INTEGRATING ALGORITHM-LEVEL DESIGN AND
SYSTEM-LEVEL DESIGN THROUGH SPECIFICATION
SYNTHESIS

A Dissertation Presented
by
Jiaxing Zhang
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in
Computer Engineering
Northeastern University
Boston, Massachusetts
August 2014
To my family.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>viii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>x</td>
</tr>
<tr>
<td>Abstract</td>
<td>xi</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Specification Gap</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Flexibility and Overhead Trade-off</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Problem Definition and Goals</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Overview</td>
<td>8</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 EDA Designs</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Algorithm Level Design</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 MATLAB/Simulink Modeling Environments</td>
<td>14</td>
</tr>
<tr>
<td>2.3 System-Level Design</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1 Design Abstractions</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2 System Level Design Languages</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Related Work</td>
<td>20</td>
</tr>
<tr>
<td>2.4.1 Bridging Design Gaps</td>
<td>20</td>
</tr>
<tr>
<td>2.4.2 Flexibility Overhead: Granularity Challenges</td>
<td>24</td>
</tr>
<tr>
<td>2.4.3 Flexibility Overhead: SW Synthesis</td>
<td>24</td>
</tr>
<tr>
<td>2.5 Summary</td>
<td>25</td>
</tr>
<tr>
<td><strong>3 Specification Synthesis</strong></td>
<td>27</td>
</tr>
<tr>
<td>3.1 Specification Synthesis Foundations</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1 Structural Semantics</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2 Execution Semantics</td>
<td>31</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Specification Gap ................................................................. 3
1.2 Flexibility vs. Efficiency ...................................................... 4
1.3 New Co-design Flow Overview ............................................... 7

2.1 Abstract Design Flow ........................................................... 11
2.2 Algorithm Level Design Environments [125] ............................. 13
2.3 Simulink Model Based Design (courtesy [83]) ......................... 14
2.4 Simulink Development Flow .................................................. 15
2.5 SLD: Y-Chart (Courtesy [47]) ............................................. 16
2.6 System Level Design Environments ........................................ 17
2.7 System Level Design Abstraction [54] .................................... 18

3.1 Simulink and SLDL Structural Semantics ............................... 29
3.2 Hierarchical Composition Example ....................................... 30
3.3 The Algo2Spec Specification Synthesis Flow ......................... 32
3.4 Atomic Unit in Simulink and SLDL ........................................ 34
3.5 Leaf Synthesis Example ....................................................... 35
3.6 Hierarchy Synthesis Example: Diagrams ............................... 38
3.7 Hierarchy Synthesis Example: the SLDL Source Code ............. 40
3.8 Algorithm-Architecture Co-Design Flow ................................ 42
3.9 MJPEG Encoder Structural Hierarchy .................................... 45
3.10 Early Estimation and Exploration of MJPEG Encoder ............ 48
3.11 DSE of MJPEG Encoder ...................................................... 49

4.1 Flexibility and Efficiency Trade-off ....................................... 55
4.2 MJPEG Encoder Algorithm Block Characteristics ................... 58
4.3 Demand Matrix ................................................................. 59
4.4 DMGT Tuning and Synthesis Flow ........................................ 60
4.5 Vertical Merging based on Computation ................................ 62
4.6 Example Model Tree Input ................................................... 66
4.7 DMGT: Horizontal Clustering based on Communication .......... 67
4.8 DMGT: Synthesis-Phase Example ......................................... 71
4.9 CGGT: MJPEG Block Reduction ........................................... 74
4.10 CGGT: MJPEG Granularity vs. AFP ..................................... 75
List of Tables

3.1 Signal Topology Categories ......................................................... 39
3.2 Design Characteristic and Productivity Gain ................................ 52
4.1 Leaf BlockSize and Tuning Threshold ............................................. 63
4.2 Example Model Demands ............................................................. 68
4.3 Benchmark Application Characteristics ....................................... 73
4.4 CGGT: Tuning Effect on MJPEG Encoder ....................................... 75
4.5 DMGT Vertical Clustering: Tuning Effect on MJPEG Encoder .......... 78
4.6 DMGT-Synthesis Results ............................................................. 79
5.1 DDO-Reduction for JPEG encoder ............................................... 99
List of Acronyms

ALD(E) Algorithm-Level Design (Environment). A design methodology (environment) for describing numerical, abstract, algorithmic models in high abstraction with simulation and verification facility.

ASIC Application-Specific Integrated Circuit. An integrated circuit (IC) customized for a particular use.

ASIP Application-Specific Instruction-set Processor. A processor with the instruction set tailored for specific applications.

Behavior An encapsulating entity, which describes computation and functionality in the form of an algorithm.

DSE Design Space Exploration. An activity of finding design alternative (candidates) prior to implementation to meet or balance one or multiple design objectives.

EDA Electronic Design Automation. A design flow with a category of tools for designing, specifying, verifying and implementing electronic systems.

ESL Electronic System Level. A design methodology focusing on design and verification with clear abstraction separations.

MoC Model of Computation. A set of rules that define the interaction and composition of components.

MPSoC Multi-Processor System-on-Chip. A highly integrated device implementing a complete computer system with multiple processors on a single chip.

PE Processing Elements. A system component that performs computation or data processing.

RTL Register Transfer Level. Description of hardware at the level of digital data paths, the data transfer and its storage.

SCE SoC Environment. Tool set for automated, computer-aided design of SoC and computer systems in general.

SLD(E) System-Level Design (Environment). A design methodology (environment) for describing complete, heterogeneous, mixed hardware/software computer systems using SLDLs.
SLDL  System-Level Design Language. A language for describing complete, heterogeneous, mixed hardware/software computer systems at high levels of abstraction.

SoC  System-On-Chip. A complete computer system implemented on a single chip or die.

TLM  Transaction Level Model. A model of a system in which communication is abstracted into channels and described as transactions at a level above pins and wires.

VHDL  VHSIC Hardware Description Language. An hardware description language commonly used for hardware design at RTL and logic levels.
Acknowledgments

This work would not have been possible without many people’s help. I would like to take this opportunity to express gratitude to all those who made this work possible.

First and foremost, I would like to thank my advisor, Professor Gunar Schirner, for his insightful guidance, generous support, unwavering patience, and consistent encouragement that made this work a success. I would also like to thank him for sparking my research thoughts and offering countless nights reviewing and improving my papers. His leadership, vision, and care helped me grow and mature in so many aspects. I am also very thankful to him for ensuring that my research was funded all along.

I would like to thank Professor Miriam Leeser and Professor David Kaeli for serving on my committee. I am very grateful to both of them for providing me valuable suggestions to improve this dissertation. I am also grateful to the members of the Embedded Systems Laboratory group for their constructive critiques, challenging discussions, and brilliant ideas during my research.

This work was supported in part by the VOTERS project (NIST grants) and NSF grants. I would like to thank the VOTERS staff, Professor Ming Wang and Professor Ralf Birken, as well as NSF and NIST, for recognizing the potential of this research and funding my doctoral studies.
Abstract

INTEGRATING ALGORITHM-LEVEL DESIGN AND SYSTEM-LEVEL DESIGN THROUGH SPECIFICATION SYNTHESIS

by
Jiaxing Zhang
Doctor of Philosophy in Computer Engineering
Northeastern University, August 2014
Prof. Gunar Schirner, Adviser

Modern Multiprocessor System-On-Chip (MPSoC) designs face challenges in tremendous design complexity imposed by the wide range of functional requirements and versatile architectural platforms. The Electronic Design Automation (EDA) research has been striving to accelerate the creation, deployment and validation cycles from initial design modeling to final system delivery. However, establishing a universal, automatic and rapid design flow from concept modeling to heterogeneous implementations with synthesis, optimization and exploration capability remains an open challenge due to the overwhelming complexity, vastly different abstractions, and market adaptability and usability.

