Optimal Loop Unrolling and Shifting for Reconfigurable Architectures

OZANA SILVIA DRAGOMIR, TODOR STEFANOV, and KOEN BERTELS
TU Delft

In this article, we present a new technique for optimizing loops that contain kernels mapped on a reconfigurable fabric. We assume the Molen machine organization as our framework. We propose combining loop unrolling with loop shifting, which is used 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 computes the optimal unroll factor and determines the most appropriate transformation (which can be the combination of unrolling plus shifting or either of the two). This method 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 several kernels from loop nests extracted from real-life applications (DCT and SAD from MPEG2 encoder, Quantizer from JPEG, and Sobel's Convolution) and perform an analysis of the results, comparing them with the theoretical maximum speedup by Amdahl's Law and showing when and how our transformations are beneficial.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Loop optimizations, reconfigurable computing

ACM Reference Format:
Dragomir, O. S., Stefanov, T., and Bertels, K. 2009. Optimal loop unrolling and shifting for reconfigurable architectures. ACM Trans. Reconfig. Techn. Syst. 2, 4, Article 25 (September 2009), 24 pages. DOI = 10.1145/1575779.1575785. http://doi.acm.org/10.1145/1575779.1575785.

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. Loops are an important

This work is supported by the FP6 EU project hArtes, with integrated project number 035143.
Authors' address: O. S. Dragomir, T. Stefanov, and K. Bertels, Computer Engineering Laboratory, TU Delft, Mekelweg 4, 2628CD Delft, The Netherlands; email: {O.S.Dragomir, T.P.Stefanov, K.L.M.Bertels}@tudelft.nl.

source of performance improvement, as they represent or include kernels of modern real-life applications (audio, video, image processing, and so on).

The case we address in our research is when hardware-mapped kernels exist in the loop body. Assuming the Molen machine organization [Vassiliadis et al. 2004] 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 general purpose processor (GPP).

Optimal is defined in this paper as the largest feasible unroll factor, given area constraints, performance requirements, and memory access constraints, also taking into account that multiple kernels may be mapped on the reconfigurable hardware. The contributions of this article are as follows:

(a) We propose an algorithm to automatically compute the optimal unroll factor for a loop containing a kernel mapped on the reconfigurable hardware and select the most suitable transformation for the analyzed loop (unrolling, shifting, or a combination of them). The algorithm is based on profiling information about memory transfers, available area, and software/hardware execution times.

(b) We present experimental results for several loops containing well known kernels—DCT (discrete cosine transformation), Sobel's Convolution, SAD (sum of absolute differences), and Quantizer—comparing the achieved speedup with the theoretical maximum speedup by Amdahl's Law. We provide an analysis of the possible cases, showing when the proposed loop transformations are beneficial and when they cannot have much influence on the performance. Also we present a case study of DCT. We compare the speedup achieved by using an aggressively optimized hardware implementation of DCT with the speedup achieved by using an automatically generated VHDL code and our loop transformations.

The rest of this article is organized as follows. Section 2 introduces the background and related work. In Section 3, we present the problem statement. In Section 4, we propose our algorithm, and we illustrate it in Section 5 with several kernels of well-known applications and an analysis of the possible cases. Final conclusions are presented in Section 6.

2. BACKGROUND AND RELATED WORK

The work presented in this article is related to the Delft WorkBench (DWB) project.[1] The DWB is a semiautomatic toolchain platform for integrated hardware-software codesign in the context of custom computing machines (CCM), which targets the Molen polymorphic machine organization [Vassiliadis et al. 2004]. In the first stage of the design process, profiling and cost estimation are performed and kernels are identified. After performing

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

the appropriate transformations by collapsing the identified kernels on `set/execute` nodes, the Molen compiler 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 [Vassiliadis et al. 2003]. An automatic tool for hardware generation (DWARV [Yankova et al. 2007]) is used to transform the selected kernels into VHDL code targeting the Molen platform. The automated code generation is envisioned for fast prototyping and fast performance estimation during the design space exploration.

This article extends our previous work on loop unrolling [Dragomir et al. 2008a] and loop unrolling plus shifting [Dragomir et al. 2008b]. In the following section we will present the methodology for choosing the more suitable of the two transformations and the optimal unroll factor (which may be 1, if only loop shifting is used). In our previous work, the methods were illustrated only on the DCT kernel. In this article, the experimental section contains a larger set of empirical results. We also provide an extended analysis of the encountered cases, showing when our method is beneficial and for what reason.

Several 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 and accelerating kernel loops in hardware. Directly connected to our work are those of Guo et al. [2005], Weinhardt and Luk [2001] and Gupta et al. [2004], where hardware is generated after optimizing the kernel loops.

To be more specific, the work of Guo et al. [2005] is part of the ROCCC—C to hardware compilation project, whose objective is the FPGA-based acceleration of frequently executed code segments (loop nests). The ROCCC 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.

Weinhardt and Luk [2001] introduce us to pipeline vectorization, a method for synthesizing hardware pipelines based on software vectorizing compilers. In their approach, full loop unrolling, as well as loop tiling, and loop merging, are used to increase basic block size and extend the scope of local optimizations.

The work in Gupta et al. [2004] is part of the SPARK project and uses shifting to expose loop parallelism and then to compact the loop by scheduling multiple operations to execute in parallel. In that case, loop shifting is performed at low level, whereas we perform it at a high functional level. The shifted loops are scheduled and mapped on the hardware, as in the case of the projects presented previously.

Our approach differs, as we do not aggressively optimize the kernel implementation to improve the application's performance. Instead, we speed up the application by executing multiple kernel instances in parallel. The benefit of our approach is that it improves the performance irrespective of the kernel's hardware implementation.

Previous approaches in predicting the impact of loop unrolling include Liao et al. [2003] and Cardoso and Diniz [2004]. In Liao et al. [2003], the authors propose a model for the hardware realization of kernel loops. The compiler is used to extract certain key parameters of the analyzed loop. From these parameters, taking 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 factor to be used, but the main difference between our approaches is that their method does not consider parallel execution. Also, their model needs to be calibrated by running several transformed loops in order to be able to make a prediction about the frequency and thus, about the execution time.

In Cardoso and Diniz [2004], the authors propose a model to predict the impact of full loop unrolling on the execution time and on the number of required resources, without explicitly performing it. However, unroll-and-jam (unrolling one or more nested loops in the iteration space and fusing inner loop bodies together) is not covered. 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 balanced, efficient design. The estimation of needed resources for unrolled loops is performed simply by multiplying the resources for the loop body with the number of iterations, similar to the way we estimate the resource usage for multiple instances of the kernel. We apply the same strategy for estimating the time for the software part of the unrolled loop, but the time for the kernels running in parallel is determined taking into account the computational time and the memory transfers inside the kernel. They also have a complex formula for computing the execution time for the transformed loop (unrolled or pipelined), taking into account the loop overhead, the length of the pipeline, and the number of successive accesses to the same memory, but their research does not consider parallel execution, nor memory bandwidth constraints.

PARLGRAN Banerjee et al. [2006] 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:

—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;
—in their case, the task instances have identical area requirements but different workloads, which translates into different execution times (a task is split into several subtasks); in our algorithm, all instances have the same characteristics in both area consumption and execution time;
—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;
—they do not consider running the software and hardware in parallel.

