{p}, T_{c}, $T_{\text{min(r,w)}}$ and $T_{\text{max(r,w)}}$.

a) $\mathbf{T_p \leq T_{max(r,w)}}$

The meaning of the relation above is that inside a loop iteration the execution time for the software part increases slower with the unroll factor than the hardware part. The total execution time for the loop will then be determined by the execution time of the hardware part, hiding the software execution completely. For one loop iteration:

$$\max\left(u \cdot T_{\text{p}},\ T_{K(\text{hw})}(u)\right) = T_{K(\text{hw})}(u) \tag{14}$$

and for the epilogue (since $R < u$):

$$\max\left(R \cdot T_{\text{p}},\ T_{K(\text{hw})}(u)\right) = T_{K(\text{hw})}(u) \tag{15}$$

By substituting (15) in (13), (14) in (10), and (9), (10), (13) in (8), the total execution time for the shifted loop is:

$$T_{\text{shift}}(u) = u \cdot T_{\text{p}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(R) \tag{16}$$

b) $\mathbf{T_p > T_{max(r,w)}}$

If $T_{\text{p}} > T_{\text{max(r,w)}}$, then the execution time in software increases faster with the unroll factor (u) than the execution time in hardware; for values of u greater than a threshold value U_1, the execution on the reconfigurable hardware in one iteration will take less time than the execution on GPP. The execution time for one iteration is:

$$\max\left(u \cdot T_{\text{p}},\ T_{K(\text{hw})}(u)\right) = \begin{cases} T_{K(\text{hw})}(u), & u < U_1 \\ u \cdot T_{\text{p}}, & u \geq U_1 \end{cases} \tag{17}$$

If $u \leq U_1$ then:

$$u \cdot T_{\text{p}} \leq T_{\text{c}} + T_{\text{min(r,w)}} + u \cdot T_{\text{max(r,w)}} \tag{18}$$

This determines the threshold value U_1 as:

$$U_1 = \left\lceil \frac{T_{\text{c}} + T_{\text{min(r,w)}}}{T_{\text{p}} - T_{\text{max(r,w)}}} \right\rceil \tag{19}$$

Intuitively, we expect to find the unroll factor that gives the smallest execution time and thus the biggest speedup in the close vicinity of U_1 (we define the close vicinity as the set $\{U_1 - 1, U_1, U_1 + 1\}$), depending if any of these values is a divisor of N or not, where the software and hardware execute concurrently in approximatively the same amount of time.

Since the computation of $T_{\text{shift}}(u)$ involves the *floor* ($\lfloor\rfloor$), *max* and *remainder* functions, it is not possible to give a formula which computes the value of u that minimizes $T_{\text{shift}}(u)$ without knowing the exact values of the parameters (T_{p}, T_{c}, T_{w}, T_{r}). However, to continue our analysis, we can express $T_{\text{shift}}(u)$ as a choice function between 3 functions, as follows. We use the notations:

$$\begin{aligned} T_1 &= u \cdot T_{\text{p}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) & (20) \\ T_2 &= \lfloor N/u \rfloor \cdot u \cdot T_{\text{p}} + T_{K(\text{hw})}(u) & (21) \\ T_3 &= N \cdot T_{\text{p}} & (22) \\ T_i^{'} &= T_i + T_{K(\text{hw})}(R), \quad 1 \leq i \leq 3 & (23) \end{aligned}$$

Then, $T_{\text{shift}}(u)$ is:

$$T_{\text{shift}}(u) = \begin{cases} T_1^{'}, & (u < U_1) \\ T_2^{'}, & (u \geq U_1) \,\&\, (T_{K(\text{hw})}(u) \geq R \cdot T_{\text{p}}) \\ T_3^{'}, & (u \geq U_1) \,\&\, (T_{K(\text{hw})}(u) < R \cdot T_{\text{p}}) \end{cases} \tag{24}$$

T_3 defined above is constant, regardless of the unroll factor, as it depends only on the software execution time: $T_3 = N \cdot T_{\text{p}}$. Because the execution time is always computed by taking the maximum between the software and the hardware times when parallel execution is involved, it means that $T_{\text{shift}}(u) \geq N \cdot T_{\text{p}} + T_{K(\text{hw})}(R)$ for all $u > U_1$. Therefore, the only possible cases when $T_{\text{shift}}(u)$ may be smaller than $T_{\text{shift}}(U_1)$ are some of those when the remainder of U_1 is greater than the remainder of u ($u > U_1$). We analyze the following cases:

- U_1 is a divisor of N: then, $T_{\text{epilog}}(U_1) = T_{K(\text{hw})}(U_1)$ and $T_{\text{shift}}(U_1) \leq T_{\text{shift}}(u), \forall u > U_1$;
- U_1 is **not** a divisor of N:
  Then, $T_{\text{epilog}}(U_1) = T_{K(\text{hw})}(U_1) + T_{K(\text{hw})}(N \bmod U_1)$.