This thesis introduces a specification synthesis approach that joins two design methodologies, Algorithm-Level Design (ALD) and System-Level Design (SLD), to establish a new Algorithm-Architecture Co-design flow. We designed and implemented an algorithm-to-specification synthesizer: Algo2Spec, which out of an algorithm model captured in ALD, such as Simulink, synthesizes into an SLD languages (SLDL) specification (e.g. SpecC/SystemC) enabling SLD exploration. Expanding the rich sets of SLD facilities into higher abstraction levels in ALD forms a new joint co-design methodology. The new flow seamlessly spans from the Simulink environment down to heterogeneous implementations crossing multiple abstractions. Our tools empower designers to construct, simulate, validate, explore, and deploy models in rapid feedback cycles. Our results illustrated the opportunities and benefits of our approach on a set of real-world applications and showed a significantly shortened design time.

In addition, we explore new optimization opportunities emerged from the specification synthesis with respect to Design Space Exploration (DSE) and software synthesis. We have identified several DSE challenges, such as selecting a suitable model granularity to balance the model
mapping flexibility, specification quality as well as synthesis constraints (i.e. computation/communication efficiency and scheduling). We design two heuristics to address these challenges to reduce the DSE complexity while increase the overall performance efficiency. Moreover, once model-to-platform mapping is known, Algo2Spec allows to generate specifications with maximized computation efficiency and reduced communication overhead.

As both ALD and SLD incorporate the principles of the component-based design, a certain degree of modeling flexibility, gained by using abstracted component compositions, appears in both domains. However, such beneficial flexibility becomes overhead once the model, passing high-level modeling and exploration periods, enters the back-end synthesis phase. In software synthesis, such flexibility comes at a cost of run-time call site resolution. We investigated into the impeded performance in current embedded software synthesis from SLDL specifications. Our results show that by eliminating unnecessary flexibility overhead such as dynamic dispatch, the SLDL-to-C compiler achieved better performance on embedded processors, improved readability, and debuggability compare to current solutions.

The outcome of the work greatly simplifies the Algorithm-Architecture Co-design with new tools, methodologies, and optimizations. The thesis has demonstrated how the new design flow and methods can deeply enhance current algorithm design solutions to leverage the vastly available computing power in today’s heterogeneous architectures.
Chapter 1

Introduction

The complexity of modern embedded system designs is dramatically increasing on both functionality and architecture. Markets demand the integration of more features that simultaneously become more complex; meanwhile design flexibility is required to react to varying customer needs as well as changing standards. To realize the challenging functional demands, and simultaneously meet stringent non-functional and performance constraints, designers often leverage the power and flexibility of multi-core heterogeneous systems such as Multi-processor System-On-Chip (MPSoC). Time-to-market is of critical importance in MPSoC designs and applications, which forms the driving force for design automation research and studies. Efficient design methodologies are needed in both domains of the algorithm and platform development to keep up with the increasing functional and architectural complexity.

Current methodologies broadly cope with design complexities from two different angles, namely functional (algorithm-level) or architectural (system-level) perspective. Algorithm-Level Design (ALD) approaches help specifying functional and behavioral aspects of the design at a high abstraction level without considering platform implementation details. Here, by algorithm selection and composition, the focus is to achieve the desired functional results, such as numerical validity or image quality of a multimedia solution, or accuracy of object detection in a security system. On the other hand, System-Level Design (SLD) methodologies help designers to identify platforms suitable for executing the desired application to meet non-functional constraints such as power or cost efficiency. Through Design Space Exploration (DSE), SLD techniques guide designers in selecting Processing Elements (PEs), communication architecture, mapping and scheduling of behaviors over heterogeneous architectures.

Various languages and ALD environments (ALDE) are available to aid the design and
CHAPTER 1. INTRODUCTION

implementation of algorithm models in fast validation cycles, such as TargetLink [33], LabView [69] and Simulink [84]. In this thesis, Simulink is used as the ALDE, as it shares common characteristics with other model-based design tools and is supported by other algorithm tools. Additionally, its features such as algorithm databases, comprehensive toolboxes, and analysis capabilities draw a wide audience in Digital Signal Processing (DSP) and multimedia communities in the Electronic Design Automation (EDA) area.

Once an authored algorithm is intended for deployment, system-level explorations are an ideal vehicle to analyze the model on heterogeneous platforms to meet constraints such as timing, power, budget, energy or cost. SLD languages (SLDL), such as SystemC [94] and SpecC [49], have been established to capture complex systems at the system-level in order to reduce the complexity of the MPSoC exploration. Built upon SLDLs, SLD environments (SLDE) automate the system-level design and exploration flow. SLDEs, such as PeaCE [60], DAEDALUS [113], OSSS [59], and SoC Environment (SCE) [36], are developed to aid system-level modeling, analysis, verification, validation, and synthesis through iterative refinement using SLDLs. SLD methodologies reduce development cost and increase productivity for large and complex MPSoC designs. In this work, SCE is selected as the SLDE [1].

Nevertheless, currently ALDEs and SLDEs primarily focus on challenges in their specific domains individually. Due to the lack of automated tool-chains, manual converting the ALD models to be suitable for SLD specification is often required. We observed a clear Specification Gap between ALDEs and SLDEs illustrated in Figure 1.1.

1.1 Specification Gap

Presently, designers first model, validate and analyze the design in an ALDE, such as Simulink, and then manually convert them to an SLDL specification where further structure and hierarchy analysis and mapping can be performed. The process of manual conversion across different abstraction layers is error-prone, time-consuming and extremely costly. A similar gap discussed in [21] also emphasizes on the loss of productivity due to abstraction disconnection, yet in a separate context. Chandraiah et al. observed [22] that it takes 12-14 weeks to manually re-code an MP3 reference model from plain C to a well-defined SLDL specification. Given the increased functional complexity, it is likely that the manual conversion time for current specifications would be dramat-

---

1We chose SCE since the system-level exploration environment is readily available. The concepts are applicable to other SLDE, such as PeaCE.
ically higher. More severely, changes in the design require multiple iterative manual conversions causing a lengthy modification of the model to match the desired specification \[53\]. Therefore, the manual effort causes a loss of productivity as it stalls in the overall design cycle. Moreover, the high model complexity often makes it impractical for manual conversion in real-world examples. This limitation inevitably reduces the opportunities of early HW/SW co-design and parallel explorations, which are essential for MPSoC design success. Hence, there is a need for new automation methods and tools to (1) avoid manual transformation from ALD to SLD, (2) improve design quality, (3) reduce development overhead, and subsequently (4) accelerate the overall design cycle.

1.2 Flexibility and Overhead Trade-off

Once there is a tool-chain to bridge the disconnected abstractions between ALD and SLD, it offers designers more control over the desirable characteristics of the specifications as a Simulink model being converted into a system-level specification. One of these controllable decisions is **granularity**, in this context defined as *how many blocks* from the algorithm models can be retained.
in the SLDL specifications. The decision of granularity influences two aspects to the DSE phase in the SLD.

- Granularity impacts **Flexibility**. A finer granularity option produces more blocks, whereas a coarser granularity choice reduces the block amount. More blocks in a specification increase the mapping alternative potential, and this is known as mapping flexibility. The degree of necessary flexibility is tightly associated with underlying platform as more independent elements require higher flexibility for mapping.

- Granularity impacts **Efficiency**. A finer granularity hinders optimization as block fusion cannot take place. Conversely, a coarser specification (i.e. fewer blocks) out of Simulink models has the potential to increase execution efficiency leveraging the power of inter-block optimizations (see Chapter 4).

Nevertheless, synthesizing into a single block (zero flexibility) is DSE design-prohibitive as it removes any Hardware/Software Codesign opportunities although it may achieve the most possible block fusion potentials. Figure 1.2 conceptually illustrates the trade-off between *Flexibility* and *Efficiency*.

