Generic Loop Parallelization for Reconfigurable Architectures

Ozana Silvia Dragomir and Koen Bertels

TU Delft, Mekelweg 4, 2628CD, Delft, The Netherlands
{O.S.Dragomir, K.L.M.Bertels}@tudelft.nl

***Abstract*—Reconfigurable Computing (RC) is one of the most intensively studied research areas nowadays due to its potential to dramatically increase application performance. RC combines a general purpose processor (GPP) and a Field Programmable Gate Array (FPGA), having the advantages of both hardware performance and software flexibility. Modern real-life applications (such as audio, video, image processing, etc) spend most of the execution time in loops, which represent or include the application kernels. These loops are an important source of performance improvement. In our work, we target loops that contain in their bodies code for the GPP (software functions) and also for the FPGA (hardware functions). We assume there are data dependencies between consecutive tasks in the loop body, but not between different loop iterations. Assuming the Molen machine organization 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 concurrently executing code on the GPP. In this paper, we focus on loop transformations that are suitable for loops containing an arbitrary number of software and hardware functions. The *extended shifting* consists of relocating the functions placed in the beginning and in the end of one loop iteration, in order to eliminate the data dependencies and allow certain software and hardware functions to be executed in parallel. The *loop distribution* consists of splitting the loop into small loops (e.g., with only one kernel) allowing in some cases a larger degree of parallelism when applying the loop unrolling and shifting techniques. We estimate the performance achieved by applying the extended shifting technique in conjunction with loop unrolling and compare it to the performance achieved when applying the loop unrolling and shifting techniques to smaller loops obtained by distributing the original loop. For the experimental results we used randomly generated tests, for loops containing a variable number of kernels (between 2 and 8 kernels).**

Keywords: loop optimizations, reconfigurable computing, Molen programming paradigm, FPGA

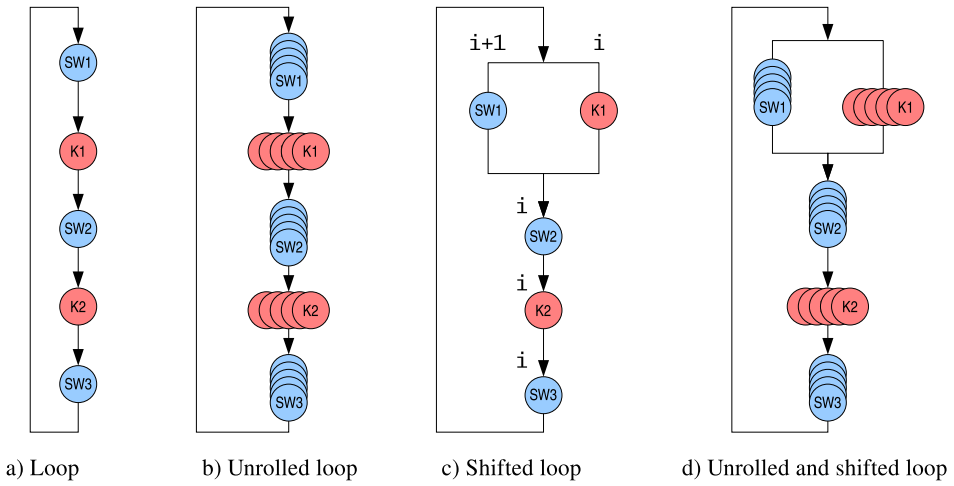
I. Introduction

Modern real-life applications (such as audio, video, image processing, etc) spend most of the execution time in loops, which represent or include the application kernels. These loops are an important source of performance improvement. Various loop transformations (such as loop unrolling, software pipelining, loop shifting, loop distribution, loop merging, or loop tiling) can be used successfully to maximize the parallelism inside the loop and improve the application performance.

The applications we target in our work have loops that contain in their bodies code for the GPP (software functions) and also for the FPGA (hardware functions). Our goal is to improve the performance for such loops, by applying standard loop transformations such as the ones mentioned above. We take into account the fact 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 level), they show a great potential for improving the performance. We describe briefly the loop transformations we found to be beneficial in the targeted applications.

Loop unrolling is traditionally used to eliminate the loop overhead, improving cache hit rate and reducing branching by replicating the loop body. We use unrolling to expose the loop parallelism, allowing us to execute concurrently multiple kernels on the reconfigurable hardware.