  For $u > U_1$ such that u is a divisor of N, $T_{\text{epilog}}(u) = T_{K(\text{hw})}(u)$ and $T_{\text{epilog}}(U_1) - T_{\text{epilog}}(u) \approx T_{\text{c}}$ (we consider $T_{\text{min(r,w)}}$ and $T_{\text{max(r,w)}}$ to be negligible when compared to T_{c}, otherwise the memory bound u_{m} will be very small, and high values for the unroll factor will not be permitted even if the theoretical speedup would be very significant).

  For this case, we consider that if $\lfloor N/u \rfloor \geq 10$ (the number of iterations is greater than or equal to 10), then the speedup increases with less than 10% and is not significant. When the iteration number is less than 10, than the speedup increase may be significant and the algorithm computes the total execution time and the speedup for all the divisors of N which satisfy the area and memory constraints.

Integrated constraints. In the end, we choose the unroll factor that maximizes the speedup, taking into account also the area consumption and memory transfers.

Let u_{min} be $u_{\text{min}} = \min(u_{\text{a}}, u_{\text{m}})$. If $T_{\text{p}} > T_{\text{max(r,w)}}$ and the unroll factor threshold U_1 satisfies the memory and area constraints ($U_1 < u_{\text{min}}$), the algorithm looks for the best execution time in the close vicinity of U_1. If the unroll factor that gives the best execution time is not a divisor of N, the algorithm checks all the divisors of N between U_1 and u_{min} and computes the execution time and the speedup. The selected unroll factor is the one that gives at least 10% speedup improvement over the speedup achieved for U_1.

If U_1 does not satisfy the memory and area constraints or if $T_{\text{p}} \leq T_{\text{max(r,w)}}$, the unroll factor is chosen as:

$$U = \min_{u} \left\{ u \in \mathbb{Z} | u \leq u_{\text{min}} \wedge S_{\text{shift}}(u) = \max_{i \leq u_{\text{min}}} \left(S_{\text{loop}}(i) \right) \right\} \tag{25}$$

Comparison with loop unrolling. We compare the execution time $T_{\text{shift}}(u)$ from (24) with the execution time obtained by applying only loop unrolling ($T_{\text{loop(hw)}}$), as presented in [11].

$$T_{\text{loop(hw)}} = N \cdot T_{\text{p}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(R) \tag{26}$$

$$T_{\text{loop(hw)}} - T_{\text{shift}}(u) = \begin{cases} (N-u) \cdot T_{\text{p}}, & \text{if } u < U_1 \\ R \cdot T_{\text{p}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) - \\ \max(R \cdot T_{\text{p}},\, T_{K(\text{hw})}(u)), & \text{if } u \geq U_1 \end{cases} \tag{27}$$

For $u = N$, $T_{\text{loop(hw)}} = T_{\text{shift}}(u)$, else $T_{\text{loop(hw)}} > T_{\text{shift}}(u)$. This means that, for the same unroll factor, the speedup obtained by combining unrolling and shifting will be bigger than the one achieved with unrolling only.

Table 3. Initial execution time (cycles)

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

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

Fig. 4. MPEG2 loop with DCT kernel.

5. EXPERIMENTAL RESULTS

In this section, we illustrate the method presented in Section 4, which computes automatically the unroll factor that gives the best performance when applying unrolling and shifting, taking into account the area constraints and profiling information. As in the case of simple unrolling, the result and the performance depend on the kernel implementation and the order of magnitude of the achieved speedup is not relevant for the algorithm. The main benefit of this method is that it exploits the hardware capabilities more efficiently and gives better performance than only loop unrolling (presented in [11]), while it needs the same input data. A comparison with previous achieved results is performed.

The loop nest presented in Fig. 4 containing the DCT kernel (2-D integer implementation) was extracted from the MPEG2 encoder multimedia benchmark and executed on the Virtex II Pro board. The following parameters were used for the execution: $width = 64$, $height = 64$, $block_count = 6$ (the picture size is 64×64, leading to $N = 96$ iterations).

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 for DCT was **automatically** generated with DWARV [3] tool. Synthesis results (using Xilinx XST tool of ISE 8.1) show that one instance of the DCT kernel uses 12% of the total available area on VirtexII Pro. The execution times with one instance of DCT running on GPP and then on FPGA were measured using the PowerPC timer registers. The times are presented in Table 3, using the notations: (i) T_K - the number of cycles for one instance of the DCT kernel; (ii) T_{p} - the number of cycles for $CPar()$; (iii) T_{loop} - the number of cycles for the loop nest.

Next, we apply the method described in Section 4 to compute the unroll factor.

Area. The upper bound that satisfies the area constraints computed using (1) is $u_{\text{a}} = 8$.