The usefulness of flexibility, however, is correlated to the number of blocks regarding the platform choices. If more independent elements were available in the platform, then more flexibility would be required. For instance, in Application-Specific Integrated Circuit (ASIC) designs, a much finer grained granularity offers more blocks to fine tune the data path and block scheduling. Conversely, considering mapping to multiple processors would require a lower degree of independent
CHAPTER 1. INTRODUCTION

elements (e.g. the platform contains tens of Processing Elements (PEs)). In this case, the extra flexibility therefore becomes overhead that potentially paralyzes DSE as the mapping options increase in exponential order to the number of blocks. In the context of this thesis, we address how to prune DSE space to achieve a suitable specification design for HW/SW codesign.

1.3 Problem Definition and Goals

This thesis identified the following challenges along with goals in the context of accelerating total design efficiency in today’s EDA industry.

   **Goal:** A specification synthesis approach targets Simulink model to SLDL specification transformation.

2. Design Space Exploration (DSE) is overburdened by the sheer number of blocks in specification input.
   **Goal:** Reduce DSE complexity through block reduction while increase specification efficient in specification synthesis.

3. Embedded code generated by software synthesis produces a *Flexibility Overhead* due to retaining structural details of the system model in the synthesized code.
   **Goal:** Eliminate the DDO to improve performance.

1.4 Contributions

By addressing the above challenges, this dissertation delivered the following contributions:

**Contribution 1** *Closed the specification gap through algorithm-to-specification synthesizer: Algo2Spec.*

*Algo2Spec* enables Simulink models to be synthesized to SpecC specifications that are functional valid, structurally composable, explorable and synthesizable in SCE. *Algo2Spec* eliminates the necessity of manual re-authoring to improve the quality of model transformation and meanwhile exploits optimization options to further increase the overall system performance.

**Contribution 2** *Identified synthesis options in specification synthesis*
As Algo2Spec creates a new synthesis domain, multiple synthesis options emerged. Granularity tuning and parallel scheduling are two major options that are identified by this work. However, the introduced concepts and methods are applicable to other high-level transformation tools bridging similar gaps.

**Contribution 3** *Identified and analyzed Granularity vs. Efficiency trade-off*

We quantified the trade-off between the model granularity and achieved fusion potential (AFP). AFP quantifies how much of the maximally available fusion potential (i.e. when synthesizing to a single block only) is achieved. Over 12% AFP has been observed in real-life benchmarks and we anticipate even higher improvement as code generation tool advances. Moreover, we analyzed the impact of granularity on the DSE complexity.

**Contribution 4** *Introduced Computation-Guided Granularity Tuning (CGGT) approach to explore the trade-off based on computation*

Introduced a Computation-Guided Granularity Tuning (CGGT) algorithm to automatically identify a suitable set of blocks (model granularity) with sufficient amount of computation per block, yet not overkilling, in the synthesized specification directed by a tuning knob. Over 80% blocks can be reduced with 80% AFP.

**Contribution 5** *Introduced Demand-/Mapping-aware Granularity Tuning (DMGT) approach to explore trade-off based on computation, communication, and mapping*

Introduced a Demand/Mapping-aware Granularity Tuning approach to horizontally cluster blocks to reduce the model traffic while allowing for more block reduction and higher AFP gain. Assuming that platform allocations and mapping are already known, a mapping-aware specification synthesis can be produced to eliminate all overhead. This further extends to 98% AFP with 0% remaining mapping flexibility.

**Contribution 6** *Eliminated flexibility overhead for synthesized embedded code*

After the DSE refinement stage, the specification is implemented by the software synthesis backend. Here, a flexibility overhead exists as well, yet, in the form of Dynamic Dispatch Overhead (DDO). We proposed a DDO reduction approach to eliminating any determinable costly run-time resolution overhead to achieve over 16% target execution speedup.
As an overall result, Algo2Spec establishes a joint Algorithm-Architecture Co-design flow (Figure 1.3) exposing system-level design features to the algorithm perspective. The new flow seamlessly extends an existing top-down work-flow [46] (SLD) to reach higher abstractions (i.e. in the ALD). The algorithm designers start with Behavioral and Functional requirements and refine the algorithms through decisions utilizing predefined algorithm blocks. Our tools enable the connection between algorithms to specification models at system-level. Designers, therefore, can perform early stage exploration with rapid feedback from heterogeneous platform explorations with System Level Refinements using Architecture and Components. TLM specifications are generated to evaluate the timing behaviors. Once the exploration is satisfactory, Backend Synthesis realizes these system decisions on hardware and software. The new flow not only significantly reduces the total design effort (both in model synthesis and DSE), but also increases performance through platform-specific algorithm improvements.
CHAPTER 1. INTRODUCTION

1.5 Overview

Figure 1.3 illustrates how the dissertation fits into the proposed Algorithm-Architecture Codesign flow with individual chapters devoted to one or several contributions listed above. The remainder of the dissertation is organized as follows: Chapter 2 provides the background in EDA and discusses relevant research effort.

Chapter 3 discusses the specification synthesis and the design and implementation of Algo2Spec. It covers the specification synthesis foundation, examines the internal design of Algo2Spec in detail and overviews the new design flow. Results of utilizing the new design flow built on Algo2Spec are also provided.

Chapter 4 examines the emerged challenges, opportunities and trade-offs during the specification synthesis. Especially, it investigates into the granularity identification. Two heuristic algorithms have been presented to select suitable synthesized specification granularity with respect to overall synthesis quality on performance efficiency and traffic overhead. As both ALD and SLD incorporate the principles of the component-based design, a certain degree of modeling flexibility, gained by using abstracted component compositions, appears in both domains.

Chapter 5 quantifies such overhead in software synthesis process. Eliminating unnecessary flexibility such as dynamic dispatch, we show that our approach achieved better performance while improving readability and debuggability of the synthesized code.

Lastly, Chapter concludes this dissertation and proposes future research paths.
Chapter 2

Background

This chapter reviews the relevant work and advances in Electronic Design Automation (EDA), Algorithm-Level Design (ALD) and System-Level Design (SLD). Afterwards, it examines current state-of-the-art research related to this thesis.

2.1 EDA Designs

For decades, embedded computer systems have been growing in numbers as a dominating computing force in many aspects of our lives. From the control-centric devices used in refrigerators to safety-critical real-time systems in automobiles, over 99% microprocessor produced today are employed in embedded systems applications [115]. Benefited from the ever-increasing integrated transistor volumes and versatile on-board components, embedded computer systems have rapidly evolved into the domain of System-On-Chip (SoC). SoC contains a full system with complete functionality and peripherals on a single chip, such as input and output, memory hierarchy and auxiliary hardware [82]. Unlike general computing systems, such as CPUs, that aim to solve the computational problems in a highly flexible domain, SoC focuses on addressing problems that are predefined, deterministic, and analyzable, which are the common characteristics of embedded systems application requirements. Besides sharing the common interests with general computing on accelerating processing power, SoC faces the challenges that they need to be energy/area/cost efficient [38]. To overcome these obstacles, heterogeneous systems are a promising solution, which brings the rise of Multiprocessor SoC (MPSoC) and in particular Heterogeneous MPSoCs [79]. From the low power chips inside our handheld devices to the acceleration units in supercomputing clusters, MP-

\[1\) In this thesis, we use MPSoCs to denote Heterogeneous MPSoCs.
CHAPTER 2. BACKGROUND

SoCs have become a stronghold for the new era of the future embedded solutions. Being able to offer both extraordinary processing strength and balanced power efficiency, MPSoC technologies have exhibited a high degree of heterogeneity in the architectural perspective [56, 116].

To harness the heterogeneity and leverage the pervasive computing power in MPSoC, System-Level Design in Electronic System Level (ESL) designs and verification have emerged aiming to accelerate the design process started from concepts down to silicon. Meanwhile, the prosperity of MPSoC has also largely driven the increase of functional input at the top level of the MPSoC design.

