Extending Loop Unrolling and Shifting for Reconfigurable Architectures

Ozana Silvia Dragomir[*,1],
Koen Bertels[*,1]

[*] *TU Delft, Mekelweg 4, 2628CD, Delft, The Netherlands*

ABSTRACT

Loops are an important source of optimization. In this paper, we propose an extension to our work on loop unrolling and loop shifting for reconfigurable architectures. By applying unrolling and shifting to a small loop containing a hardware kernel and some software code, we relocate the function calls contained in the loop body such that in every iteration of the transformed loop, software functions (running on the GPP) execute in parallel with multiple instances of the kernel (running on FPGA). For larger loops containing an arbitrary number of kernels with pieces of code occurring in between the kernels, we study the effects of splitting the loop and applying the unrolling and shifting technique to the small achieved loops.

KEYWORDS: loop optimizations, reconfigurable computing

1 Introduction

Loops represent an important source of optimization in many real life applications. 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 performance. Although some loop transformations may not beneficial in most compilers because of the large overhead that they introduce when applied at instruction level, they may show a great potential for improving the performance when applied at a coarse-level (*i.e.*, function level). Note that in the rest of this paper we will refer to the generic loop shifting (when the shift is more than 1) as *loop pipelining*.

Target code. The applications we target in our work have loops that contain kernels inside them. In our previous work [Drag08a, Drag08b] we focused on simple loops containing only one hardware kernel that would be accelerated on the FPGA and some software code that

[1]E-mail: {O.S.Dragomir, K.L.M.Bertels}@tudelft.nl
This work is supported by the FP6 EU project hArtes, with integrated project number 035143.