3. PROBLEM STATEMENT

In many real life applications, loops represent an important source of optimization. A number of loop transformations (such as loop unrolling, software pipelining, loop shifting, loop distribution, loop merging, or loop tiling) can be successfully 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 we mentioned. We also keep in mind that there are loop transformations that are not beneficial in most compilers because of the large overhead that they introduce when applied at instruction level, but at a coarse level (i.e., function), they show a great potential for improving the performance.

In this article, we study the effects of applying loop unrolling versus loop unrolling plus shifting. In related work such as Gupta et al. [2004] and Darte and Huard [1999], loop shifting is performed at instruction level. In our research, loop shifting means moving a function from the beginning of the loop body to the end, while preserving the correctness of the program. 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 Figure 3).

The problem statement is the following: find the optimal transformation (unrolling or unrolling plus shifting) and unroll factor u which maximize the performance for a loop nest containing a kernel K, such that u identical instances of K run in parallel on the reconfigurable hardware. The method proposed in this article addresses this problem, given a C implementation of the target application and a VHDL implementation of the kernel. Our 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 (we assume no constraints regarding the placement of the kernel). Note that we consider that the execution time in hardware is constant for all kernel instances, independent of the input data.

We target the Molen framework, which allows multiple kernels/applications to run simultaneously on the reconfigurable hardware. Because of the reconfigurable hardware's flexibility, the algorithm's output depends on the hardware configuration at a certain time. The Molen architecture is based on the tightly coupled processor coprocessor paradigm. Within the Molen concept, a general purpose core processor (GPP) controls the execution and reconfiguration of a reconfigurable coprocessor. The Molen machine organization [Vassiliadis et al. 2004] has been implemented on a Virtex II Pro device [Xilinx Inc. 2007]. The memory design uses the available on-chip memory blocks of the FPGA; this memory is shared by the GPP and the reconfigurable processor. The resources consumed by the Molen implementation on the XC2VP30 chip are less than 2% [Kuzmanov et al. 2004].

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.

Table I. 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++) {

    /* Function that executes always on the GPP */
    do_SW (blocks, i, ...);

    /* Kernel function */
    K (blocks, i, ...);
}
```

Fig. 1. Loop containing a kernel call.

In this context, performance can be improved even when the kernel is already optimized. Our assumptions regarding the application and the framework are summarized in Table I.

Motivational example. Throughout the article, we will use the motivational example in Figure 1. It consists of a loop with two functions: do_SW—which is always executed on the GPP—and K, which is the application's kernel and will be executed on the reconfigurable hardware in order to speed up the application. Implicitly, the execution time for do_SW is smaller than the execution time of K on the GPP.

Note that the focus of this article is to present how the loop unrolling and loop shifting can be applied to loops containing only one kernel mapped on the reconfigurable hardware. Considering that there is just one kernel in the loop body, implies that the software code (do_SW) can be placed before the kernel (preprocessing), after the kernel (post-processing) or as both pre- and post-processing. In this article, we show only the formulas for the pre-processing case because the other two cases are similar and the formulas will not be much different. We consider the case of software code occurring in the middle of a loop as a case where a loop contains multiple kernels. This is not within the scope of this article, but a subject of our ongoing work. For these loops, a different approach is needed and sometimes it is more beneficial to split them

```
for (i=0; i<N; i+=u) {

    /* u instances of do_SW(), sequentially */
    do_SW (blocks, i+0, ...);
     ...
    do_SW (blocks, i+u-1, ...);