Facing the exponential growth of the amount of transistors and ever-advancing functional features in modern MPSoC systems, today’s Electronic Design Automation (EDA) faces tremendous complexity in specifying, modeling, designing, verification and implementation of MPSoC. However, the fast advances of MPSoC technologies, further spurred by functional demands, have unequally outgrown the development of corresponding design methodologies. Current methodologies broadly cope with the design complexities from two different angles, namely the functional (algorithm-level) perspective or the architectural (system-level) perspective.

Due to the high complexity of the functional requirements, Algorithm-Level Design is often involved in modeling the algorithmic attributes aspect, such as numerical validity. Figure 2.1 illustrates the abstract design flows in today’s ESL design. The entire flow can be conceptually separated into the broad domains of Marketing Input and Engineering Solutions. The Market Input often specifies the functional requirements bound by behavioral constraints. The functional requirements define a certain input producing a certain output over time. Whereas behavioral constraints are used to apply functional/non-functional boundary of the behavior. Typical examples include timing (deadline), power consumption, or system throughput.

We have seen constantly growing features from the marketing input (i.e. functional requirements and behavior constraints) that become more complex and richer in functionality. As predicted by ITRS [66], the number of features demanded in current electronic applications doubles every two years. In addition, more complex kernels and advanced algorithms, such as real-time background subtraction [96] or 4G LTE wireless baseband [26], aim to address the challenges that were previously computationally unsolvable. They have appeared as embedded implementation in favor of the ubiquitous computing flexibility and cost effectiveness [37]. Moreover, market input demands the integration of top-level features with design flexibility to react to varying customer needs (e.g. cost, timing), changing standards, and shortened time-to-market.

To realize the challenging functional demands, and simultaneously meet stringent non-
functional constraints, system abstractions must be applied. One design trend is to start with algorithm modeling to obtain a high-level behavioral description using an Algorithm-Level Design (ALD). Shown in Figure 2.1, algorithms can be modeled strictly complying with the functional needs. Constraints can be applied through the evaluation and verification step as the embedded target are concerned. The overall solution provides a powerful automatic toolchain offering conceptual constructs to final implementation. However, the weakness is that, in ALD, it is infeasible to directly target MPSoC solutions using such methodology due to the extra degree of modeling complexity such as structures and Design Space Exploration (DSE).

To address the structural and DSE complexity in the ESL design flow, the System-Level Design (SLD) methodologies focuses on the architectural perspective operating on a given behavioral specification captured by functional requirements and behavioral constraints at the same
CHAPTER 2. BACKGROUND

time (see Figure 2.1). The strength of this flow is the ability to perform architecture refinements experimenting on multiple implementation alternatives (re-targeting) at various abstraction levels with different estimation accuracy and simulation speed. In particular, the inherent heterogeneity in MPSoC is ideal for Hardware/Software (HW/SW) co-design in the SLD flow.

Although aiming at different abstractions, both design methodologies have incorporated a number of crucial design principles in common. In order for any product to succeed, commonly practiced design philosophies involve three key aspects [118, 19, 48, 62, 71]: design abstractions, rigorous top-down design flow and component-based module reuse.

A certain level of abstractions must be orthogonally raised in order to separate concerns [49, 104]. In ALD, such separation can be underlying timing characteristics (timed/untimed model), whereas, in SLD, they can be modeling details (TLM vs. RTL). To connect multiple abstractions, a top-down modeling flow with rigorous design constraints can greatly simplify modeling-to-implementation process with reduced complexity as such flow can approximate the overall details incrementally through guaranteed and verifiable refinements. Component-based design and module reuse have been shown beneficial in boosting productivity [19] in various application and abstractions, in favor of functionality encapsulation and IP protections.

Although ALD and SLD handle abstractions in their own domains, it remains an open question on how to connect these two domains. Bridging the extra level of abstraction between algorithm models and system specifications requires filling the semantic gap in the middle, which is currently done manually [21, 125]. In this next section, we will first briefly look into the ALD to explain its relevant features for algorithmic modeling.

2.2 Algorithm Level Design

An ALD environment is typically where the concepts of the overall system is formed. Started with initial requirements, designers can model such requirements in highly abstracted descriptions. For instance, a video encoder application requires high-level specification of the input and output as its functionality constraints. A certain input would produce a predictable output with certain constraints such as timing requirements or throughput threshold. Figure 2.2 shows an exemplary ALD environment.

ALD contains modeling tools to aid designers construct algorithm models that can be simulated and validated. With the results obtained in the environment, designers can easily conduct analysis of the model characteristics with a fast and often real-time feedback about the model
CHAPTER 2. BACKGROUND

The modeling tools in ALD often rely on mathematical representations, where the relation between components can be formally defined and verified using mathematical expressions and attributes. Various languages and ALD environments (ALDE) are available to aid the design and implementation of algorithm models in fast validation cycles, such as Modelica [44], TargetLink [33], LabView [69] and Simulink [84]. In this paper, Simulink is used as the ALDE, as it shares common characteristics with other model-based design tools and translatable from other modeling environments [5]. Additionally, its features such as algorithm databases, comprehensive toolboxes, and analysis capabilities draw a wide audience in Digital Signal Processing (DSP) and multimedia communities in the Electronic Design Automation (EDA) area. As the algorithm modeling serves as a pivotal point for directing the overall system quality, here we briefly review a few high-level algorithm modeling tools.

The typical steps involved in ALD starts with problem definition, algorithm specification, model construction, and system validation and verification. These steps are tightly enclosed in a design loop empowered by a set of tools, such as simulation, validation and model checkers.

Being able to understand the model performance and to provide functional and behavioral verifications at early design stage is instrumental and crucial to the overall product. Wrong or improper design inputs at the top level have a disastrous impact on the final product quality and exponentially increase the development costs. Thus, in order to improve the cost effectiveness, design quality, as well as time-to-market, Model-Based Design (MBD) is a popular methodology to enforce those goals through a central model presentation [33] in ALD. With the proliferation of MBD tools, Simulink is a popular tool in today’s algorithm development suite both in industrial support as well as academic communities. In the research of this thesis, we focus on applying Simulink as the front-end algorithmic input and thus, the ALDE instance for our work.
CHAPTER 2. BACKGROUND

2.2.1 MATLAB/Simulink Modeling Environments

Figure 2.3: Simulink Model Based Design (courtesy [83])

Simulink [84] is an ALDE using block diagrams for multi-domain simulation and Model-Based Design. It offers simulation, execution, continuous test and verification. Through a graphical editor and comprehensive predefined algorithmic block libraries, Simulink significantly reduces the cost for system modeling and eases the effort for new algorithm development.

It has been adopted as the primary modeling tool for control engineers in automotive and avionic industries for years [111]. Moreover, Simulink, together with MBD principles and modularized algorithmic blocks, are particularly suitable in multimedia and computer vision applications [91][34]. In general, Algorithm-Level Design (ALD), such as Simulink, approaches facilitate specifying functional and behavioral aspects of the model at a high abstraction level without concerning platform implementation details. Here, by algorithm selection and composition, the focus is to achieve the desired functional results, such as numerical validity or image quality of multimedia solution, or accuracy of object detection in a security system.

A case of model based design principle of Simulink is illustrated in Figure 2.3. Besides pure algorithm aspects, such as simulation and logic verification, the model-centric environment also offers implementation with automatic code generation. The code generation facility, such as Simulink Coder [85], extends the pure algorithm design flow to target implementation (see Figure 2.4).

Simulink allows the process from requirements to model and from model to implementation in C code. The generated C code can be deployed onto specific homogeneous targets supported by Simulink. However, it cannot be used to produce MPSoC solutions as in essence, Simulink is
CHAPTER 2. BACKGROUND

![Simulink Development Flow](image)

Figure 2.4: Simulink Development Flow

not an architectural exploration environment and thereby cannot provide system-level functionality.

