Transformations for polyhedral process networks

Meijer, S.

Chapter 7

Conclusions

In this dissertation, we addressed the problem of how to transform a Polyhedral Process Network (PPN) in order to meet performance/resource constraints. Transformations are crucial because deriving PPNs from a sequential program specifications without performing any transformations does not guarantee that the resource and/or performance constraints are met. The reason is that the `pn` compiler creates one process in the parallel application specification (PPN) for each program statement in the sequential program. As a result, the derived PPN and its processes can be highly imbalanced as some program statements can be much more computationally intensive than others. Therefore, compile-time analysis of PPNs and transformations should assist the designer in transforming the PPN when some design constraints are not met.

The research work presented in this dissertation mainly focused on how the process splitting and merging transformations should be applied to achieve the best possible performance results. The process splitting transformation creates more processes in a given PPN to exploit more data-level parallelism in the application. The process merging transformation is used to reduce the number of processes in a PPN. Before our work presented in this dissertation, i.e., our compile-time approaches to evaluate the process splitting and merging transformations, the problem was that the transformations were defined but it was the designer's responsibility to apply them. We have shown in this dissertation that it is not trivial for a designer to apply these transformations. The reason is that there are many possibilities to apply a particular transformation and many factors influence the final performance results. As a consequence, there can be great differences in the achieved performance results, and they also can easily get worse than the results of the initial PPN if the transformations are not applied carefully. To assist the designer in transforming a PPN, we have defined metrics that are important for the final performance results. Furthermore, we pre-

sented compile-time approaches to evaluate these metrics, such that the designer can select the best possible alternative. For the process splitting transformation discussed in Chapter 3, the analysis is performed locally on the process, while a throughput model for PPNs has been introduced for evaluating the process merging transformation in Chapter 4. Based on the results of the work presented in Chapters 3 and 4, we draw the following conclusion.

- **Conclusion I:** by defining all major factors that are important for the process splitting/merging transformation, and by taking into account the target platform characteristics, we can, at compile-time, evaluate and correctly predict how the process splitting/merging transformation should be applied to obtain the best performance results.

Compile-time hints to transform PPNs in a particular way were missing in the Daedalus tool-flow, as it could only explore different platform and mapping specifications. Thus, the research work presented in this dissertation addresses one very important aspect of the Y-chart approach, i.e., to evaluate and change the application specification after performance analysis. With our compile-time approaches, we can evaluate the process splitting and merging transformations, such that the best option to apply a transformation can be selected. Changing the application specification was identified in Chapter 1 as an important step in order to obtain a desired design point.

Besides approaches to help the designer in evaluating and applying the process splitting and merging transformations in isolation, we have also devised a holistic approach in Chapter 5 that combines both transformations. This solved the problem of ordering the process splitting and merging transformations, which is a difficult problem as there are many alternatives to apply the transformations one after the other and with different parameters. Furthermore, we solved the problem of selecting the processes on which a transformation should be applied.

- **Conclusion II:** by first splitting up all processes and by subsequently merging the different process instances into load-balanced compound processes, we solved the problem of ordering the different transformations and also on which process a particular transformation should be applied to obtain the largest positive performance impact.

There are two perspectives to look at our approach to combine the process splitting and merging transformations. The first one is presented in Chapter 5, i.e., to consider the combination of transformations as an optimization after the initial PPN has been derived. The second perspective is to look at this as an approach to derive PPNs in a different way than currently implemented in the `pn` compiler. That is, instead of creating one process for each program statement in the sequential application, a number of compound processes are created that contain a number of executions of

all program statements. Then, the designer will not be confronted with the initial PPN, but only with the transformed and load-balanced PPN. However, we did not emphasize on this perspective in this dissertation as this requires more research on the number of compound processes to be generated. Choosing the number of processes could be the responsibility of either the designer or the compiler, but the latter is clearly the preferred option as it may not be straightforward for the designer to decide when saturation of the performance occurs. For example, cycles in a PPN may, or may not lead, to sequential execution of the processes involved in the cycle. When the processes in a cycle execute sequentially, then we refer to it as a true cycle. Splitting the processes involved in true cycles would only introduce more processes and not improve the performance, because the processes already execute sequentially as we have also explained in Chapter 5. On the other hand, when the process executions in a cycle overlap, then the splitting transformation can result in performance gains. However, how much can be gained depends on the behavior of that cycle.