    /* u instances of K() in parallel */
    #pragma parallel
     K (blocks, i+0, ...);
      ...
     K (blocks, i+u-1, ...);
    #end parallel
}
```

Fig. 2. Parallelized loop after unrolling with factor *u*.

in small loops where unrolling and shifting can be applied as presented in this article.

In each iteration in our example, data dependencies between `do_SW` and K may exist. In order to be able to apply loop unrolling and run in parallel, multiple instances of the kernel, data dependencies between K(i) and K(j), for any iterations i and j, $i \neq j$, may not exist. For instance, `do_SW` can be the code that computes the parameters for the kernel instance to be executed in the same iteration. In order to perform loop shifting and then concurrently execute the code on the GPP and on the FPGA, one more constraint needs to be satisfied: there should be no data dependencies between `do_SW(`i`)` and K(j), for any iterations i and j, $i \neq j$.

The example in Figure 1 is a generalized version of a loop extracted from the MPEG2 encoder multimedia benchmark, where the kernel K is DCT. The `blocks` variable is written in `do_SW` and then read (or read+written) in K, thus there are data dependencies between `do_SW` and K in the same iteration.

4. PROPOSED METHODOLOGY

In this section, we present two techniques that can be applied for loops that contain hardware kernels inside. One technique consists of only loop unrolling, and the second technique is based on loop unrolling combined with loop shifting. In both cases, we compute the optimal unroll factor taking into consideration the execution times on GPP/FPGA, the memory transfers, and the area usage for one instance of the kernel. We will provide a detailed demonstration of the fact that combining unrolling and shifting gives better results than unrolling only. In the end, we show how the decision is made on what technique and what unroll factor to use.

Figure 2 presents a parallelized loop when applying the unrolling method for the simplified case $N \mod u = 0$. Each iteration consists of u sequential executions of the function `do_SW()` followed by the parallel execution of u kernel instances (there is an implicit synchronization point at the end of the parallel region).

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

Fig. 3. Parallelized loop after unrolling and shifting with a factor *u*.

The loop unrolling is extended in Figure 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 `do_SW` are executed in parallel with *u* identical kernel instances. The loop body has one iteration less than in the previous case (when applying only unrolling), as the first *u* calls of `do_SW` are executed before the loop body (namely, the loop prologue) and the last *u* kernel instances are executed after the loop body (the loop epilogue).

Next we will show how the unroll factor depends on the area, memory transfers, and execution times for the software part and for the kernel running in software/hardware.

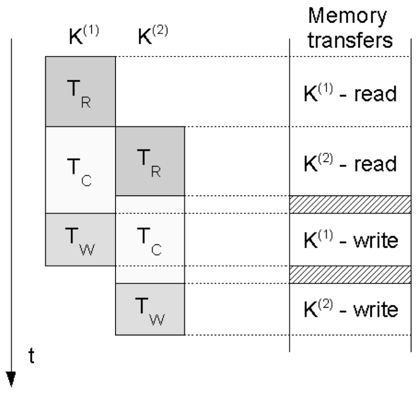
Area. Taking into account only the area constraints and assuming no constraint regarding the placement of the kernel, an upper bound for the unroll factor is set by:

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

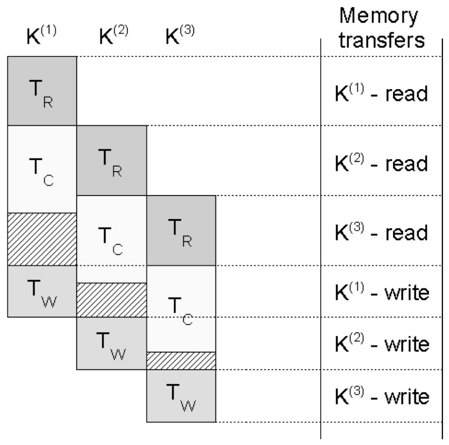
where:

—$Area_{(available)}$ 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);



(a) $T_{k(hw)}(u) = T_C + T_W + u \cdot T_r$



(b) $T_{k(hw)}(u) = u \cdot (T_r + T_W)$

Fig. 4. Parallelism on the reconfigurable hardware.

—$Area_{(K)}$ 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. Ideally, all data would be available immediately, and the degree of parallelism would be limited only by the area availability. However for many applications, the memory bandwidth is an important bottleneck in achieving the maximum theoretical parallelism. We consider that T_r, T_w, and T_c are, respectively, the times for memory read, write, and computation on hardware for kernel K, as indicated by the profiling information. If $T_{K(hw)}$ is the execution time for one instance of the kernel K on the reconfigurable hardware, then:

$$T_{K(hw)} = T_r + T_w + T_c \tag{2}$$

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 Figure 4,[2] a new instance of K can start only after a time T_r (we denote kernel instances by $K^{(1)}$, $K^{(2)}$, and so on).

Note that this is actually the worst case scenario in our analysis and our algorithm will always give results which are on the safe side. More precisely, using this assumption to compute the unroll factor bound, u_m, with respect to the assumed memory accesses is just a very quick worst-case estimation, which does not require further study of the target application. Moreover, determining u_m with this assumption guarantees that for any unroll factor that is less than u_m, there will be no computation stalls due to memory accesses/transfers. Depending on the hardware implementation, the real threshold value u_m for

[2]Figure 4 represents only one of the 8 possible cases ($T_W \leq T_r < T_c$ and $T_W + T_r > T_c$).

the unroll factor regarding memory transfers might be more permissive than our estimation.

Performance increases until the computation is fully overlapped by the memory transfers performed by the kernel instances running in parallel—see Figure 4—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 computation stalls occur due to waiting for the memory transfers to finish, as can be seen in Figure 4(b). Using the following notations:

$$T_{\mathrm{min(r,w)}} = \min\left(T_{\mathrm{r}}, T_{\mathrm{w}}\right); \qquad T_{\mathrm{max(r,w)}} = \max\left(T_{\mathrm{r}}, T_{\mathrm{w}}\right), \tag{3}$$

the time for running u instances of K on the reconfigurable hardware is:

$$T_{K(\mathrm{hw})}(u) = \begin{cases} T_{\mathrm{c}} + T_{\mathrm{min(r,w)}} + u \cdot T_{\mathrm{max(r,w)}}, & \text{if } u \leq \mathrm{u_m} \\ u \cdot (T_{\mathrm{r}} + T_{\mathrm{w}}), & \text{if } u > \mathrm{u_m} \end{cases} \tag{4}$$

The speedup at kernel level is $S_K(u) = \dfrac{u \cdot T_{K(\mathrm{hw})}(1)}{T_{K(\mathrm{hw})}(u)}$. For $u > \mathrm{u_m}$ the speedup is constant:

$$S_K(u > \mathrm{u_m}) = \frac{u \cdot (T_{\mathrm{r}} + T_{\mathrm{w}} + T_{\mathrm{c}})}{u \cdot (T_{\mathrm{r}} + T_{\mathrm{w}})} = \frac{T_{\mathrm{r}} + T_{\mathrm{w}} + T_{\mathrm{c}}}{T_{\mathrm{r}} + T_{\mathrm{w}}}, \tag{5}$$

thus it is not worth it to unroll more. Performance increases with the unroll factor while the following condition is satisfied:

$$T_{\mathrm{c}} + T_{\mathrm{min(r,w)}} + u \cdot T_{\mathrm{max(r,w)}} < u \cdot (T_{\mathrm{min(r,w)}} + T_{\mathrm{max(r,w)}}) \tag{6}$$

The memory bound can be derived:

$$u \leq \mathrm{u_m} = \left\lfloor \frac{T_{\mathrm{c}}}{T_{\mathrm{min(r,w)}}} \right\rfloor + 1. \tag{7}$$

