Estimation and Optimization of the Performance of Polyhedral Process Networks

Haastregt, S. van

CHAPTER 1

INTRODUCTION

ELEMENT NUMBER 14, or silicon, has been important for many ancient civilizations, albeit mostly as a constituent of sand and rocks. Silicon was essential for the construction of houses, temples, and roads, which together formed the centers of society. In 1954, a new and very different use for silicon was found that would have a dramatic impact on the established centers of society: Gordon Teal and his team produced the first silicon transistor [Che04]. Many electronic devices have become available since then, in which silicon transistors are an essential component. By miniaturization, more and more transistors could be fit onto a small area, thereby enabling the construction of complex processing systems. Contemporary examples of such processing systems include the special purpose processors found in automotive, mobile communications, medical, industrial, and entertainment application domains. Many of these processing systems are tightly coupled to their environment and perform a specific task, and are therefore classified as *embedded systems* [LS11, Mar11, SB00]. Central to this dissertation is the design of the special purpose processors in these embedded systems.

1.1 Problem Context

The special purpose processors in embedded systems are highly optimized to perform their application-specific computations in a fast and area- and energy-efficient way. The design of those processors is becoming increasingly challenging due to increasing application complexity, the ever-increasing demand for computational power, and worldwide time-to-market pressure. To satisfy the demand for computational power, *Multi-Processor System-on-Chip* (*MPSoC*) solutions are deployed in modern

embedded systems. Such MPSoCs consist of many different components such as programmable processing components, specialized processing components, memory components, and input/output interfaces. By letting multiple components work in parallel, the demand for computational power is met. Unfortunately, the design of an MPSoC is even more challenging than the design of a single-processor system. The challenge for the designer is to distribute computations over different processors of the MPSoC. While doing so, the designer should guarantee functional correctness of the system and at the same time make tradeoffs between orthogonal design aspects such as circuit area and performance [Mar06]. Thus, the shift to multi-processor systems may address the demand for computational power, but this comes at the expense of a further increase in design complexity.

Traditionally, processors have been designed at the *Register Transfer Level* (*RTL*). An RTL specification of a processor consists of registers that are interconnected by signals and combinational logic. RTL design of modern MPSoCs is becoming increasingly error-prone and time-consuming because of the abundance of registers, signals, and combinational logic needed for a modern MPSoC's functionality. To cope with the design complexity of modern MPSoCs, the designer needs to work at a level of abstraction above the RTL. This has led to the emergence of *Electronic System-Level* (*ESL*) design methodologies [GAGS09, BM10]. In such a design methodology, the designer first specifies a system at a high level of abstraction. Next, the designer constructs an RTL implementation from the initial specification with the aid of system-level design automation tools.

An example system-level design tool set is the open-source Daedalus tool set which has been developed at the Leiden Embedded Research Center (LERC) [NSD08b, Lei08]. We leverage the Daedalus tool set in this thesis. This means that we want to develop the special purpose processors of an embedded system with Daedalus. An overview of the Daedalus system-level tool flow is depicted in Figure 1.1. Daedalus enables a designer to obtain a deployable gate-level specification from a system-level specification in a fully automated way. The functional behavior of the system-level specification is specified as a sequential C program, as shown at the upper right part of Figure 1.1. The elaboration from one specification level to a lower specification level is done in a fully automated way. We discuss the different aspects of Daedalus in the following paragraphs.

Many applications in the embedded systems domain are specified using an imperative model of computation, in for example the C language. Such models are well-suited and widely adopted to specify the functionality of single-processor systems. Unfortunately, mapping an imperative specification onto a multi-processor system is difficult because of two mismatches. First, the sequential nature of an imperative specification does not match the parallel nature of a multi-processor system. Sec-