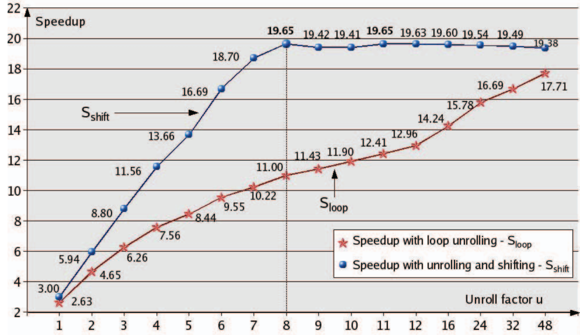

Fig. 5. Speedup obtained with loop unrolling+shifting

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, $T_{\text{min(r,w)}} = 64$ and $T_{\text{max(r,w)}} = 192$ cycles. The computation time is $T_{\text{c}} = T_{K(\text{hw})} - (T_{\text{r}} + T_{\text{w}}) = 37\,022$ cycles. Using these values in (5), $\Rightarrow \text{u}_{\text{m}} = 579$.

Speedup. Since $T_{\text{max(r,w)}} < T_{\text{P}}$, we need to compute the threshold value U_1 according to (19). For values of the unroll factor u greater than U_1, the parallel execution of the u kernel instances in hardware will finish faster than the software execution which is performed in parallel.

Using in (19) the values of $T_{\text{min(r,w)}}$, $T_{\text{max(r,w)}}$ and T_{c} computed above, $U_1 = 8$. By analyzing the total execution time for all possible unroll factors, we find the global optimum value for $u = U_1$, but also for $u = 11$ and $u = 22$. We observe that the execution times and the speedup (see Fig. 5) for all factors $u > 8$ are within 10% of the optimum.

Integrated constraints. As $\min(u_{\text{a}}, u_{\text{m}}) = 8$ and the execution time has a global optimum for $u = 8$, it is easy to conclude that the chosen unroll factor is 8, leading to a speedup of 19.65x and area consumption of 96%.

Comparison with loop unrolling. Figure 5 presents the speedup obtained for different unroll factors, for both unrolling and unrolling+shifting. For the chosen factor ($u = 8$), the speedup improvement with the new technique is 1.8x compared to previous results. As it can be seen from the graph, the theoretical performance for the maximum unroll factor tends to be the same for both techniques, as $S_{\text{loop}}(u)$ is a monotonously increasing function and $S_{\text{shift}}(u)$ oscillates within 10% of the maximum value (19.65x) for $u \geq 8$.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a new technique that combines loop unrolling and loop shifting, suitable for reconfigurable architectures. Using profiling information about area utilization, memory transfers and execution times in software and in hardware for a certain kernel implementation, the presented algorithm automatically computes the most suitable unroll factor which allows parallel execution of several kernel instances in hardware, concurrently with software execution. This model can be easily extended to the case when more than one kernel are called from inside the same loop with similar input sets, by taking into account the maximum of the execution times for the hardware kernels. The implementation of this algorithm in the compiler decreases the time for design-space exploration and exploits the architectural capabilities more efficiently.

The performance achieved when applying this method is better than when applying only loop unrolling. Experimental results show a speedup increase with a factor of 1.8x, while the total speedup (compared to pure software execution) is 19.56x, with an area consumption of 96%.

As future work we consider gathering results for more applications and relaxing some of the assumptions regarding the loop and the memory accesses.

7. REFERENCES

[1] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, “The MOLEN Polymorphic Processor”, *IEEE Transactions on Computers*, pp. 1363–1375, Oct. 2004.

[2] E. M. Panainte, K. Bertels, and S. Vassiliadis, “The PowerPC Backend Molen Compiler”, *FPL ’04*, pp. 434–443, Aug.2004.

[3] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, “DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator”, *FPL ’07*, pp. 697–701, Aug.2007.

[4] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, “Optimized Generation of data-path from C codes for FPGAs”, *DATE ’05*, pp. 112–117, Mar. 2005.

[5] 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*, pp. 114–119, Feb.2004.

[6] 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*, Mar.2003.

[7] J. M. P. Cardoso and P. C. Diniz, “Modeling loop unrolling: approaches and open issues”, *SAMOS ’04*, pp. 224–233, July 2004.

[8] M. Weinhardt and W. Luk, “Pipeline vectorization”, *IEEE Transactions on CAD*, pp. 234–248, Feb.2001.

[9] J. Liao, W.-F. Wong, and T. Mitra, “A model for hardware realization of kernel loops”, *FPL ’03*, pp. 334–344, Sep.2003.

[10] A. Darte, and G. Huard, “Loop Shifting for Loop Compaction”, in *LCPC*, pp. 415–431, 1999

[11] O. S. Dragomir, E. Moscu-Panainte, K. Bertels, and S. Wong, “Optimal Unroll Factor for Reconfigurable Architectures”, in *ARC ’08*, Mar.2008.