- **Conclusion III:** with our holistic approach that combines the splitting and merging transformations, we exploit all available data-level parallelism to the maximum such that our approach gives the best performance results using the two considered transformations when there is something to be gained, and the same performance results as the initial PPN when there is nothing to be gained.

In order to know how much can be gained by splitting processes, the behavior of the (self)-cycles that restrict the data-level parallelism in a certain way must be investigated. We sketched an approach how to detect true cycles, but left the question how many times a process should be split-up for future research.

In Chapter 6, we have presented two approaches to execute PPNs on commercial off-the-shelf (COTS), programmable MPSoC platforms, i.e., the Intel IXP network processor and the Cell platform. While the IXP has hardware support for FIFO communication to some extent, this is completely absent in the Cell platform. Thus, both the Cell and the IXP platform do not support the operational semantics of the PPN model of computation as efficiently as the ESPAM platform, which is especially tailored to execute PPNs as efficiently as possible. To make the FIFO communication more efficient on the Cell, we deployed an approach to transfer multiple data tokens when only one is requested by a consumer process. Thus, by grouping multiple tokens into one package, less FIFO read/write accesses need to be performed during the execution of a PPN. The execution of PPNs on the Cell and IXP processors enabled us to compare the execution times with the ESPAM platform. In Chapter 6, we showed that the ESPAM platform always gives the lowest cycle count, while the Cell is better in terms of execution times as a result of its very high clock frequency.

- **Conclusion IV:** The cycle count for PPNs executed on the ESPAM platform is always lower compared to the IXP and Cell platforms. It does not provide

the fastest execution times since its clock frequency is restricted to 100 Mhz, only because it is prototyped on an FPGA. With higher clock frequencies (e.g., an ASIC implementation, or advances in FPGA technology), the ESPAM platform would not only be the most efficient, but also the best platform to obtain the lowest execution times.

Thus, the most benefit from executing PPNs onto MPSoC platforms is obtained when the operational semantics of the PPN model of computation are supported by the target platform. The IXP processor, for example, runs at 600 Mhz which is 6 times higher than the ESPAM platform. Despite this higher clock frequency, however, the execution times are worse than for the ESPAM platform. The reason is that FIFO communication is supported to some extent, but not as efficiently as on the ESPAM platform. In the Cell platform on the other hand, FIFO communication is completely implemented in software. This makes the ESPAM platform the most efficient platform because it is especially tailored to execute PPNs and supports FIFO communication with hardware components. The Cell's clock frequency is 30 times higher than the ESPAM platform, but its performance results are only 10 times better. The only reason that the ESPAM is restricted to 100 Mhz, is because it is prototyped on FPGA technology and not in ASIC such as the Cell. If the ESPAM platform is implemented using ASIC technology, then it would not only be the most efficient, but also the fastest.

Finally, we remark that it is not beneficial for all applications to be executed as PPNs on MPSoC platforms. With the experiments in Chapter 6, we showed that performance results can also get worse compared to the sequential versions of the applications.

- **Conclusion V:** for applications with very fine-grain computations, and/or target platforms with high synchronization and communication costs, the gain of parallelization can be canceled by the costs for synchronization/communication.

In Chapter 5, we presented an approach to create compound processes by using the process splitting and merging transformations in combination. In that work, we assumed that the designer selects the number of compound processes to be created, which can, for example, be the number of available processors in the target platform. In our future work, we want to investigate if we can decide at compile-time how many compound processes to create before saturation of the system performance occurs. This optimization could, for example, result in a number of compound processes for a given PPN that is less than the available processors, which means that the other processors are available for other applications. Thus, we want to investigate how the maximum parallelism available in an application can be determined, and how it

can be exploited using the minimum number of resources by applying the process splitting and merging transformations.