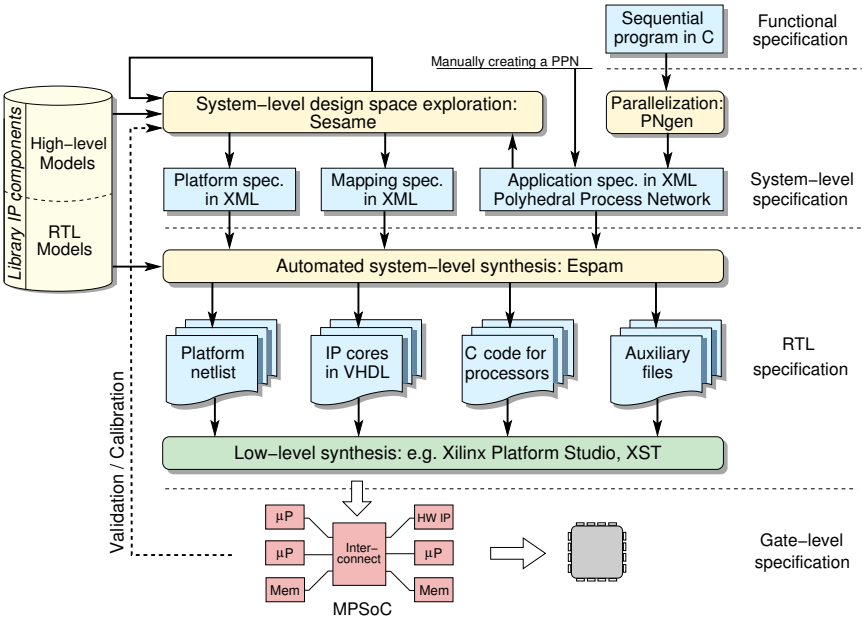

Figure 1.1: Daedalus system-level tool flow overview [NSD08b].

ond, an imperative specification assumes shared memory which is likely to become a performance bottleneck on a multi-processor system. A distributed memory model better matches a multi-processor system, but it is not possible in the general case to extract a distributed memory model from an imperative specification.

The functionality of a multi-processor system is more naturally specified using a parallel model of computation such as a network of processes communicating over channels. A model that has gained widespread popularity is the Kahn Process Network model [Kah74]. Specifying the functionality of a system using a parallel model of computation is considered more difficult compared to using an imperative model of computation. This is because the human brain tends to solve problems as a sequence of steps, which matches the sequential nature of an imperative model of computation. Moreover, in a parallel specification deadlocks and race conditions may occur that are very difficult to detect or predict beforehand [Mar06]. Such difficulties do not occur in a sequential specification. As such, many designers prefer specifying an application using a sequential specification, despite the subsequent difficulties of implementing the specification as an MPSoC. The mismatch between the programmer-preferred sequential specifications and the parallel specifications desired for multi-processor systems is known as the *specification gap* [Ste04].

Various approaches exist to bridge the specification gap. One approach is to extend a sequential program with library calls or compiler pragma directives to indicate tasks that can execute concurrently. Examples of this approach include pthreads, OpenCL [Khr08], and OpenMP [Ope97]. Another approach is to automatically extract concurrent tasks from a sequential program using a parallelizing compiler such as LooPo [GL97], Polaris [BEF+94], Pluto [BBK+08], or PNGEN [VNS07]. The latter is part of Daedalus to bridge the specification gap. PNGEN generates a parallel specification from a sequential program written in a subset of the C language. We discuss PNGEN in more detail in Section 2.3.

A system-level specification lacks many details that are present in the RTL specification because these details are irrelevant at the system level. For example, at the system level the designer reasons about sending data from one processor to another without specifying the registers and logic that implement such communication in the RTL specification. Not exposing the designer to such implementation details allows a designer to better cope with complex systems. However, the omission of implementation details opens up a gap between the system-level specification and the RTL implementation, which is known as the *implementation gap* [NSD08b]. To obtain a functional implementation from a system-level specification, the implementation gap needs to be bridged by adding low-level implementation details to the system-level specification. This is done by a system-level synthesis tool which refines a system-level specification into an RTL specification in a systematic and automated way.

The Daedalus tool set provides the ESPAM tool for automated system-level synthesis. A system-level specification for ESPAM is composed of three individual specifications: an implementation platform specification describing the number and types of processing and interconnect components of the system; a parallel application specification consisting of a network of communicating tasks; and a mapping specification that maps the application tasks onto processing components. The ESPAM tool generates an RTL specification from the three specifications. This RTL specification is then taken through commercial *low-level synthesis* tools that convert the RTL into a gate-level specification. Place-and-route tools take such a gate-level specification and create a layout of the circuit which can be implemented on a *Field-Programmable Gate Array* (*FPGA*) or provided to an *Application-Specific Integrated Circuit* (*ASIC*) manufacturing process. This last step yields a complete MPSoC implementation.