On the contrary, System-Level Design (SLD) methodologies help designers to identify platforms suitable for executing the desired application to meet performance, power, area and other non-functional constraints. Through Design Space Exploration (DSE), SLD techniques guide designers in selecting Processing Elements (PEs), communication architecture, mapping and scheduling of behaviors over heterogeneous architectures.

2.3 System-Level Design

Conceptually, System-Level Design (SLD) can be viewed using the Gajski’s Y-Chart by navigating multiple abstraction levels in three design areas: namely, functional behavior, architectural structure, and physical layout, on different axes shown in Figure 2.5.

Connecting to the specification in Figure 2.1, the functional behavior domain again specifies the functionality of the system. On the other axis, the architectural structure domain identifies the architecture (hardware) platform including processing elements (PEs), communication elements (CEs), and memory elements. The physical domain defines the implementation details of the structure, such as the size and position of the components, the printed circuit board, and the port connection between components.

The dotted concentric circles represent the different levels of abstractions, ranging from the transistor level to the system level. Design flows can be illustrated as paths in the Y-Chart moving from one axis to another. The SLD can be viewed as the design choices move from the behavior domain to the structure domain shown in the figure. As an example in SLD, given behavioral description of the design, designer can map the behaviors to certain architectures (structural) at the system-level, and this process can be iteratively applied to lower abstractions (center to the chart) through refinements. Refinement is the process of approximating the specification to more detailed specification. For instance, the architecture refinement for a specification is to add the architecture layer to the existing behaviors such that structure characteristics can be reflected. CPU parameters
such as frequency, cache sizes, memory hierarchy can be previously modeled in a centralized library to enhance the model reusability and modularity. Lastly, the design flow can traverse to the physical domain to allow the abstract specification to be finally implemented. This step is commonly accomplished through Hardware Synthesis tools or software synthesis approach.

The design philosophies enclosed in the Y-Chart can be realized in System-Level Design environments. Figure 2.6 abstractly illustrates such an environment. The top-down design methodologies are also incorporated into such environment, which naturally fits the step-wise refinements. Operating on specifications as input, SLD environments include multiple refinement stages to obtain more accurate results through architecture mapping, communication selection, and task scheduling. Typically, Transaction Level Modeling (TLM) is a standard level of model representation with satisfactory modeling accuracy with fast simulation speed. Similar to ALD environments, multiple loops and feedback can be easily employed to aid system-level designers to analyze the specification performance through SLD simulation engines and validation tools. Lastly, once the refined TLM specification meets the designers’ requirements, it can be synthesized to implementation of both HW and SW, fitting the MPSoC capability.
CHAPTER 2. BACKGROUND

2.3.1 Design Abstractions

The key ingredients of SLD methodologies are design abstractions. It permits the designers to handle the complexity of the entire system at different levels without overwhelming by unnecessary details. This separation of concerns is a critical solution for dealing with a high degree of design complexity and, therefore, exploits the hierarchy [29].

Figure 2.7 shows the benefits gained by raising abstraction levels in SLD. Higher abstractions result in few numbers of components to be exploited at a given level. The overall system delivery comes at a cost of tens of millions of transistors, which simply paralyzes any human labor involved design effort. Moving up a level higher, gate-level abstracts the characteristics of transistors through logic gates consists of a set of transistors. Similarly, register-transfer level (RTL) further reduces the number of elements appeared in the system using even higher abstractions. The abstraction can be applied iteratively to extra the commonality and essence of components to achieve processor-level or system-level, respectively.

Although comes at a cost of accuracy, the high-level abstraction principle is a powerful methodology to handle today’s design addressing overwhelming flexibility. This allows designers to maintain a high-level and system-oriented overview in an effective way. The advantages of deploying the abstraction principles help alleviate the designer’s burden. By focusing on the crucial features needed to address at certain abstraction level, and then, with the aid of automatic toolchains, models can be automatically generated or improved to contain more elements. By gradually closing the accuracy gap between abstraction levels, the system model can maintain intact. Essentially, this also complies with the model-based design principle. Centered on a common model representation
CHAPTER 2. BACKGROUND

![System Level Design Abstraction](54)

(in the form of SLD specification), abstraction can be applied to reflect the features.

The modeling tools must be built upon a formal definition that is formally definable and analyzed. For SLD, this is accomplished by using System-Level Design Language (SLDL). In the following section, we will briefly overviews current prevailing SLDL approaches and focus on SpecC [54, 29] - our SLDL choice in this thesis.

2.3.2 System Level Design Languages

System Level Design Languages (SLDLs) are the core of ESL design flows as all system-level analysis, simulation and synthesis tools built around these languages. They have appeared to capture complex systems at the system-level in order to reduce the complexity of the MPSoC exploration. More precisely, the input specification, intermediate presentation and model structures are all captured using an SLDL. In this section, we will look into a few popular languages currently being used in system-level domain.

C: C language has been the dominating choice in embedded software programming for decades. Designed as an imperative language, C is suitable for describing the software perspective of the system. However, certain C features such as pointers, arrays, malloc, and global variables, make it ambiguous and non-analyzable. Nevertheless, a certain subset of C has been supported for

---

2For the purpose of this thesis, we have chosen SpecC [54] as the SLDL. The principles and concepts, however, are equally transferable to other SLDL, such as SystemC.
CHAPTER 2. BACKGROUND

the hardware perceptive by several vendors and solutions. Synthesizable constructs of the standard ANSI C language can be used in High-Level Synthesis. To name a few, HardwareC [74] and Celoxica’s Handle-C [8] are some of the subsets/variants of C used for hardware synthesis. The popularity of the C language has made it as the foundation for other SLDLs, such as SystemC and SpecC.

**SystemC:** SystemC [94 57] is a C++ extension adding model concurrency, structural hierarchy, signals, bitvector data types, etc., via extra C++ library supports. C++ classes are inherently used for model hierarchy and thread libraries are leveraged for the concurrency execution semantic. Stemmed from the C++ language, SystemC is highly adaptable due to the vast availability of C++ compiler/debugger as well as development environment. SystemC has been standardized and received a consortium of supports ranging from leading industrial partners and academia in EDA.

**SpecC:** Similar to SystemC, SpecC [54 29] is another true system-level language based on C. It comes at a superset of the ANSI C language. It introduced new language semantic to explicitly construct model hierarchy using “behaviors” and communication using “channels”. SpecC supports naturally timing and concurrency as part of the new language features. Unlike SystemC, which relies on any existing C++ compiling toolchain, SpecC requires new compilers and analysis tools to handle new language constructs. Nevertheless, without the syntactical bondage of an imperative language such as C++, SpecC brought in new blood for accurate modeling behavior with concise grammar and clean syntax. For instance, SpecC inherently supports more execution semantic compare to SystemC, such as sequential, concurrent, FSM or pipelined.

**System Verilog:** Instead of bridging software languages to hardware like SystemC or SpecC, System Verilog [110] aims to extend the Verilog HDL (hardware) with system-level modeling constructs. For example, it introduces classes to couple with object-oriented programming paradigm, abstract interfaces, new system-level data types, dynamic arrays, C-like structures, assertions, etc. However, many of the added features are mainly for simulation and verification purpose. Only a limited subset of these features supports synthesis. Furthermore, it requires steeper learning curve for training hardware engineers to adapt software solutions [11].

**UML:** Unified Modeling Language (UML) [2] by Object Management Group (OMG) was proposed for design issues of meta applications and enterprise systems. UML adapts a relax modeling semantic that can be utilized to describe the system-level features from a structural perspective. It supports hierarchy compositions, execution sequences, and data abstractions. However, UML is often considered too abstract for direct system modeling and, therefore, used as an early design tool for modeling the abstract system requirements.
CHAPTER 2. BACKGROUND

SysML: As a subset of UML, SysML [1] is a general-purpose modeling language with additional extensions for Systems Engineering. SysML supports the specification, analysis, design, and verification and validation of broad categories of complex systems such as hardware, software, information, processes, and even facilities. Some subsets are synthesizable [99]. There are a few works [99, 95] that investigated the system level design using SysML and SystemC.