When applied to the example in Figure 4, $\mathrm{u_m} = 2$. The case $u \leq \mathrm{u_m}$ corresponds to Figure 4(a) and the case $u > \mathrm{u_m}$ corresponds to Figure 4(b). In our example, $T_{\mathrm{w}} \leq T_{\mathrm{r}}$, thus $T_{\mathrm{max(r,w)}} = T_{\mathrm{r}}$ and $T_{\mathrm{min(r,w)}} = T_{\mathrm{w}}$. In Figure 4(a), the time for running two kernel instances ($\mathrm{K}^{(1)}$ and $\mathrm{K}^{(2)}$), in parallel, is given by the time for $\mathrm{K}^{(1)}$ ($T_{\mathrm{c}} + T_{\mathrm{r}} + T_{\mathrm{w}}$) plus the necessary delay for $\mathrm{K}^{(2)}$ to start (T_{r}). In Figure 4(b), $\mathrm{K}^{(1)}$ writing to memory is delayed because of $\mathrm{K}^{(3)}$ 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);

—T_{sw}—number of cycles for one instance of the software function (the function that is always executed by the GPP—in our example, the `do_SW` function);

—$T_{K(\mathrm{sw})}/T_{K(\mathrm{hw})}$—number of cycles for one instance of K() running in software/hardware;

—$T_{K(\mathrm{hw})}(u)$—number of cycles for u instances of K() running in hardware, as defined in (4) (only the case $u \leq \mathrm{u_m}$);

—$T_{\mathrm{loop(sw)}}$—number of cycles for the loop nest executed completely in software. Note that it does not depend on the unroll factor:

$$T_{\mathrm{loop(sw)}} = (T_{\mathrm{sw}} + T_{K(\mathrm{sw})}) \cdot N; \tag{8}$$

—$T_{\mathrm{loop(hw)}}(u)$—number of cycles for the loop nest with K() running on the FPGA, when applying loop unrolling with factor u (considering that $u < \mathrm{u_m}$);

—$T_{\mathrm{shift}}(u)$—number of cycles for the loop nest transformed with unrolling and shifting with factor u, with K() running on the FPGA;

—$S_{\mathrm{loop}}(u)/S_{\mathrm{shift}}(u)$—the speedup at loop level when applying unrolling/unrolling and shifting.

Speedup with loop unrolling. The speedup at loop nest level when only loop unrolling is used is defined as:

$$S_{\mathrm{loop}}(u) = \frac{T_{\mathrm{loop(sw)}}}{T_{\mathrm{loop(hw)}}(u)}. \tag{9}$$

For the simplified case in Figure 2, where $N \bmod u = 0$, the total execution time for the loop with the kernel running in hardware is:

$$T_{\mathrm{loop(hw)}}(u) = (T_{\mathrm{sw}} \cdot u + T_{K(\mathrm{hw})}(u)) \cdot (N/u).$$

For the general case where u is not a divisor of N, the remainder instances of the software function and hardware kernel will be executed in the loop epilogue. We denote by R the remainder of the division of N by u:

$$R = N - u * \lfloor N/u \rfloor, 0 \leq R < u. \tag{10}$$

We define $T_{K(\mathrm{hw})}(R)$ as:

$$T_{K(\mathrm{hw})}(R) = \begin{cases} 0, & R = 0 \\ T_c + T_{\mathrm{min(r,w)}} + R \cdot T_{\mathrm{max(r,w)}}, & R > 0 \end{cases}. \tag{11}$$

Then, $T_{\mathrm{loop(hw)}}(u)$ is:

$$\begin{aligned} T_{\mathrm{loop(hw)}}(u) &= \lfloor N/u \rfloor \cdot (u \cdot T_{\mathrm{sw}} + T_{K(\mathrm{hw})}(u)) + (R \cdot T_{\mathrm{sw}} + T_{K(\mathrm{hw})}(R)) \\ &= N \cdot T_{\mathrm{sw}} + \lfloor N/u \rfloor \cdot T_{K(\mathrm{hw})}(u) + T_{K(\mathrm{hw})}(R), \end{aligned} \tag{12}$$

which, by expanding $T_{K(\mathrm{hw})}(u)$ from (4) with $\mathrm{u} \leq \mathrm{u_m}$, is equivalent to:

$$T_{\mathrm{loop(hw)}}(u) = (T_{\mathrm{sw}} + T_{\mathrm{max(r,w)}}) \cdot N + (T_{\mathrm{c}} + T_{\mathrm{min(r,w)}}) \cdot \lceil N/u \rceil. \tag{13}$$

Since $T_{\mathrm{loop(sw)}}(u)$ is constant and $T_{\mathrm{loop(hw)}}(u)$ is a monotonic decreasing function, then $S_{\mathrm{loop}}(u)$ 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 may not want to increase the unrolling if the gain in speedup would be by a factor of 0.1%, but the area usage would increase by 15%.) The simplest relation to be satisfied between the speedup and necessary area is:

$$\Delta S(u+1, u) > \Delta A(u+1, u) \cdot F, \tag{14}$$

where $\Delta A(u+1, u)$ is the relative area increase, which is constant, since all kernel instances are identical:

$$\Delta A(u+1, u) = A(u+1) - A(u) = Area_{(K)} \in (0,\ 1), \tag{15}$$

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

$$\Delta S(u+1, u) = \frac{S_{\text{loop}}(u+1) - S_{\text{loop}}(u)}{S_{\text{loop}}(u)}. \tag{16}$$

Note that only in the ideal case $\dfrac{S_{\text{loop}}(u+1)}{S_{\text{loop}}(u)} = \dfrac{u+1}{u}$, which means that:

$$S_{\text{loop}}(u+1) < 2 \cdot S_{\text{loop}}(u),\ \forall u \in \mathbb{N},\ u > 1, \tag{17}$$

and the relative speedup satisfies the relation:

$$\Delta S(u+1, u) \in [0,\ 1),\ \forall u \in \mathbb{N},\ u > 1. \tag{18}$$

Thus, F is a threshold value that sets the speedup bound for the unroll factor (u_s). How to choose a good value for F is not within the scope of this research. However, it should be mentioned that a greater value of F would lead to a lower bound, which translates to the price we are willing to pay in terms of area compared to the speedup gain is small. Also, the value of F should be limited by $\dfrac{\Delta S(2, 1)}{Area_{(K)}}$, which is the value that would allow the unroll factor of 2—a larger value would lead to the unroll factor 1 (no unroll):

$$F \in \left[0,\ \frac{\Delta S(2, 1)}{Area_{(K)}}\right]. \tag{19}$$

By using (8) and (13) in (9) as well as the following notations:

$$x = \frac{T_\text{c} + T_{\text{min(r,w)}}}{(T_{\text{max(r,w)}} + T_{\text{sw}}) \cdot N} \quad \text{and} \quad y = \frac{T_{\text{sw}} + T_{K(\text{sw})}}{T_{\text{max(r,w)}} + T_{\text{sw}}}, \tag{20}$$

the total speedup is computed by:

$$S_{\text{loop}}(u) = \frac{y}{1 + x \cdot \lceil N/u \rceil}. \tag{21}$$

The speedup bound is defined as:

$$\text{u}_\text{s} = \min(u) \quad \text{such that} \quad \Delta S(u+1, u) < F \cdot Area_{(K)}. \tag{22}$$

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 $u < \text{u}_\text{m}$, we add another condition for u_s:

$$\Delta S(\text{u}_\text{s} + 2, \text{u}_\text{s} + 1) < F \cdot Area_{(K)}. \tag{23}$$

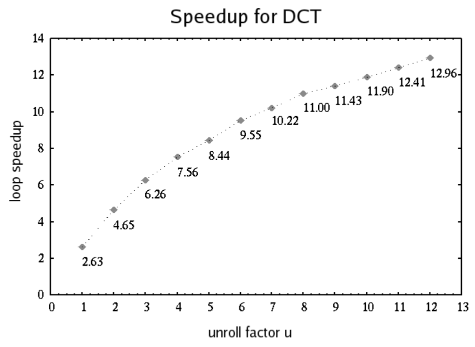


Fig. 5. DCT loop speedup.

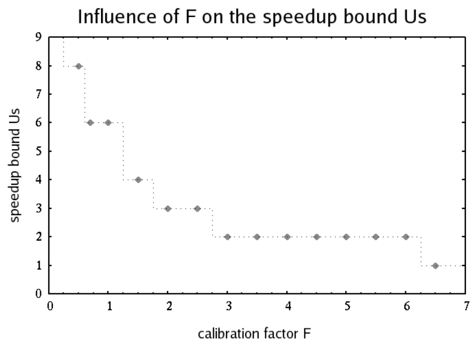


Fig. 6. Influence of F on u_s.

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 (u_a and u_m), as long as there is no performance degradation. In this case, we set $F = 0$ and $u_s = u_m$.

On the basis of (21), binary search can be used to compute in $O(\log N)$ time at compile-time, the value of u_s that satisfies the conditions $\Delta S(u_s + 1, u_s) < F \cdot Area_{(K)}$ and $\Delta S(u_s + 2, u_s + 1) < F \cdot Area_{(K)}$.

Figure 5 illustrates the speedup achieved for different unroll factors for the DCT kernel, as presented in Dragomir et al. [2008a]. The area for one instance of DCT is 0.12 (12% of the area on Virtex II Pro). With respect to these data, Figure 6 shows how different values of F influence the unroll factor. Note that in this case, $\frac{\Delta S(2, 1)}{Area_{(K)})} = 6.23$, which means that for $F > 6.23$ there will be no unroll ($u_s = 1$).

Speedup with unrolling and shifting. The speedup at loop nest level when unrolling and shifting are used is:

$$S_{\text{shift}}(u) = \frac{T_{\text{loop(sw)}}}{T_{\text{shift}}(u)}, \tag{24}$$

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

$$T_{\text{shift}}(u) = T_{\text{prolog}}(u) + T_{\text{body}}(u) + T_{\text{epilog}}(u). \tag{25}$$

Looking at Figure 3, the corresponding notations are the following:
(1) $T_{\text{prolog}}(u)$ is the time for the loop prologue:

$$T_{\text{prolog}}(u) = u \cdot T_{\text{sw}}; \tag{26}$$

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

$$T_{\text{body}}(u) = (\lfloor N/u \rfloor - 1) \cdot \max\left(u \cdot T_{\text{sw}},\ T_{K(\text{hw})}(u)\right); \tag{27}$$

(3) $T_{\text{epilog}}(u)$ is the time for the loop epilogue.
For the simplified case in Figure 3, the epilogue consists of the hardware parallel execution of u kernel instances: $T_{\text{epilog}}(u) = T_{K(\text{hw})}(u)$.
For the general case ($N \mod u \neq 0$), the time for the epilogue is:

$$T_{\text{epilog}}(u) = \max\left(R \cdot T_{\text{sw}},\ T_{K(\text{hw})}(u)\right) + T_{K(\text{hw})}(R), \tag{28}$$

where $T_{K(\text{hw})}(R)$ was defined in (11).

In order to compute $T_{\text{body}}(u)$ from (27) and $T_{\text{epilog}}(u)$ from (28)—where the max function is used—there are different cases, depending on the relations between T_{sw}, T_{c}, $T_{\min(\text{r,w})}$, and $T_{\max(\text{r,w})}$. 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. If $T_{\text{sw}} \leq T_{\max(\text{r,w})}$, then the execution time for the software part inside a loop iteration increases more slowly than the hardware part and the threshold value is $U_1 = \infty$. Otherwise, the execution time of the software part increases more quickly than the hardware part and we have to compute the threshold value. The execution time for one iteration is:

$$\max\left(u \cdot T_{\text{sw}},\ T_{K(\text{hw})}(u)\right) = \begin{cases} T_{K(\text{hw})}(u), & u < U_1 \\ u \cdot T_{\text{sw}}, & u \geq U_1 \end{cases}. \tag{29}$$

If $u \leq U_1$ then:

$$u \cdot T_{\text{sw}} \leq T_{\text{c}} + T_{\min(\text{r,w})} + u \cdot T_{\max(\text{r,w})}. \tag{30}$$

This determines the threshold value U_1 for the case $T_{\text{sw}} > T_{\max(\text{r,w})}$ as:

$$U_1 = \left\lceil \frac{T_{\text{c}} + T_{\min(\text{r,w})}}{T_{\text{sw}} - T_{\max(\text{r,w})}} \right\rceil. \tag{31}$$

The execution time for the loop transformed with unrolling and shifting is:

$$T_{\text{shift}}(u) = \begin{cases} u \cdot T_{\text{sw}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) + T_{K(\text{hw})}(R), & (u < U_1) \\ \lfloor N/u \rfloor \cdot u \cdot T_{\text{sw}} + \max\left(R \cdot T_{\text{sw}}, T_{K(\text{hw})}(u)\right) + T_{K(\text{hw})}(R), & (u \geq U_1) \end{cases} \tag{32}$$

Intuitively, we expect that the unroll factor that gives the smallest execution time and thus the largest speedup is the one where the software and hardware execute concurrently in approximatively the same amount of time. This happens in the close vicinity of U_1 (we define the close vicinity as the set $\{U_1 - 1, U_1, U_1 + 1\}$), depending upon whether any of these values is a divisor of N or not. More specifically, if U_1 is a divisor of N, then it is the value that maximizes the speedup function. Otherwise, this value might be found for $u > U_1$ when u is a divisor of N, but then $T_{\text{shift}}(u)$ is significantly smaller (more than 10%) then $T_{\text{shift}}(U_1)$ only if $\lfloor N/u \rfloor \geq 10$.

Unrolling and shifting versus unrolling. We compare $T_{\text{shift}}(u)$ from (32) with $T_{\text{loop(hw)}}(u)$ from (12).

$$T_{\text{loop(hw)}}(u) - T_{\text{shift}}(u) = \begin{cases} (N - u) \cdot T_{\text{sw}}, & \text{if } u < U_1 \\ R \cdot T_{\text{sw}} + \lfloor N/u \rfloor \cdot T_{K(\text{hw})}(u) - \\ \quad \max(R \cdot T_{\text{sw}},\ T_{K(\text{hw})}(u)), & \text{if } u \geq U_1 \end{cases} . \tag{33}$$

This means that the execution time for the two methods is the same when using the maximum unroll factor ($u = N$) or when the loop has no software part. Otherwise, the execution time for the loop transformed with unrolling and shifting is smaller than the one for unrolling only; thus the performance is better.

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.

(a) If $T_{\text{sw}} = 0$ (the loop has no software part), or if the loop dependencies between `do_SW` and K from different iterations do not allow parallelization (running `do_SW` and K in parallel) after shifting, then the loop will be transformed with loop unrolling and the chosen unroll factor will be:

$$U = \min(u_{\text{a}}, u_{\text{m}}, u_{\text{s}}). \tag{34}$$

(b) If $T_{\text{sw}} \neq 0$ (the loop has a software part) and no data dependencies between the software and hardware parts in different iterations (`do_SW(i)` and `K(j)` with $i \neq j$), then loop unrolling will be combined with shifting in order to obtain better performance. Let u_{min} be $u_{\text{min}} = \min(u_{\text{a}}, u_{\text{m}})$.

If $T_{\text{sw}} > T_{\text{max(r,w)}}$ and the unroll factor threshold U_1 satisfies the memory and area constraints ($U_1 < u_{\text{min}}$), the algorithm looks for the unroll factor that gives the best execution time in the close vicinity of U_1. If this unroll factor 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 a speedup improvement over the speedup achieved for U_1.

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

$$U = \min_{u} \left\{ u \in \mathbb{Z} | u \le u_{\min} \text{ and } S_{\text{shift}}(u) = \max_{i \le u_{\min}} \left(S_{\text{shift}}(i) \right) \right\}. \tag{35}$$

Note that unrolling is not beneficial if $U = 1$—only loop shifting will be used.

5. EXPERIMENTAL RESULTS

The purpose of this section is to illustrate the presented method which automatically computes the unroll factor and selects the most appropriate loop transformation (unrolling, shifting, or unrolling plus shifting) taking into account the area and memory constraints and the profiling information. We ran several kernels from well-known applications and analyzed the relative speedup obtained by running multiple instances of a kernel in parallel, compared to running a single one. The achieved performance depends on the kernel implementation, but is also subject to Amdahl's Law.

5.1 Selected Kernels

We selected four different kernels to illustrate the presented methods, some with multiple implementations. The dependencies between different iterations of the loops containing the kernels have been eliminated, if there were any. The VHDL code for the analyzed kernels was automatically generated with the DWARV [Yankova et al. 2007] tool and synthesized with the Xilinx XST tool of ISE 8.1.

The kernels are as follows.

(1) *DCT (Discrete Cosine Transformation)*—extracted from the MPEG2 encoder. Results for DCT have also been presented in Dragomir et al. [2008a]. In this case, `do_SW` performs a preprocessing of the blocks, by adjusting the luminance/chrominance.