1.2 Problem Statement

Existing system-level design tools such as Daedalus present a forward synthesis flow to bridge the specification and implementation gaps. This allows a designer to obtain

a working prototype of a system in only a few hours of time [NSD08b]. However, many different implementations of an application specification are possible that have identical functionality but differ in performance and implementation cost aspects. This presents the designer with another problem: selecting an implementation from a vast *design space* of possible implementations. Only a subset of the *design points* in this design space represent implementations that satisfy a set of given *design constraints* on performance and circuit area. Thus, solely closing the specification and implementation gaps still leaves open the problem of selecting the design point that best matches a set of design constraints.

A Daedalus system-level specification consists of the application, platform, and mapping subspecifications, as described in the previous section and shown in Figure 1.1. Each of these subspecifications may be transformed to yield a functionally equivalent implementation that has different performance and resource cost properties, as described by the Y-chart approach [KDWV02]. For example, a designer can transfrom the platform specification by adding or removing processors, or transform the mapping specification by moving a task from one processor to another, or transform the application specification by splitting a tasks into smaller subtasks and thereby exposing more parallelism. Many combinations of such *transformations* are possible and this number grows rapidly as application and platform sizes increase. As a result, the design space for a modern MPSoC is typically very large.

Despite the existence of fully automated system-level synthesis tools, implementing and evaluating all design points is infeasible for modern MPSoC design because of the large design space. Therefore, the design space should be explored in such a way that only the "promising" design points need to be implemented and evaluated. Finding the promising design points is a non-trivial multi-objective optimization problem. Many *Design Space Exploration* (*DSE*) techniques have been proposed to efficiently search large design spaces [Gri04]. Daedalus incorporates the SESAME DSE tool to explore the design space using an evolutionary algorithm [PEP06]. SESAME relies on trace-based simulation to estimate the performance of candidate design points. Although SESAME's simulation is intended for fast performance analysis, conducting many simulations may still take a considerable amount of time [PP12]. This leads to unreasonably long design times.

An alternative way of finding a satisfactory design point is the (naive) iterative design flow depicted in Figure 1.2. The design flow starts with an initial system-level specification. A parallel application specification is automatically derived from an imperative program using for example PNGEN, thereby bridging the specification gap. The designer synthesizes this system-level specification into an FPGA prototype to verify if for example performance constraints are satisfied. The designer uses a system-level synthesis tool such as ESPAM in this step, thereby bridging the

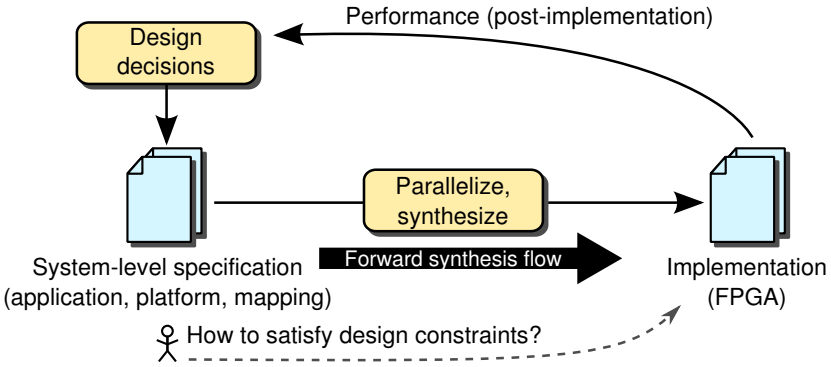

Figure 1.2: Iterative system-level design flow.

implementation gap. If a performance constraint is not satisfied, the system-level specification is transformed based on performance and cost metrics obtained from the prototype implementation. These transformations entail modifying the application by rewriting the C code, modifying the platform by adding processors, or modifying the mapping by assigned tasks to different processors. The designer relies on experience and expertise to come up with transformations that most likely have the desirable effect on performance and cost aspects. Building up this knowledge is referred to as the "acquisition of insight" [Spe97]. However, it is not trivial to predict beforehand if and by how much a certain transformation affects performance and cost aspects. At this moment, the best a designer can do is to perform a new time-consuming synthesis step after transforming the system-level specification. This procedure is repeated until an implementation is obtained that satisfies performance constraints. The designer can then proceed with the actual manufacturing of the system.