Loop shifting is a transformation that moves operations from one iteration of the loop body to the previous iteration. The operations are shifted from the beginning of the loop body to the end of the loop body and a copy of these operations is also placed in the loop prologue. In our research, loop shifting means moving a function from the beginning of the loop body to the end and we use it to eliminate the data dependencies between software



a) Loop b) Unrolled loop c) Shifted loop d) Unrolled and shifted loop

Fig. 1. Loop containing several kernels

and hardware functions, allowing concurrent execution on the GPP and FPGA.

Loop distribution is a technique that breaks a loop into multiple loops over the same index range but each taking only a part of the loop's body. We use loop distribution to break down large loop bodies into smaller ones that can be parallelized more with loop unrolling and loop shifting in order to improve the performance.

II. 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 our previous work [10], [11] we presented results of applying loop unrolling and loop shifting to small loops containing only one software and one hardware function.

A. Target architecture.

Our target architecture is Molen [1], which allows running multiple kernels/applications at the same time on the reconfigurable hardware. 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 memory bandwidth.

Our assumptions regarding the application and the framework are the following:

1) There are no data dependencies between different iterations.
2) The loop bounds are known at compile time.
3) The loops are perfectly nested.
4) Inside the kernel, all memory reads are performed in the beginning and memory writes in the end.
5) On-chip memory shared by the GPP and the CCUs is used for program data.
6) All necessary data are available in the shared memory.
7) All transfers to/from the shared memory are performed sequentially.
8) Kernel's local data are stored in the FPGA's local memory, not in the shared memory.

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

9) The area constraints do not include the shape of the design.
10) The placement is decided by a scheduling algorithm such that the configuration latency is hidden.
11) The interconnection area needed for CCUs grows linearly with the number of kernels.

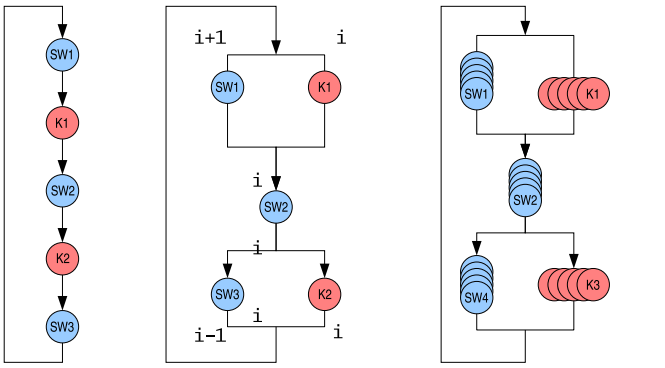
III. Motivation

The applications we target in our work have loops that contain kernels as well as pieces of software code in the loop body. In our previous work [10], [11] we focused on simple loops containing only one hardware kernel that would be accelerated on the FPGA and some software code that will always execute on the GPP. For this kind of simple loops, we proposed algorithms for loop unrolling and loop unrolling plus shifting to determine which would be the best unroll factor that would allow the maximum parallelization and performance. We want to extend the model to more generic loops with an arbitrary number of kernels and pieces of software code occurring in between the kernels, as illustrated in the example from Fig. 1a).

The example shows a loop with several functions – the SW_j functions are executed always on the GPP, while the K_i functions are the application kernels that are meant to be accelerated in hardware. These can be viewed as a task chain, where we assume that there are dependencies between consecutive tasks in the chain, but **not** between any two tasks from different iterations.

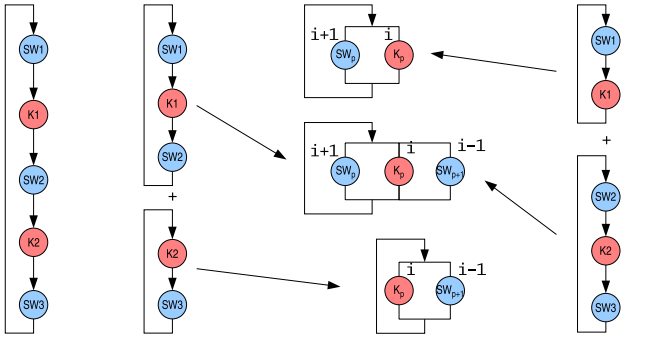
In Fig. 1b) we illustrate the execution pattern of the loop when the unrolling technique is applied: different instances of each software function are executed sequentially, and the different instances of each kernel are executed in parallel. In Fig. 1c) we illustrate the execution model of the loop when the simple loop shifting technique is applied. In this case, the first kernel of the loop body will execute in parallel with the first software function from a different iteration. The loop prologue and epilogue resulted from the shifting are not shown on the figure. The new loop body resulted when combining loop unrolling and loop shifting is shown in Fig. 1d).