(2) *Convolution*—from the Sobel algorithm. We have chosen not to implement the whole Sobel algorithm in hardware because most of the time is spent inside a loop that uses the convolution; and accelerating the convolution also leads to a large speedup for the whole Sobel, but with much smaller area consumption. As the kernel consists only of the convolution, the part of the algorithm that adjusts the values of the pixels to the interval [0, 255] is contained in the `do_SW` function.

(3) *SAD (Sum of Absolute Differences)*—also extracted from the MPEG2 encoder. For this kernel, we have chosen two VHDL implementations, one that is faster but occupies more area and one that is slower but takes less area; we will call them SAD-time and SAD-area, although the VHDL code was automatically generated and the performance does not compare to a handwritten implementation. The loop nest containing the SAD has been transformed into a perfect nest, and the execution times for one instance of SAD on GPP/FPGA used in the algorithm are taken as the weighted average for the execution times of all instances within the loop

Table II. Profiling Information for the Analyzed Kernels

Kernel	Area [%]	T_r/T_w (cycles)	$T_K(\text{sw})$ (cycles)	$T_K(\text{hw})$ (cycles)	N iter.	$T_{\text{loop(sw)}}$ (cycles)	T_{sw} (cycles)
DCT	12.39	192/64	106626	37278	6*16	10751868	5292
Convolution	3.70	24/1	2094	204	126*126	35963184	168
SAD-area	6.81	330/1	4013	2908	1*13	619392	84
SAD-time	13.17	330/1	4013	1305	11*13	619392	84
Quantizer-1	2.98	192/64	12510	2970	64*16	20925786	8112
Quantizer-2	4.35	192/64	12510	1644	64*16	20925786	8112
Quantizer-4	7.08	192/64	12510	1068	64*16	20925786	8112
Quantizer-8	12.13	192/64	12510	708	64*16	20925786	8112

nest. The length of the execution of a SAD function is determined by the values of some parameters that are updated after each execution. Therefore, `do_SW` is the post-processing code, which updates the values of these parameters.

(4) *Quantizer*—an integer implementation, extracted from the JPEG application. For this kernel, we have four implementations (denoted by Q-1, Q-2, Q-4, and Q-8), the one with the smallest area consumption is the slowest (Q-1), and the fastest one also requires the most area (Q-8). The `do_SW` is a post-processing function, it performs a zig-zag transformation of the matrix.

Table II summarizes the profiling information for the analyzed kernels. It includes the area occupied by one kernel instance, memory transfer times, various execution times, and the number of iterations in the loop nest. The execution times were measured using the PowerPC timer registers (for the kernels running on FPGA, the times include the parameter transfer using exchange registers). They are the following:

(1) the execution time for a single instance of each kernel running in software/hardware—$T_K(\text{sw})/T_K(\text{hw})$;
(2) the execution time for the whole loop—$T_{\text{loop}}(\text{sw})$;
(3) the execution time for the software part (in one iteration)—T_{sw}.

The times for the memory transfers (T_r/T_w) are computed considering three cycles per memory read (in the automatically generated VHDL, the memory transfers are not pipelined) and one cycle per memory write.

The experiment was performed with one instance of the kernel running on the FPGA. The results for the execution time of the loop for higher unroll factors were computed using (13) and (32), and the speedups for the two methods were computed using (9) and (24).

5.2 Discussion of the Results

We computed the theoretical maximum achievable speedup using Amdahl's Law for parallelization, considering that we have maximum parallelism for the kernels executed in hardware (*e.g.*, full unroll). The serial part of the program (that part that cannot be parallelized) consists of the N instances (N is the number of iterations in the loop) of the software function `do_SW` plus one instance of the kernel running in hardware. The parallelizable part consists of the N instances of the kernel running in software minus one instance of the kernel running in hardware, because the execution time of N kernels in hardware is at least the time for running a single kernel—that is the ideal case. We denote by P_{K} the percentage of the loop time that can be parallelized:

$$P_{\mathrm{K}} = \frac{N \times T_{K(\mathrm{sw})} - T_{K(\mathrm{hw})}}{T_{\mathrm{loop(sw)}}}. \tag{36}$$

From now on, we will refer to the computed theoretical maximum speedup using Amdahl's Law as *the speedup by Amdahl's Law*. Then, the speedup by Amdahl's Law is:

$$S_{Amdahl} = \frac{1}{1 - P_{\mathrm{K}} + \dfrac{P_{\mathrm{K}}}{N}}. \tag{37}$$

Note that Amdahl's Law neglects potential bottlenecks, such as memory bandwidth.

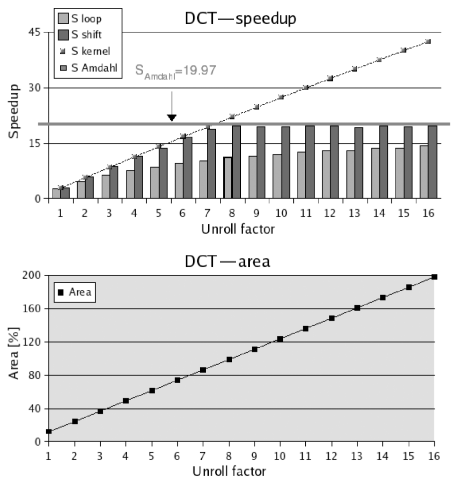
The results for the analyzed kernels are illustrated in Figure 7. Together with the achieved speedups for different unroll factors, the speedup by Amdahl's Law is also shown for each of the kernels. Table III summarizes these results, showing the optimal unroll factor, the area requirements, the achieved speedup, and the speedup by Amdahl's Law for each kernel.

(1) *DCT (MPEG2)*. It can be seen that $S_{\mathrm{shift}}(u)$ grows significantly faster than $S_{\mathrm{loop}}(u)$ until unroll factor $u = U_1 = 8$. At this point, $S_{\mathrm{loop}}(u) = 11.06$ and $S_{\mathrm{shift}}(u) = 19.65$, which is the maximum value. For $u > 8$, $S_{\mathrm{shift}}(u)$ will have values between 18.5 and 19.65, while $S_{\mathrm{loop}}(u)$ is monotonically increasing, with the maximum value 19.07 for $u = N$.

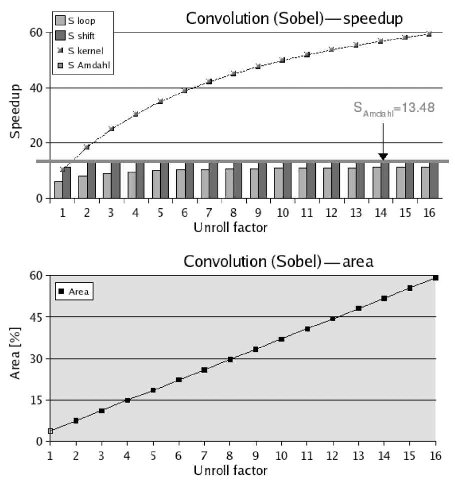
    For comparison, the speedup at kernel level—which would be achieved if the loop would not have a software part—is drawn with a dashed line on the same figure, while the speedup by Amdahl's Law is represented as a horizontal continuous line with the value 19.97. The area consumption is illustrated in a chart following the speedup chart.

    In conclusion, if the area is not a constraint, the best solution is unrolling and shifting with unroll factor 8. For Virtex II Pro, considering only the area requirements for DCT and for Molen, the optimal unroll factor is 7, with area consumption 87% and speedup of 18.7.

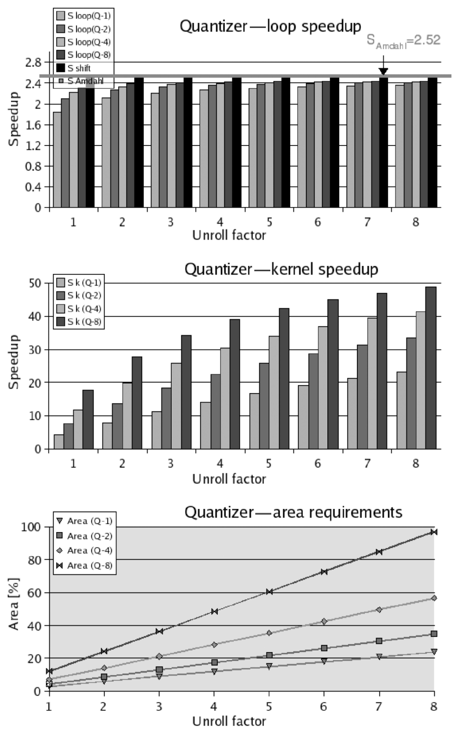
(2) *Convolution (Sobel)*. The execution time for the software part of the loop is 82% of the kernel's execution time in hardware, which leads to $U_1 = 2$. Indeed, as suggested by Figure 7(b), the maximum for $S_{\mathrm{shift}}(u)$ is achieved for $u = 2$. $S_{\mathrm{loop}}(u)$ is an increasing function, reaching its maximum value of 11.8 for $u = N$. Note that the speedup obtained by combining unrolling with shifting is approximatively equal to the speedup by Amdahl's Law: 13.48.



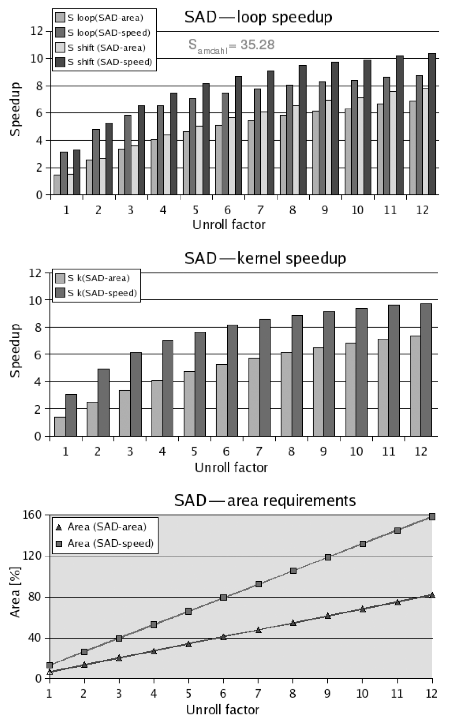
(a) DCT—speedup and area



(b) Convolution—speedup and area



(c) Quantizer—speedup and area