A naive iterative design flow may appear to be more deterministic than a random-search driven DSE flow. Because the designer iteratively transforms a system-level specification in a pragmatic manner, a system that satisfies all performance constraints should eventually be the result. However, this only holds if the designer always makes the optimal decisions. This does not always happen in practice, because the designer may for example overlook solutions or ignore solutions that seem counter-intuitive. Another problem with a naive iterative design flow is that evaluation of a single specification may easily take a few hours of time. This reduces the number of iterations a designer can make in a given time frame, increasing time-to-market.

The naive iterative design flow bridges both the specification and implementation gaps by employing advanced parallelizing compilation techniques and system-level

synthesis tools. However, it does not address the following problem: given a performance constraint, which transformations should the designer apply to obtain an implementation that meets this performance constraint? For example, consider the scenario in which a designer constructs a video processing system under the constraint that the system should meet a throughput of 20 frames per second. After synthesizing the system-level specification, the designer finds that the system works at only 11 frames per second. This puts a burden on the designer to transform the system-level specification such that the performance constraint of 20 frames per second is met. We therefore argue that solely bridging the specification and implementation gaps is not sufficient to solve a design problem.

In this dissertation, we consider the iterative system-level design flow of Figure 1.2 and address a designer's problem that is currently not addressed. That is, we ask how to modify this design flow to obtain a constraint-satisfying implementation of a system in a short amount of time. This modification is needed as synthesizing a design in the current flow takes too long, keeping the designer in the dark whether the design will satisfy the designer's constraints. Performance estimation methods are lacking that could provide an early indication of whether a design will satisfy the designer's constraints at all. After obtaining an implementation not meeting the constraints, there is little guidance to help a designer transform his design in such a way that his performance constraints will be satisfied. In this context, we formulate our three central research problems as follows:

1. **Synthesis**: How to automatically obtain efficient RTL implementations from a high-level specification that enable application of established transformations such as splitting, merging, stream multiplexing, and scheduling?
2. **Performance estimation**: How to assess the absolute performance of a design point, possibly in different ways by trading off evaluation speed against accuracy?
3. **Optimization**: How to obtain an implementation that satisfies a performance constraint while reducing the number of design iterations?

Only after addressing these three problems from a designer's perspective, Daedalus can become a powerful system-level synthesis tool capable of solving design problems.

1.3 Related Work

We address the central problems listed above in this dissertation by leveraging and extending the underlying theory of the Daedalus methodology [NSD08b, Lei08]. The

Daedalus methodology addresses the problem of obtaining an efficient FPGA implementation from a high-level application specification in a short amount of time. As such, Daedalus provides an important stepping stone to address the three central problems, ultimately leading to an extended Daedalus design flow that also considers performance constraints.

Daedalus is only one of many methodologies to (semi-)automatically obtain special purpose processor implementations from high-level application and system specifications. In this section, we give a brief overview of related approaches to obtain RTL implementations from high-level application specifications. We discuss related high-level synthesis techniques in Section 1.3.1 and related electronic system-level synthesis techniques in Section 1.3.2. Related work specific to each of the three central problems is discussed separately in Chapters 3, 4, and 5.

1.3.1 High-Level Synthesis

Automated synthesis of RTL implementations from specifications above the register transfer level, known as *High-Level Synthesis* (*HLS*), has been subject of research since the late 1980s [MK88, MPC88, PK89]. Since then, many academic and commercial HLS tools have been developed. In 1994, electronic design automation company Synopsys released its Behavioral Compiler tool that is widely regarded as the first commercial HLS tool [CM08]. This tool took a behavioral description of a design in VHDL or Verilog as input and generated a cycle-accurate VHDL or Verilog description. During synthesis, the tool allowed the designer to trade off throughput against chip area. Since then, many different HLS tools have been released by different companies, with varying degrees of commercial success. As of 2013, three of the major commercially available HLS tools are Synopsis SynphonyC [Syn10], Xilinx Vivado HLS [Xil13], and Calypso Catapult [Cal11]. A difference with Behavioral Compiler is that modern commercial HLS tools have anchored on C, C++, or SystemC input specifications instead of input specifications using Hardware Description Languages like VHDL or Verilog [Fin10]. Meeus et al. conducted a comparison between twelve different commercial and open-source high-level synthesis tools [MVBG+12].