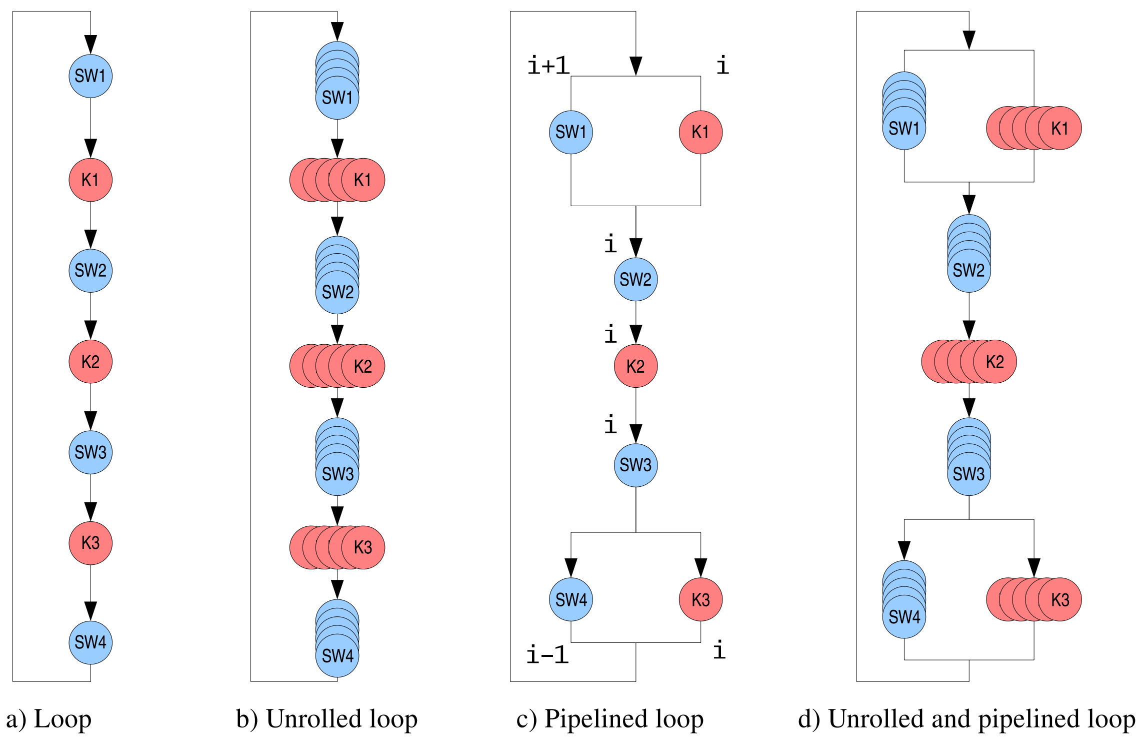


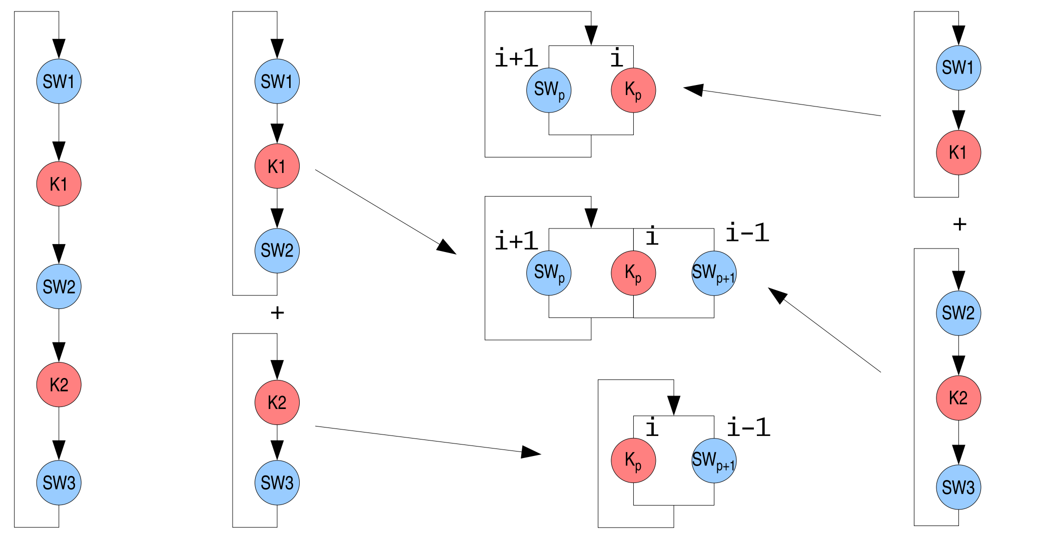
Figure 1: Loop containing several kernels

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 $\mathtt{SW}_j$ functions are executed always on the GPP, while the $\mathtt{K}_i$ 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 loop shifting technique 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 loop prologue and epilogue resulted from the pipelining are not shown on the figure. The new loop body resulted when combining loop unrolling and loop pipelining is shown in Fig. 1d).

Target architecture. Our target architecture is Molen [Vass04], 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:



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

Figure 2: Possibilities of loop distribution

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.
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.

2 Overview of the problem

In our current work, we analyze loops that contain an arbitrary number of tasks, which can be either kernels or software functions. We consider that all the software code between two kernels in the loop body is a software task. We have already proven in [Drag08b] that for a loop containing a hardware kernel and a software function, it is always beneficial to apply loop shifting (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.

The problem of deciding how to partition the loop into smaller loops is not trivial, as it is possible that between any two consecutive kernel tasks there is a software task. Then, in case of splitting the loop between the two kernels, a decision has to be made whether to distribute the software task with the first kernel or with the second one. Figure 2a) shows a loop with two hardware kernels and three software functions. The possibilities of splitting this loop between the two kernels are illustrated in Fig.2b) and Fig.2d). In Fig.2c) 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.

3 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 these transformations on loops containing several kernels and pieces of software code. Preliminary results on randomly generated tests show that there is potential for improving the performance by splitting such loops and applying the loop unrolling and loop shifting transformations to the resulted smaller loops.

References

[Drag08a] O. 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)*, pages 4–14, March 2008.

[Drag08b] O. 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) (to appear)*, September 2008.

[Vass04] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte. The MOLEN Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363–1375, November 2004.