(d) SAD—speedup and area

Fig. 7. Speedup and area consumption for: (a) DCT; (b) Convolution; (c) Quantizer; (d) SAD.

The speedup at kernel level is drawn with a dashed line on the same figure. Its shape is due to the fact that the time for memory transfers represents a significant amount of the kernel's execution time in

Table III. Results Summary for the Analyzed Kernels

Kernel	U opt.	Area usage [%]	Speedup	Max speedup (Amdahl)	Percent
DCT	7	87.00	18.70	19.97	93.6 %
Convolution	2	7.40	13.48	13.48	99.9 %
SAD	6	79.00	8.71	35.28	24.7 %
Quantizer	1	12.12	2.52	2.52	99.9 %

hardware—almost 12%. The area requirements for different unroll factors are represented in the following.

In conclusion, unrolling combined with shifting gives an optimal unroll factor of 2, which leads to a speedup of 13.48 for the whole loop, and area consumption of 7.4% of the Virtex II Pro area.

(3) *SAD (MPEG2)*. The software part of the loop executes faster than the memory transfers performed by the hardware kernel, thus we consider $U_1 = \infty$. The consequence of this fact is that $S_{\text{shift}}(u)$ will look like an increasing function in the interval $[0, N]$, with a maximum at $u = N$.

For SAD we have two different implementations, one is more time-optimized and area-consuming (SAD-time), and the other is more area-optimized and time-consuming (SAD-area). We note that for SAD-time, the optimal unroll factor is 6 with speedup of 8.71 and area consumption of 79% (92.2% occupied area for unroll factor 7 would be too much—it would make the design slow). For SAD-area, the optimal unroll factor is 13, with area consumption of 88.5% and speedup of 8.08.

The decision is to use SAD-time, which gives better performance and requires less area. Nevertheless, we must mention that the decision on which implementation to use differs from case to case as it cannot be taken without comparing the best solutions for the different implementations.

The maximum Amdahl based speedup is 35.28. In this case, the performance achieved with our method is only at 24.7% of the theoretical maximum. Part of the explanation is that Amdahl's Law does not take into account memory bottlenecks. Memory accesses represent an important part of the total execution time of the SAD kernel (11.34% for SAD-area and 25.28% for SAD-time), as they are not optimized in the hardware implementations generated by the DWARV tool. For this reason, the speedup increases slowly when unrolling. Also, the time for the software function (T_{sw}) is smaller than the time for the memory transfers $T_{\max(\text{r,w})}$, meaning that the execution on the GPP is completely hidden by the execution on the FPGA—when loop unrolling and shifting are performed.

In conclusion, this is an example that would benefit more from an optimized hardware implementation of the kernel that would be able to reduce the memory access time.

(4) *Quantizer (JPEG)*. Out of the four Quantizer implementations we tested (Q-1, Q-2, Q-4 and Q-8), Q-1 is the slowest but requires the least area, and Q-8 is the fastest and requires the most area. However, T_{sw} (the execution time for the software function) is significantly larger than $T_{K(\text{hw})}$ (the execution time for the kernel running in hardware). This leads to a very

small performance increase, as can be seen in Figure 7(d). When also applying loop shifting, the speedup increases slightly compared to the case where only loop unrolling is used, but S_{shift} is ≈ 2.52 for all unroll factors, and for all Quantizer implementations.

Kernel speedup and area consumption are represented separately. We notice that a kernel speedup of 40 is possible, with area consumption of 50% (Q-4 for $u = 7$), but the loop speedup for the same unroll factor is only 2.4.

As a conclusion, this loop example shows that if the software part of the loop needs more time to execute than the kernel's time in hardware, there will be very little or no gain in unrolling, no matter how optimized the kernel implementation is. This is nothing but a practical proof for Amdahl's Law. In this context, the best solution is to apply shifting without unrolling. For our example, this leads to a speedup of 2.52, slightly better than the speedup achieved with Q-8 without unrolling, which is 2.32, and very close to the theoretical maximum given by Amdahl's Law, which is also ≈ 2.52. The area consumption is 12.13% of the Virtex II Pro area.

5.3 Study Case: DCT

Our automatically generated VHDL implementation of DCT has a speedup factor of 2.86 over the software. The maximum loop speedup is 19.65, achieved when parallelizing the loop using unrolling and shifting with a factor of 8. This requires 99% of the area of Virtex II Pro, but is perfectly feasible for the Virtex-4 we are targeting for the near future.

Vassiliadis et al. [2004] show a speedup of 302× for the hardware implementation of DCT compared to the software version. However, the loop nest we extracted from MPEG2 contains the DCT kernel plus some additional software code (`do_SW`). Thus, the speedup reported by our algorithm for the whole loop including `do_SW` and using this aggressively optimized DCT implementation is 19.83x, while the maximum speedup that can be achieved for the whole loop is, according to Amdahl's Law, 21.42. By applying loop shifting and running `do_SW` and DCT in parallel, the speedup for the whole loop grows to 21.13. The impact of our transformations is much smaller because the I/O becomes the bottleneck. However, the achieved speedup is 98.64% of the speedup by Amdahl's Law.

5.4 Conclusion of the Experimental Results

In summary, the experiments show the following possible situations.

(1) $T_K(\text{hw}) >> T_{\text{sw}}$ (DCT, SAD). In this case, loop unrolling plus shifting will be the most beneficial. It will also matter how optimized the kernel implementation is, but only up to the point when $T_K(\text{hw})$ becomes comparable to, or less than, T_{sw}. When that happens, the conclusions for the second or third category apply (see the following).

(2) $T_K(\text{hw}) \approx T_{\text{sw}}$ (Convolution). Most probably, a combination of unrolling and shifting with a factor of 2 is the best solution.

(3) $T_K(\text{hw}) << T_{\text{sw}}$ (Quantizer). In this case, unrolling is not beneficial, which is also seen from Amdahl's Law. By applying loop shifting without unrolling, there will be a slight performance improvement, while the area requirements will not change.

By using automatically generated VHDL code for the analyzed kernels, we obtained the following results:

—DCT speedup of 18.7 for loop unrolling plus shifting with factor 7 (the speedup by Amdahl's Law is 19.97);
—Convolution speedup of 13.48 for unrolling plus shifting with factor 2 (the speedup by Amdahl's Law is 13.48);
—SAD speedup of 8.71 for unrolling plus shifting with factor 6, using SAD time-optimized (the speedup by Amdahl's Law is 35.28, but it does not consider memory bottlenecks);