Next to commercial tools, numerous academic high-level synthesis tools have been developed. One of the early academic tools was Hercules [MK88] which has been integrated in the Olympus Synthesis System [MKMT90]. Olympus takes an input specification written in HardwareC and provides synthesis and simulation of designs. HardwareC is a C-like language in which a system is described as a set of concurrent modules. The modules are interconnected using communication primitives. This requires the designer to split the application functionality across different modules and

interconnect them manually using communication primitives. Advanced compilation techniques such as those employed by Daedalus allow tools to automatically derive such interconnects from a sequential specification. The ROCCC tool takes a subset of the C language as input and generates RTL targeted towards FPGAs [GNB08]. ROCCC requires that loop iterators are used in at most one array dimension. This poses a problem when expressing for example a loop skewing transformation. Such a restriction is not necessary in Daedalus as any affine expression of loop iterators is analyzable using the polyhedral model. Other HLS approaches that employ the polyhedral model include for example PARO [HRDT08] and MMALPHA [GQR03]. These approaches use functional languages as input, while commercial tools and Daedalus all use an imperative language. Related early work included modeling affine nested loop programs using uniform recurrence equations to generate systolic array implementations [Qui84]. The FCUDA approach takes C code annotated using NVIDIA's CUDA primitives and generates C code annotated with AutoESL pragmas to obtain an FPGA implementation [PGS+09]. This allows a designer to express parallelism in a single specification and target both GPU and FPGA platforms. Besides the main FPGA backend, Daedalus also includes a GPU backend, allowing a designer to also target both GPU and FPGA platforms. Unlike FCUDA, Daedalus does not require CUDA-like annotations of the C code.

High-level synthesis should not be confused with design entry using a high-level language, because the use of a high-level language does not necessarily imply that the design is specified at a high level of abstraction. For example, Handel-C is a subset of the C language with extensions to describe hardware succinctly [Pag96]. Parallel behavior is expressed using the Communicating Sequential Processes (CSP) model of computation [Hoa85]. Similar to RTL design, the designer should perform scheduling and pipelining manually, whereas this is performed automatically in an HLS flow. Cobble is a language similar to Handel-C [TCL05] with support for custom compilation schemes. This allows a designer to define how a particular pattern in the source program should be mapped to hardware. Cobble is compiled into Pebble, which is a simplified hardware description language supporting design parameterization and run-time reconfiguration [LM98]. MyHDL allows a designer to specify hardware in the Python language [Dec03]. MyHDL still requires the designer to specify the behavior of the hardware at the register-transfer level using constructs provided by the MyHDL Python package.

Daedalus may be regarded as an HLS tool to some extent, since it generates RTL from a specification in the C language when a process is mapped onto an application-specific hardware processor. But in contrast to a conventional HLS tool, Daedalus only generates the control path RTL of a hardware processor. Daedalus does not generate data path RTL, as it relies on the designer to provide IP cores that implement

the data path [NSD08a]. Moreover, Daedalus generates complete heterogeneous MPSoC implementations, which is a task that is beyond the scope of high-level synthesis. Another difference between HLS tools and Daedalus is the model used to represent applications. HLS tools predominantly employ *Control Data Flow Graphs* (*CDFGs*) [MPC88, CGMT09], whereas Daedalus employs a process network based model [VNS07].

1.3.2 Electronic System-Level Synthesis

During the late 1990s, electronic system-level synthesis gained interest of system designers as it provided means to cope with the increasing design complexity of systems. A system-level synthesis flow focuses on an entire system possibly containing programmable processors. In contrast, a high-level synthesis flow focuses on a highly optimized application-specific RTL architecture implementing one or more kernels. Many different system-level synthesis tools exist besides Daedalus. SystemCoDesigner takes a set of SystemC modules as input and implements a system by mapping these modules onto hardware and software components [KSS+09]. Ptolemy is an environment for simulation and prototyping of heterogeneous systems [BHLM94, EJL+03]. A system design in Ptolemy may consist of subsystems that employ different models of computation, such as continuous time or process network based models. PeaCE [HKL+08] provides a system-level design environment based on Ptolemy, but restricts itself to an extension of the synchronous data flow model and an extension of the finite state machine model. SystemCoDesigner, Ptolemy, and PeaCE require the designer to specify a system as a set of actors interconnected using communication channels, while Daedalus automatically derives actors (processes) and channels from sequential code. The StreamIt approach [GTA06] requires the designer to specify an application graph using actors and communication channels in a custom language. StreamIt employs the Synchronous Data Flow (SDF) model, which is more restrictive than the PPN model employed by Daedalus. The System-on-Chip Environment (SCE) [DGP+08] uses the SpecC language [ZDG97] to describe system behavior. The SCE design flow consists of similar design steps as Daedalus, such as parallelization, communication synthesis, and RTL generation. However, the parallelization step is automated in Daedalus, whereas SCE requires the designer to explicitly specify the system as a set of concurrent tasks interconnected using communication channels. The MPSoC Application Programming Studio (MAPS) [LC10] is a framework that aids the MPSoC designer with C application parallelization. The parallelization in MAPS is relies on profiling information, whereas parallelization in Daedalus is static. Like Daedalus, MAPS can also incorporate already parallelized applications specified as a (Kahn) process network. Unlike