Irrespective of the language peculiarity, most of the supplied language tools specify a subset of analyzable/synthesizable features within the entire language domains to guide designers to create system models to be suitable for the synthesis purpose. This research focuses on the SLDLs that expose system-level features and challenges. Thus, SpecC is the SLDL choice for the work in this thesis. As a true superset of ANSI C, SpecC includes features like signals, wait, notify, etc., to support hardware description. It supports constructs for hierarchical description of system components and provides native support to describe parallel and pipeline execution. With all these features, the designer has the flexibility to choose and describe the system at any desired level of abstraction.

Built upon SLDL, SLD environments (SLDE) automate the system-level design and exploration flow. SLDEs, such as PeaCE [60] and SoC Environment (SCE) [56] are developed to aid the specification creation, analysis, verification, validation, and synthesis through iterative refinement using SLDLs. SLD methodologies (SLDL and SLDE) reduce development cost and increase productivity for large and complex MPSoC designs.

2.4 Related Work

This section briefly discuses relevant research. In general, we categorize them into three major categories: bridging design gaps, flexibility overhead on granularity, and flexibility overhead on SW synthesis.

2.4.1 Bridging Design Gaps

Previous research has explored the automation/productivity gap in the field of EDA attempting to joint distinct domains [27, 108, 120]. Each of the work targeted a specific subset of the gap in a particular area (i.e. flat C code or abstract models) to improve the overall productivity and efficiency. However, the Specification Gap between ALD and SLD has not received much attention yet. Nevertheless, manually modifying models across multiple abstractions has been shown to be
CHAPTER 2. BACKGROUND

tedious, time-consuming, error-prone and highly unreliable or even infeasible in modern complex MPSoC systems [67, 81].

2.4.1.1 Bridging Abstractions

Spanning across design abstractions is the key element in shielding users from the overwhelming complexity of the entire design. In [78, 90, 117], language conversions were intensively studied and proposed to transfer models in UML/SysML to SystemC. These approaches captured the hierarchical information annotated in UML diagrams. Designers are enabled to construct system models abstractly in a top-down design flow established to simplify the model description and analysis. Designers can describe the majority of the design in UML followed by a language-to-language transformation to SystemC together with generated application code for simulation and implementation. The International Telecommunication Union (ITU) defined the Specification and Description language (SDL) [41] originally to describe telecommunication protocols, but was later extended for defining system level specifications. A user guided conversion of SDL to HW/SW architecture in C/VHDL was proposed for functional partitioning [76]. The Compaan tool-chain [72] focuses on transforming MATLAB applications into a Kahn Process Network for architecture exploration on MPSoC. Unlike these approaches most of which in this category primarily concentrated on the structural conversion whereas behavior transformation is less studied, we focus on how to capture both structural and behavioral characteristics of the design across multiple abstraction levels.

2.4.1.2 Language Transformation

Instead of bridging the abstractions directly, some approaches seek to combine the strength of different domain specific languages for particular targets trying to reduce the overall design complexity. SystemJ [58] extended Java with synchronous reactive semantics originated from Esterel. It combined Esterel and Java to better describe specification and modeling for reactive embedded processors. Esterel and C language (ECL) [75] was designed to leverage the strength of both languages to be more versatile for specifying system designs with reactive Esterel code and data dominated C code. These approaches, however, are less concerned about creating a unified design flow by extending one environment to another whereas they instead care more about how designs can be validated and synthesized for particular targets or platforms.
2.4.1.3 MATLAB/Simulink and ESL Design

MATLAB/Simulink has drawn research attention in multiple application domains related to SLDLs. In [14], they designed a multi-abstraction model co-simulation in a unified environment by connecting MATLAB and SystemC. Residing on the level of the algorithm environment, a co-simulation and verification environment is designed to break the simulation barrier between multiple domains such as MATLAB and SystemC. In [18], the authors proposed to transfer Simulink models to SCADE/Lustre in the Time-Triggered Architectures (TTA) domain. Others [10] aimed to address the continuous aspect of MATLAB/Simulink in SystemC or transforming Simulink models via a Synchronous Reactive (SR) layer to achieve correct-by-construction semantics between Simulink and ASCET [42]. Instead of focusing on verification or high-level simulation across Simulink and SLDLs, our approaches enrich the overall flow from Simulink to heterogeneous exploration and implementation, which provides the framework for simulation and verification in both algorithm-level and system-level designs.

2.4.1.4 Generating SLDL Specifications

A top-down refinement flow from the high level modeling in Simulink to TLM (Transaction Level Modeling) model in SystemC is demonstrated in [61] and [97]. Their work shows a prototype platform where Simulink models can describe the overall system requirements given hardware related information. In their approach, the refinement and component exploration happens at the algorithm developing environment, and it is designed to be annotated in Simulink model directly, which may require additional model modification. Their approach focused mainly on design refinement and less of jointing two environments together. Therefore, providing the tools and features of SLD to a higher level user was not yet particularly addressed in their approach.

With the goal of generating SLDL automatically, in [21], they proposed a work of transforming flat C code into a structured SpecC specification using a recoding scheme. Their work helped designer to construct the necessary structural hierarchy so that the system model is analyzable and synthesizable. It is a similar approach to the work we proposed, but with a different scope and source target. They target a different input language, namely C. Our approach, on the other hand, reaches higher abstraction by using Simulink as input models. In addition, human knowledge and guidance are required in the recoding process to interactively decide the model structure and the results show significant productivity gains through a substantial modeling time reduction whereas in our approach, the fully automated flow requires no interactions during the model transition. Users
CHAPTER 2. BACKGROUND

can interact with the new design flow from the top Simulink models along with decisions at various stages.

2.4.1.5 Shortening Development Cycles

Industry also focuses on speeding up the process of model creation, configuration and deployment. In particular, the V diagram \[4\] is frequently referred to illustrate the attention of model-based approaches to carry the design across multi-domains. Several methods on how to shorten the V diagram have been studied \[3, 92\] using model-based designs and code generation tools. Simulink Coder and Simulink Embedded Coder (SEC) \[87\] are capable of generating target especially embedded C code from Simulink models. However, it is limited to pre-defined target platforms and binary execution without heterogeneous platform exploration. Simulink can also generate a SystemC testbench to validate the model at the TLM level \[89\], however, its sole focus is being used as a testbench for validation purpose only. MATRIXx from NI \[3\] also focuses on the model-based control software that is capable of attaching to real-time hardware for quick control prototyping and hardware in the loop. However, heterogeneous platform exploration with VP generation is not part of their design methodology, which makes them not suitable for large and complex modern design space.

2.4.1.6 Unified System Synthesis

A rather different approach to treating the gap is to entirely avoid it by creating a unified system synthesis environment. Ptolemy \[39\] and Kepler \[7\] were proposed for modeling, simulating and deploying heterogeneous components concentrating on concurrent real-time systems in a fully integrated environment. However, spanning models across distinct domains as a system synthesis approach poses numerous challenges such as model semantic, design environments, simulation integrity, validation, verification and specially the model complexity, which require more work to be accomplished in these approaches. In our work, instead of starting from scratch, we explore to merge two mature, well-developed and user-endorsed environments together to bring the tributes to both communities in one accelerated design cycle that tremendously benefits designers on model analyzability, explorability and deployment on heterogeneous platforms.
2.4.2 Flexibility Overhead: Granularity Challenges

Synthesizing algorithm models to specifications has not received much attention. The majority of the research has focused on system-level design given an authored system specification. Similar works on combining multiple abstraction layers [117] proposed to integrate UML in the context of SoC design as part of methodologies. A pure language abstraction bridging was provided in [100] to convert UML to SystemC. However, these approaches primarily focused on capturing hierarchical structure to maintain a structural level matching. Thus behavior translation on the functional aspect is often less examined and the issue of the trade-off was not observed. Recent work explored on the topic of translating Simulink to SystemC TLM [68, 97]. They proposed to bring the refinement and exploration phase into Simulink, which requires updating and modifying Simulink models. However, the issue of model granularity impact was not discussed as their work focused more on exploration side. Unlike their approach, we address the issue and explore opportunities once multiple abstraction layers are jointed.