—Quantizer speedup of 2.52 using only loop shifting and any of the Quantizer implementations (Q-1 requires the least area). Loop unrolling does not bring any benefit, as the result is already approximatively equal to the speedup by Amdahl's Law, which is 2.52.

These results prove that there are many cases when applying our methods leads to a speedup comparable to the theoretical maximum by Amdahl's Law. However, when the kernel is I/O intensive and also $T_{\text{sw}} < T_{\max(\text{r,w})}$, the performance improvement is not as high as expected and a more memory-optimized hardware implementation would lead to significant further improvement.

6. CONCLUSION AND FUTURE WORK

In this article, we presented a method based on loop unrolling and loop shifting for computing the optimal number of instances of a kernel K that will run in parallel on the reconfigurable hardware, with the possibility of concurrently executing code on the GPP. The input data for our algorithm consists of profiling information about memory transfers, execution times in software and hardware, and information about area usage for one kernel instance and area availability. The algorithm also selects the best transformation to be applied to the loop nest containing the kernel K.

One of the main benefits of this algorithm is that it can be used to improve performance for any VHDL implementation of the kernel, if there are enough resources available (for instance, when moved to a different platform). Moreover, its implementation in the compiler decreases the time for design-space exploration and makes efficient use of the hardware resources.

An important contribution of this article consists of the experimental results. An extended analysis of the possible cases that may be encountered in real applications is included. We analyzed four well-known kernels (DCT, Sobel's Convolution, SAD, and Quantizer) and compared the achieved results with the theoretical maximum speedup computed with Amdahl's Law assuming maximum parallelism (full unroll) for the hardware. We showed the following.

(1) When a loop contains a kernel call and also a part that will always be executed by the GPP, it is always beneficial to perform loop shifting, if the data dependencies constraint is met.
(2) When more than one hardware implementation is available for a kernel, the one that gives the better overall performance is chosen.
(3) Depending on the ratio between the execution time of the software part of the loop and that of the kernel executing in hardware, we see different aspects of Amdahl's Law. If the software is faster, different results will be obtained for different kernel implementations, depending on how much they are optimized. However, the speedup cannot be larger than the theoretical maximum given by Amdahl's Law. If the execution time of the software part is much larger than the kernel's execution time in hardware, then unrolling would not be beneficial, and in this case only loop shifting should be performed.
(4) For I/O intensive kernels, the performance that can be achieved by applying our methods with automatically generated VHDL is quite far from the theoretical maximum. This happens because: (1) Amdahl's Law does not consider memory bottlenecks; (2) in the current stage, the DWARV tool does not optimize the memory accesses.

Also, we have studied the hypothetical case of using an aggressively optimized implementation of DCT (speedup of 302× for the DCT hardware implementation compared to the software, while the automatically generated hardware gave a speedup of 2.86× compared to the software). Making use of the same profiling information for the execution times in software and for the memory transfers, we found that the speedups achieved in the two cases for the loop containing the DCT kernel and the software function are comparable.

In our future work, we are considering relaxing some of the assumptions regarding the loop and the memory accesses. We are also considering extending the model by supporting loops with an arbitrary number of hardware kernels and with pieces of software code that also occur in between the kernels.

REFERENCES

BANERJEE, S., BOZORGZADEH, E., AND DUTT, N. 2006. PARLGRAN: Parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures. In *Proceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC'06)*. 491–496.

CARDOSO, J. M. P. AND DINIZ, P. C. 2004. Modeling loop unrolling: Approaches and open issues. In *Proceedings of the 4th International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS'04)*. 224–233.

DARTE, A. AND HUARD, G. 1999. Loop shifting for loop compaction. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC'99)*. 415–431.

DRAGOMIR, O. S., MOSCU-PANAINTE, E., BERTELS, K., AND WONG, S. 2008a. Optimal unroll factor for reconfigurable architectures. In *Proceedings of the 4th International Workshop on Applied Reconfigurable Computing (ARC'08)*. 4–14.

DRAGOMIR, O. S., STEFANOV, T., AND BERTELS, K. 2008b. Loop unrolling and shifting for reconfigurable architectures. In *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL'08)*.

GUO, Z., BUYUKKURT, B., NAJJAR, W., AND VISSERS, K. 2005. Optimized generation of data-path from C codes for FPGAs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*. 112–117.

GUPTA, S., DUTT, N., GUPTA, R., AND NICOLAU, A. 2004. Loop shifting and compaction for the high-level synthesis of designs with complex control flow. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)*. 114–119.

KUZMANOV, G., GAYDADJIEV, G., AND VASSILIADIS, S. 2004. The Virtex II Pro MOLEN processor. In *Proceedings of the 4th International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS'04)*. 192–202.

LIAO, J., WONG, W.-F., AND MITRA, T. 2003. A model for hardware realization of kernel loops. In *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL'03)*. 334–344.

VASSILIADIS, S., GAYDADJIEV, G. N., BERTELS, K., AND PANAINTE, E. M. 2003. The Molen programming paradigm. In *Proceedings of the 3rd International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03)*. 1–7.

VASSILIADIS, S., WONG, S., GAYDADJIEV, G., BERTELS, K., KUZMANOV, G., AND PANAINTE, E. M. 2004. The Molen polymorphic processor. *IEEE Trans. Comput. 53*, 11, 1363–1375.

WEINHARDT, M. AND LUK, W. 2001. Pipeline vectorization. *IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst.* 234–248.

XILINX INC. 2007. Virtex II Pro and Virtex II Pro X platform FPGAs: Complete data sheet. http://www.xilinx.com/bvdocs/publications/ds083.pdf.

YANKOVA, Y. D., KUZMANOV, G., BERTELS, K., GAYDADJIEV, G., LU, Y., AND VASSILIADIS, S. 2007. DWARV: DelftWorkbench automated reconfigurable VHDL generator. In *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL'07)*. 697–701.

Received May 2008; revised September 2008; accepted January 2009