Daedalus, MAPS does not provide an automated way to obtain a parallelized variant of a sequential application. The Multi-Application and Multi-Processor Synthesis (MAMPS) flow maps synchronous dataflow graphs onto homogeneous MPSoCs. In contrast, Daedalus uses the more expressive PPN model and targets heterogeneous MPSoCs. MAMPS on the other hand supports multiple applications at once, whereas the Daedalus version used in this thesis supports only one application at once, although the Daedalus^{RT} extension does support multiple applications [BZNS12].

Bluespec SystemVerilog (BSV) is a high-level hardware description language intended to describe complete systems [NC10]. In a comparison conducted by Nikolov et al., a C specification of an H.264 video decoder was implemented using both the automated Daedalus flow and as a semi-custom design in BSV [NRD$^+$09]. The Daedalus design employed programmable components as processing elements, on which the C specification was mapped. The Bluespec design employed dedicated hardware processing elements, requiring manual conversion of the C specification to BSV. The authors found that the design time for the Daedalus approach was roughly 6 times shorter than the design time for the Bluespec design. This difference was mainly caused by the manual conversion and verification in the BSV design. However, the shorter design time in Daedalus came at the expense of higher resource cost caused by the use of programmable processors. Replacing programmable processors with dedicated RTL cores may reduce the resource cost footprint in the Daedalus flow. Such cores can be obtained automatically from C using HLS tools. We discuss the integration of HLS in Daedalus in Chapter 3.

Several ESL tools focus on graphical entry of a system-level design. For example, a system is specified in Koski using Unified Modeling Language (UML) [KKO$^+$06]. Xilinx System Generator provides a block-based design environment [Xil02]. A System Generator design can be compiled into a netlist, which can then be synthesized onto an FPGA. The latest Vivado design suite from Xilinx integrates System Generator, AutoESL, and RTL synthesis into a single ESL design environment.

Many of the discussed high-level synthesis and system-level design tools do not address the specification gap, as they require the designer to provide a parallel specification. On the other hand, Daedalus employs the PNGEN tool flow to bridge the specification gap as it can automatically find a parallel specification. As the designer does not have to provide a parallel specification, the design process is accelerated. A key challenge of automated parallelization is detecting the statements that are independent of each other, such that they can execute in parallel. PNGEN employs exact data dependence analysis to precisely find the dependence relations between statements [Fea91]. Obtaining exact data dependence information is complicated and is not always possible for arbitrary code. Most HLS and ESL tools that start from a sequential specification therefore rely on approximate data dependence anal-

ysis techniques such as Banerjee's test [Ban88]. But as a result of the approximate nature, tools may need to conservatively assume a data dependence exists between statements, possibly preventing any parallel execution. Such false data dependences can be circumvented using tool-specific compiler pragmas that need to be inserted by the designer. Daedalus requires that the application is specified as a SANLP (cf. Section 2.3) for which exact data dependence analysis is always feasible. This eliminates the need for conservative data dependence assumptions, freeing the designer from having to manually analyze data dependences.