a) Extended shifting: It is obvious that for loops containing more than one kernel, more parallelism can be exploited. The next natural step to do is to transform the loop such that each kernel executes in parallel with the preceding software function. The disadvantage of this idea is that it increases too much the loop prologue and loop epilogue when there are more than two kernels.



a) Loop b) Extended-shifted loop c) Unrolled and ext-shifted loop

Fig. 2. Applying extended shifting to a loop with two kernels



a)Loop b)Distributed loop(1) c)Execution model d)Distributed loop(2)

Fig. 3. Possibilities of loop distribution

Therefore, we propose the *extended shifting* to be the transformation that relocates only the software functions placed in the beginning and in the end of one loop iteration, in order to eliminate the data dependencies and allow the software and hardware functions to be executed in parallel.

In Fig. 2b) we illustrate the execution model of the loop when the extended shifting is applied. In this case, the first and the last kernels of the loop body will execute in parallel with software functions from different iterations. The new loop body resulted when combining loop unrolling and the extended loop shifting is shown in Fig. 2c).

b) Loop distribution: We have proven in [11] that for a loop containing a hardware kernel and a software function, it is always beneficial to apply loop shifting and parallelize (if the data constraints allow it). However, if a loop contains more than one kernel, it might be more beneficial to distribute it into smaller loops where different unroll factors might be applied due to different area or memory constraints for the kernels, leading to

better performance.

Assuming that between two kernel tasks there is a software task, a decision has to be made whether to distribute the software task with the first kernel or with the second one. This kind of decision has to be taken at each breaking point of the loop.

Figure 3a) shows a loop with two hardware kernels and three software functions. The possibilities of breaking this loop between the two kernels are illustrated in Fig.3b) and Fig.3d). In Fig.3c) we illustrated the parallel execution model for each of the three cases of a loop containing one hardware kernel inside, depending on the position of the software code – where $i-1$, i, and $i+1$ are iteration numbers.

We consider that a performant distribution algorithm is a Deep First Search algorithm, applied to the sorted list of kernels. The sorting is performed according to a heuristic based on the hardware execution time and the memory-constrained maximum unroll factor for each kernel.

IV. Experimental results

For the experimental results, we created a random test generator. We generated loops with sizes between 2 and 8 (the size is given by the number of kernels inside) and for each loop size we created 600 tests.

For each test, the performance improvement has been estimated based on the algorithms for parallelization through extended shifting and parallelization through loop distribution. A test case is determined by the following parameters:

- N - the number of iterations;
- $T_{\text{sw}}[j]$ - the execution time for each software function $\texttt{SW}_j$ (cycles);
- `MAX(sw)` - the maximum of the execution times of the software functions (cycles):

$$\texttt{MAX(sw)} = \max_j(T_{\text{sw}}[j]);$$

- $T_{K(\text{sw})}[i]$ - the execution time in software for each kernel $\texttt{K}_i$ (cycles);
- $S[i]$ - the speedup for $\texttt{K}_i$ kernel ($S[i] \in \mathbb{R}$);
- $A[i]$ - the area occupied by K_i in hardware, in percents ($A[i] \in \mathbb{R}$);
- $T_{K(\text{hw})}[i]$ - the execution time in hardware for $\texttt{K}_i$ (cycles):

$$T_{K(\text{hw})}[i] = (\text{int})\ \frac{T_{K(\text{sw})}[i]}{S[i]};$$

TABLE I
Parameter values

Parameter	Min. value	Max. value
N	64	256*256
$T_{\text{sw}}[j]$	0	800
$T_{K(\text{sw})}[i]$	1.5 * `MAX(sw)`	41.5 * `MAX(sw)`
$S[i]$	2.0	10.0
$A[i]$	1.2%	12.0%
$T_{\text{r}}[i]$	1	$T_{K(\text{hw})}[i]/3$
$T_{\text{w}}[i]$	1	$T_{K(\text{hw})}[i]/6$