On the other hand, the granularity issue appeared in other domains. A re-coding approach [21] was designed to convert plain C code into a structured SLDL specification. Their approach however, relied on a manual granularity selection as there was no abstraction level to be bridged and no inter-component optimization existed. In SW/HW co-design partitioning [63, 40], they proposed to identify the granularity of mapping applications ranging from statements to blocks to procedures with an annealing approach or cost function. Unlike their approaches tailored to specific platform, our work emphasizes on the specification generation, offering more flexibility and generality.

2.4.3 Flexibility Overhead: SW Synthesis

Embedded software synthesis has received much research attention. Approaches ranging from high-level analysis and synthesis approaches often utilize specialized Models of Computation. Examples include POLIS [9] (Co-Design Finite State Machine), DESCARTES [101] (ADF and an extended SDF), and Cortadella et al. [25] (petri nets). Integrated Development Environments (IDEs), at the other end of the spectrum, typically provide limited automation support but aim to simplify manual development (e.g. Eclipse [43] with its wide range of plug-in modules). The focus of this paper is system-level design, where HW and SW are jointly captured in an SLDL and embedded SW is synthesized out of an SLDL specification. Software generation enables transferring the results from abstract modeling into a real implementation. Herrara et al. [64] introduced software generation from SystemC models by overloading SystemC library elements. This approach,
in advantage, uses the same model both for specification and target execution; however, it potentially replicates aspects of the simulation engine on the target. Other approaches focus on the RTOS integration, such as Krause et al. [73] that generates source code from SystemC for application to RTOS mapping, and Gauthier et al. [52] for generation of an application-specific operating system.

Some research effort has been invested into solving the overhead in the design flexibility offered by imperative programming paradigms such as object-oriented languages. Qian et al. [98] investigated through class hierarchy analysis (CHA) in a Java JIT environment aiming to reduce the dispatch cost through inlining. Sundaresan et al. [109] partially avoids the overhead through static call graph analysis in Java. In [35], they performed empirical analysis of time-efficiency in generic OO languages such as C++, C# and JAVA. Others [24] investigated into the evaluation of performance and power of C++ code on embedded processors.

Although our DDO-reduction approach addresses similar aspects in dynamic type analysis, many differences exist. While [109, 98] looked into function call type analysis in generic OO languages, such as Java, our approach targets SLDLs, analyzing connectivity between SLDL modules particularly for embedded target synthesis. Leveraging domain knowledge (targeting modules) and mapping information, our approach reduces the call site analysis graph size (potentially being faster in analysis). We distinguish modules shared across different processing cores that support efficient multi-core code generation. In addition, [109, 35, 24] mainly focused on improving the analysis. Rather, our DDO-reduction turns the analysis into tangible results for synthesized target C-code improving speed and readability. On top of that, our work evaluates the performance benefits in detail for varying execution targets (e.g. x86 and ARM).

2.5 Summary

Most EDA research has focused on exploiting the specification model in system-level context or modeling complex systems in abstract levels. Beside our work, bridging multiple abstraction levels between ALD and SLD has recently attracted some attention [102, 126]. However, these research focuses on different particular domains, and are not enabling an algorithm-architecture co-design flow. Moreover, due to the early stage of these research, optimization opportunities such as granularity and scheduling have not been considered in their work. This thesis targets the domain that have not been fully explored - that is to how to effortlessly obtain specifications, with optimization in place, from higher abstraction models aiming to create a new algorithm-architecture
CHAPTER 2. BACKGROUND

coe-design flow. Therefore, our work introduces new aspects on the grand EDA challenge of accelerating the concepts-to-silicon design cycles.
Chapter 3

Specification Synthesis

This chapter overviews the specification synthesis process. We start with the synthesis foundation to examine the semantic difference between Simulink and SpecC in order to assess the synthesis quality or trade-off.

3.1 Specification Synthesis Foundations

As an algorithm modeling tool, Simulink models system dynamics in both continuous and discrete domains through a flexible and non-rigorous semantics based on the dynamic theory [123]. However, not all elements supported in Simulink simulation semantics can be (or should be) synthesized to hardware or software. On the contrary, SLDLs seek stricter semantics to meet the stringent synthesis requirements. Driven by underlying architectures, it imposes a semantics of discrete systems for constructing synthesizable designs [103].

Therefore, a synthesizable subset of Simulink primitives to be supported by Algo2Spec needs to be identified. Restricting the scope of Simulink semantics for the synthesis purpose has been shown effective in earlier approaches [114]. The current design of Algo2Spec supports the following Simulink primitives:

1. Environment. Discrete and fixed step solvers are supported. Other categories of solver or variable steps are not supported.

2. Blocks. For generating blocks, Algo2Spec uses Simulink Embedded Coder (SEC) for intermediate C code generation. As such, Algo2Spec inherits the scope of blocks supported by
CHAPTER 3. SPECIFICATION SYNTHESIS

SEC. Additionally, complex blocks, asynchronous blocks and blocks with continuous time are not supported.

3. **Sampling Rates.** Given that Simulink has a time-based simulation semantics, we fully support Single-Rate in the current release. Multi-Rate, however, is supported partially with constraints. As Multi-Rate is built upon Single-Rate semantics, carefully studying the Single-Rate is the foundation for further exploration of Multi-Rate systems. In this work, we focus on Single-Rate systems.

The current supported Simulink subset already covers a large variety of Simulink models. Moreover, *Algo2Spec* provides a flexible and expandable framework that simplifies expanding the support for more Simulink elements.

The validity of specification synthesis depends on the correct semantic reconstruction of Simulink in SLDL. As both Simulink and SLDL are composable programming paradigms, we consider the validity of the synthesis in a modeling perspective as follows:

1. **Functional reassembly.** SEC produces embedded ANSI C code for a Simulink block if it is permissible for code generation. The code generation procedure ensures that the numerical results of a Simulink block can be represented in C routines pertained to the dynamic theory. SLDLs, being described in imperative programming languages (C/C++), rely on C code for functional constructs. We ensure that each SLDL element matches the numerical results of the corresponding Simulink block by embedding the SEC generated C code into the SLDL element.

2. **Model composition.** Simulink models are functionally composable; that is complex functionality can be constructed by exercising composition rules, such as grouping, looping, or enabling, on sub-blocks with valid signal connections. Similar hierarchical composition is inherently supported by SLDLs. However, in order to reassemble the Simulink hierarchical structure in SLDL, structural semantics (see Section 3.1.1) analysis and mapping are necessary as language constructs are different. The formal language validation methods such as model transformation [6] are part of the future work.

3. **Behavioral scheduling.** Besides valid functional compositions, the numerical validity also relies on the correct execution order. This is discussed in Section 3.1.2.

We examine the structural and execution semantics in more detail for understanding the limitations and potential trade-offs in the specification synthesis process.
CHAPTER 3. SPECIFICATION SYNTHESIS

3.1.1 Structural Semantics

Structural semantics is defined as the compositional elements and rules. It specifies what elements can be combined to create a new element that is structurally valid. The essential structural semantics of Simulink and SLDL are shown in Figure 3.1. For brevity, Blocks (Figure 3.1a) represent both leaf (atomic) and hierarchical Simulink blocks, whereas only leaf blocks contain Dynamic Functions. Classes (Figure 3.1b) represent both SLDL leaf and hierarchical behaviors. The basic block units in Simulink are atomic blocks and in SLDL are simply noted as leaf behavior.