1.4 Contributions and Outline

In Section 1.2, we identified three central problems in an iterative system-level design flow. We solve these three problems in this dissertation in the context of the Daedalus methodology [NSD08b], which we discuss in more detail in **Chapter 2**. The Daedalus methodology is the result of many dissertations in the LERC group [Rij02, Ste04, Tur07, ZI08, Nik09, Mei10, Nad12, Bal13]. We want to build further upon the contributions made in these dissertations. The research conducted in this thesis has led to the following four contributions:

Contribution I [HK09, NHS+11, HK12]: As a solution to the synthesis problem, we add extensions to the Daedalus methodology. These extensions enable us to obtain FPGA implementations from C programs which were already accepted by Daedalus, but for which an FPGA implementation was not yet feasible. With these extensions, we are now able to obtain complete FPGA implementations for industrially relevant applications, like the sphere decoder application discussed in Chapter 6. We have shown that we can characterize and integrate functional kernels (IP cores) from a broad set of conventional HLS tools like the industrial tools Vivado HLS and Synphony C, and the academic tool DWARV. Our extensions to the Daedalus methodology provide an enabling step to realize the complete conventional "forward" system-level synthesis flow for FPGAs in the flow shown in Figure 1.3. The position of these extensions is indicated by the ❶ in Figure 1.3 and are discussed in **Chapter 3**.

Contribution II [HHK10]: As a solution to the performance estimation problem, we investigate four different performance estimation techniques, that differ in accuracy and assessment effort. We want to emphasize on two techniques in particular. We show a novel analytical approach to estimate the performance of cyclic PPNs. The analytical approach is based on the well-known *Maximum Cycle Mean* (MCM) theory from the HSDF model of computation, but avoids exponential complexity explosion in the PPN-to-HSDF conversion. Although providing a theoretical basis, the practical use is limited due to unknown accuracy of the result. In that respect, an-

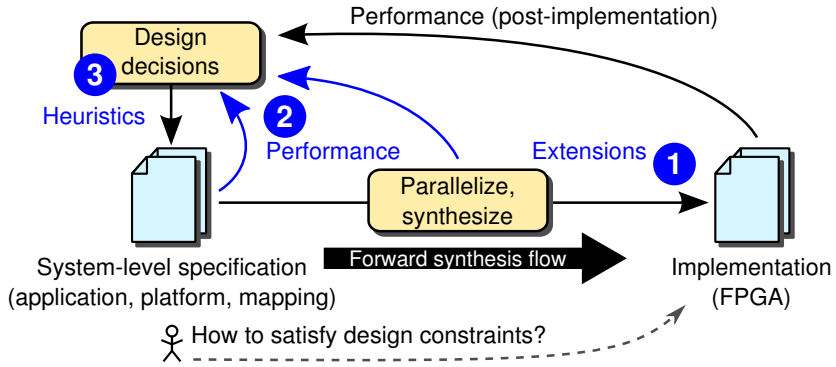

Figure 1.3: Contributions positioned in the iterative design flow.

other contribution of this thesis is *cprof*, which is a novel profiling based approach that completely bypasses the forward synthesis flow. The approach is robust as it relies only on an ordinary C++ compiler to obtain accurate performance estimates of PPNs. Moreover, the approach allows for early estimation of the effects of transformations. The approach provides the designer with an upper bound on the degree of parallelism in an application specification. This allows a designer to assess at a very early stage in the design flow whether he can meet his constraints. The position of the alternative performance estimation techniques in the overall design flow is indicated by the ❷ in Figure 1.3 and is discussed in **Chapter 4**.

Contribution III [HK12]: As a solution to the optimization problem, we provide heuristics to optimize a design by leveraging the insight gained from the performance estimation techniques discussed in Chapter 4. The heuristics provide a concrete set of criteria that guide the designer in selecting standard transformations such as splitting, merging, stream multiplexing, and scheduling. This position of the heuristics is indicated by the ❸ in Figure 1.3 and is discussed in **Chapter 5**.

Contribution IV [HNVK11, NNH+10, NNH+11]: We have shown that we can apply the extended forward system-level synthesis flow depicted in Figure 1.3 on an industrially relevant application. This case study also shows that PPNs are a feasible alternative to conventional CDFG-based C-to-RTL flows. Using the heuristics from Chapter 5, in particular merging, we were able to transform the design to obtain a new pareto design point that was not achievable with a state-of-the-art industrial HLS tool. The use of the profiling-based cprof performance estimation technique presented in Chapter 4 was essential to gain insight in the application performance and the optimization opportunities. The case study is discussed in **Chapter 6**.

We summarize our work and conclude in **Chapter 7**.