Chapter 7

Summary and Conclusions

In this dissertation, we addressed the problem of automated derivation of Polyhedral Process Network (PPN) specifications from *dynamc* sequential programs. Our work is essential for the development of parallelization compilers exploiting task-level parallelism inherent to many dynamic applications. As an example, the work presented in this dissertation inspired the development of further extensions in the *pn* [3, 48] compiler, where at the moment, most of the research done in this dissertation has been implemented. The derivation of a parallel specification described by our approach is an essential for the systematic and automated design of the emerging embedded systems-on-chip platforms. In designing the platforms the parallel specification allows for systematic and efficient exploration and mapping of the application onto the platform. As an example, the work presented in this dissertation is used in a methodology, implemented in a system-level design tool-flow called DAEDALUS [57, 58], for automated design, programming, and implementation of MPSoCs starting at a high level of abstraction. The methodology is built on the concept of Platform-Based Design (PBD) [59] being a promising new approach to master the ever growing complexity of today's embedded systems.

Many system-level design flows and application modeling and exploration approaches reported in the literature use the Kahn Process Network (KPN) [7] model of computation for a parallel application specification. In this dissertation, we target Polyhedral Process Networks (PPNs) [6] which is a special case of the KPN model. The PPN allows to specify an application, manipulate and optimize its representation in terms of polyhedra. This model is well suited for data-flow dominated applications in the realm of multimedia, imaging, and signal processing that naturally contain tasks communicating via streams of data. In this dissertation, we target such applications as being natural for extracting task-level parallelism.

The work presented in this dissertation is directly related to previous works on systematic and automated derivation of process networks from affine nested loops programs initiated by Rijpkema et al. [15, 30]. Further, Turjan et al. [14] proposed an automated derivation of process networks from a class of application called *static* affine nested loop programs (SANLPs). In SANLPs the memory array subscripts, loop bounds and conditional control structures are affine constructs of surrounding loop iterators, program parameters and constants. Also, they put a restriction on the input program to be *static* in order to enable the automatic analysis and conversion of the input program to a PPN. Although, many scientific, matrix computation, and signal processing applications can be specified as SANLPs, the *static* restriction limits the applicability of their approach, i.e., their approach cannot handle applications that have adaptive and dynamic behavior, such as multimedia applications (MPEG coders/decoders, Smart Cameras, Software Radio), adaptive filters, iterative algorithms, etc. Therefore, in this dissertation, we addressed the important question: whether some of the static restrictions of the SANLPs can be relaxed while keeping the ability to perform compile-time analysis and to derive PPNs in an automated way. Achieving this would significantly extend the range of applications that can be parallelized in an automated way.

By studying different dynamic applications we distinguished three relaxations to SANLP programs that would allow one to specify dynamic applications as sequential programs. These relaxations are:

I. dynamic `if`-conditions are allowed in a program;

II. for-loops with dynamic bounds are allowed in a program;

III. while-loops are allowed in a program.

In [21], the first relaxation has been considered: a novel automated procedure has been developed that derives a PPN from a class of affine nested loop programs called *Weakly Dynamic Programs* (WDPs). In this class of programs, the `if`-conditions might be dependent on some information that is unknown at compile-time and may change at run-time. In this dissertation, we have considered the other two more difficult relaxations. In Chapter 3, we considered relaxation II and presented a first approach for automated translation of affine nested loops programs with dynamic loop bounds (`Dynloop`) into input-output equivalent PPNs. Relaxation III, we considered in Chapter 4 by presenting a novel approach for automated translation of affine nested loop programs with while-loops (WLAPs) into input-output equivalent PPNs.