- $T_{\text{r}}[i]$ - the time for memory read for $\texttt{K}_i$ running in hardware (cycles);
- $T_{\text{w}}[i]$ - the time for memory write for $\texttt{K}_i$ running in hardware (cycles);
- $T_{\text{c}}[i]$ - the time for computation for $\texttt{K}_i$ running in hardware (cycles):

$$T_{\text{c}}[i] = T_{K(\text{hw})}[i] - T_{\text{r}}[i] - T_{\text{w}}[i].$$

The values for the presented parameters have been generated according to the Table I. Note that in some cases, the maximum value of a parameter depends on the generated value of another parameter (for instance, the kernel time for read is at most one third of the total execution time of the kernel in hardware).

In Fig. 4 we illustrate the performance improvement for the different loop sizes when applying the loop distribution method, compared to the extended shifting method. The results show that there is no performance improvement in more than 55% of the cases for loop size equal to 2, but the greater the loop size, the less cases without improvement (less than 10% for loop sizes 7 or 8). More than 20% of the test cases for loop size greater than 2 present an improvement between 10 and 20% and more than 20% of the test cases for loop size greater than 3 present an improvement between 20 and 30%. A speedup of more than 50% is achieved in approximately 2% of the test cases.

V. Conclusion

In our previous work we discussed the performance enhancement obtained by parallelizing a loop with a hardware kernel and some software code, using loop unrolling and loop shifting. In our ongoing work we analyze the effects of extending these transformations to loops containing several kernels and pieces of software code. One solution is to extend the shifting technique in

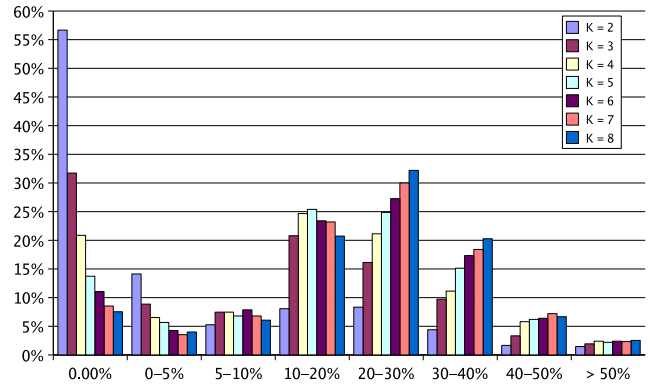


Fig. 4. Performance distribution

order to execute in parallel with the software both the first and the last kernels in the loop. Another solution is to distribute the loop and apply the loop unrolling and loop shifting transformations to the resulted smaller loops. Preliminary results on randomly generated tests show that there is potential for improving the performance by distributing the loops. However, we are aware of the fact that the loop distribution overhead can be quite expensive, depending on the amount of intermediary results that are needed.

REFERENCES

[1] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, November 2004.

[2] E. M. Panainte, K. Bertels, and S. Vassiliadis, "The PowerPC backend Molen compiler," in *Proceedings of the 14th International Conference on Field-Programmable Logic and Applications (FPL'04)*, August 2004, pp. 434–443.

[3] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, J. Lu, and S. Vassiliadis, "DWARV: DelftWorkbench automated reconfigurable VHDL generator," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL'07)*, August 2007, pp. 697–701.

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

[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," in *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, February 2004, pp. 114–119.

[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," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, March 2003, pp. 296–301.

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

[8] M. Weinhardt and W. Luk, "Pipeline vectorization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 234–248, February 2001.

[9] J. Liao, W.-F. Wong, and T. Mitra, "A model for hardware realization of kernel loops," in *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL'03)*, September 2003, pp. 334–344.

[10] O. S. Dragomir, E. Moscu-Panainte, K. Bertels, and S. Wong, "Optimal unroll factor for reconfigurable architectures," in *Proceedings of the 4th International Workshop on Applied Reconfigurable Computing (ARC'08)*, March 2008, pp. 4–14.

[11] O. S. Dragomir, T. Stefanov, and K. Bertels, "Loop unrolling and shifting for reconfigurable architectures," in *Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL'08)*, September 2008.