In contrast to deriving a PPN specification of a SANLP program, converting dynamic programs into PPNs in a systematic and automated way is a challenging and complex problem. This stems from the fact that the exact behavior of a dynamic program is unknown at compile-time. Therefore, for example, formal tools such as Exact Array Dataflow Analysis cannot be used to extract dependence relations in a dynamic program as this has been shown in Section 1.3. In Chapters 3 and 4, we demonstrated that although the exact behavior of dynamic programs with relaxations II and III is unknown at compile-time, still such programs can be analyzed and converted to an executable PPN specification in a systematic and automated way.

At a high-level, our approaches presented in Chapters 3 and 4 are similar and consist of three main steps. First, we extract an *approximated* dependency relation information from a dynamic program using the Fuzzy Array Dataflow Analysis (FADA) [37, 38] technique. We explain what approximation means. In Section 1.3 we demonstrated the difference of dependency patterns between dynamic and static programs. In static programs, different instances of a program correspond to one and the same single dependency pattern which is known at compile-time. In dynamic programs, data dependency patterns correspond to different *instances* of a dynamic program, and are unknown at compile time. This also means that the exact data dependency patterns in a dynamic program cannot be determined at compile-time. Therefore, we parameterize or *approximate* them using parameters whose values have to be set dynamically at run-time in order to preserve the initial data-flow in a program.

Second, we translate the initial program into dynamic Single Assignment Code (dSAC) [21] form and implement the general approach how to set the values of parameters introduced by FADA at run-time. A dSAC program is input-output equivalent to the corresponding dynamic program and it has the property that every data variable or an array element is written *at most once*. This implies that some variables may not be written at all. Also, at this step, the storage structure of the initial application is transformed such that each pair of statements communicates data over a dedicated multidimensional array.

Third, we demonstrate how the topology of the corresponding PPN and the code executed in each process are derived. Additionally, because the target model, Polyhedral Process Network (PPN), requires FIFO channels as communication medium, at this step memory array accesses are converted into managed dataflow over FIFO queues. Mapping such array communication onto FIFO channels requires complex address generators, especially if the arrays have multiple dimensions. Therefore, in Chapter 5, we addressed the Communication Model Identification (CMI) problem, which investigates communication characteristics of each Producer/Consumer (P/C) pair.

On the basis of the results obtained from Chapters 3,4,5,6 and experimental results, we can draw the following conclusions:

Conclusion I The work presented in this dissertation can be implemented efficiently in a compiler that will help to reduce significantly the time for parallelizing sequential dynamic programs. The *pn* [48] compiler which implements our approaches drew the attention of Intel Corporation and actually, Intel sponsored and evaluated these implementations.

Conclusion II With our approach that uses FADA analysis, we reveal all available task-level parallelism presented in the initial specification of a dynamic program. This allows for the utilization of multiple cores in an efficient way. This has been demonstrated in Chapter 6.

Conclusion III In some cases our parallelization approaches may exhibit some overhead introduced by creation of an excessive amount of control channels. This will result in more run-time communication of control data in comparison to the control data communication in a PPN carefully optimized and derived by hand. Therefore, some optimization techniques have to be added to our approach in order to improve the quality of the generated PPNs in terms of communication control overhead. The investigation of such optimization techniques has already been started and sponsored by Intel Corporation. Moreover, some of the optimizations are under development and test in the current version of the *pn* [48] compiler.

Conclusion IV The control FIFO channels appear in a PPN derived from dynamic programs because the behavior of these programs is not completely known at compile-time. The presence of control FIFO channels introduces extra workload and communication overhead that are the consequences of the dynamic nature of the initial application. The analytical analysis conducted in Section 3.7 showed that the effect of these extra control structures and operations (overhead) on the performance of the PPN significanlty decreases when the granularity of the function calls executed inside the processes increases.

In the work presented in this dissertation, we considered a parallelization strategy that exploits task-level parallelism. Although, our approaches extract the maximum parallelism available, some other techniques can be used to extract other types of parallelism. For example, in future work, one can investigate on the transformations of PPNs similar to [20, 60], where a data-level partition strategy can be considered in order to achieve better execution performance and to automatically derive PPNs from dynamic applications.