The Simulink atomic blocks include primitive blocks (e.g. blocks predefined in toolboxes), model reference blocks, S-Function blocks and other non-dividable blocks. As atomic blocks cannot be further decomposed into small sub-blocks, they contain the actual computation, such as addition or complex kernels. The computation in such block is formulated through dynamic functions that define the block state corresponding to inputs (Inports), outputs (Outports), shown in Figure 3.1a. The dynamic functions of a Simulink block can be generated to ANSI C procedures by SEC expressing the identical functionality. On the other hand, SLDL classes contain member functions written in ANSI C code. This indicates that the SLDL member functions are compatible for capturing the dynamic functions in Simulink, which retains the Simulink functionality in SLDLs.

SLDL inherently supports ports as required in Simulink. Besides port direction, Simulink defines the data transmitted through ports can be composed by data type and width, which are also supported by SLDL ports. A signal in Simulink is composed of a trunk and branches, where a branch is defined as the split segment from the trunk signal. SLDL semantics, however, differs in

---

1 Classes are an abstract representation of behaviors and channel.
2 In the rest of the thesis, we will use Behaviors to denote elements in the SLDL. The concept is equivalent to modules or blocks.
3 Complex signals in Simulink are currently not supported such as non-virtual buses.
that signals are handled as variables (or channels) using communication primitives to address signal routing. The SpecC SLDL does not distinguish trunk and branches natively. This necessitates the insertion of extra elements to maintain the semantic correctness by signal synthesis (Section 3.2.2.1) to resolve the fan-out connection problem as exposed by Simulink signals.

Hierarchical compositions are supported in both semantics through Simulink sub-blocks or SLDL instances. However, slight discrepancies exist between the distinctions for types and instances in these two domains.

Figure 3.2 shows a simple Simulink to SLDL comparison. A Simulink hierarchical block A consists of one inport (In1), one output (Out1), two sub-blocks: B and C. Signal s1, s2, and s3 connect these blocks to corresponding ports. Simulink does not explicitly distinguish block instances from block types. This means Block B and C are their own types and instances at the same time using the name as a unique identifier. Simulink restricts this in the modeling environment by disallowing the same block names in the visible namespace (i.e. within the same block). In SLDLs, however, types and instances need to be named differently. Thus, Algo2Spec explicitly distinguishes these two different kinds of naming by using Simulink block types and extra suffixes for instance name creation.
CHAPTER 3. SPECIFICATION SYNTHESIS

Figure 3.2b shows the synthesized SLDL behavior by Algo2Spec for the Simulink example. Behavior A contains two child behaviors of type: B and C, as instances B_0 and C_0 with _0 as the instance naming suffix. The ports are named identically as in the Simulink model. Variables are instantiated to connect ports and behaviors to maintain the correct signal routing (float as an example) and connection optimizations are applied (e.g. s1 and s3 are optimized away). The details will be discussed in hierarchical synthesis (Section 3.2.2).

3.1.2 Execution Semantics

Simulink evaluates blocks in the time domain [84] for simulation. Derived from the dynamic system theory, it primarily describes the relationship among input, state and output regarding time changes [23]. In the scope of our work, continuous time is not considered. Hence, we limited the execution semantics to discrete and fixed step solver\(^4\). In essence, the solver determines when a block can be invoked. We focus on Single-Rate systems - blocks will be invoked depending on the execution dependencies.

SLDLs, such as SystemC and SpecC, are based on discrete event simulation semantics [93]. SpecC allows a behavioral hierarchy (e.g. sequential, pipeline, parallel, FSM), which captures how behaviors can be scheduled. During the specification synthesis, the scheduling of the entire design can be statically determined by Algo2Spec through dependency analysis. The scheduling is then realized using behavioral hierarchy, which eliminates explicit requirements of synchronization through events. With a statically determinable execution sequence, the SpecC semantics can be used to control scheduling in discrete simulation honoring dependencies in the original Simulink models.

The semantic comparison directs the synthesis process from Simulink to SpecC to achieve both structural and functional validity. The next section looks into the Algo2Spec internal design.

3.2 Algo2Spec Design

This section first overviews the internal structure of Algo2Spec followed by discussing its components in detail to explain the synthesis process. Figure 3.3 illustrates the overall synthesis flow. Algo2Spec consists of two main components: the Model Generator (front-end) and the Model Linker (back-end) connected through intermediately generated files. With a Simulink algorithm\(^4\) A Simulink solver computes the next simulation time with an interval called step size.
model as input, the Model Generator first starts a sanity check ensuring the model is suitable for synthesis. For instance, it rejects a model with unconnected ports or unresolved algebraic loops (i.e. erroneous Simulink models).

Algo2Spec implements a DFS-based tree traversal following the bottom-up synthesis principle for full model coverage with low runtime complexity. The leaves of the model tree, atomic Simulink blocks, are synthesized by the module Leaf Synthesizer (see Section 3.2.1) to SLDL leaf behaviors. The Hierarchy Synthesizer synthesizes Simulink subsystem blocks to SLDL composite behaviors (i.e. hierarchical nodes) (Section 3.2.2). Lastly, the Model Linker imports the generated behaviors to construct the overall specification and resolves external linkages. The front-end is implemented in MATLAB for the maximum compatibility, portability and integration with the Simulink environment. The back-end is realized based on the SpecC Compiler (SCC) [31]. During the synthesis process, the designer can direct two optimization opportunities: scheduling decision (Section 3.2.2.3) and granularity decision (Section 3.2.3), to flexibly adjust and fine-tune the overall system performance.
3.2.1 Leaf Synthesizer

The aforementioned semantic analysis in Section 3.1 shows that the functionality of a single Simulink block can be expressed in C routines generated by SEC. The Leaf Synthesizer, thereby, utilizes the generated C code for SLDL behavior functionalities.

For each identified Simulink leaf, the Leaf Synthesizer invokes SEC to generate C functions. The produced code, however, is insufficient for a synthesizable SLDL construction due to the lack of structures and analyzable communication interfaces \[21\]. Moreover, the generated routines contain global functions and variables (see Figure 3.5a) that are undesirable for modular embedded implementation. Additionally, the appropriate port allocation and mapping for each block need to be ensured for inter-behavior communication.

Thus, the Leaf Synthesizer employs a three-step approach to synthesize a leaf block depicted in Figure 3.3 (1) Generating an empty SLDL behavior skeleton with captured interfaces (Skeleton Generator); (2) Invoking SEC to produce the application C code; (3) Integrating the C code into the behavior skeleton (Integrator). These steps result in synthesizing a fully functional and synthesizable SLDL leaf behavior for a given Simulink atomic block.

3.2.1.1 Skeleton Generator

The Skeleton Generator outputs an empty SLDL behavior and prepares its interface connections for communication analyzability. It identifies the block port structures in Simulink and creates SLDL equivalents. For this, it first analyzes the data type and access pattern of ports of a Simulink block. It iterates through all ports of a block to detect the data type, width as well as direction through block and port parsing.

Simulink block ports have similar semantics to SLDL behavior ports which allow simple mapping of the data types and data width (see Section 3.1.1). A static mapping of Simulink standard data types to ANSI C data types is predefined in a mapping lookup table. Simulink only supports two directions: "inport" and "outport", which corresponds to "in" and "out" port directions in SLDLs.

An illustrative example in Figure 3.4b shows the SLDL skeleton capturing the Simulink block ports of Figure 3.4a. The Skeleton Generator determines the Simulink port type as "float". Incoming ports are established as In1 and In2 and outgoing ports are created as Out1 and Out2. Inside the behavior, empty bodies of initialization (init), termination (term) and main methods (main) are created as place-holders to be filled later by the Integrator.
In order for \texttt{init} and \texttt{term} to be triggered outside of the behavioral hierarchy, each SLDL behavior implements a standard interface \texttt{iSimBlk}. The interface method \texttt{init} is called hierarchically through all behaviors in a top-down approach prior to all behaviors’ main execution. Likewise, \texttt{term} is called upon model termination for all behaviors. The method \texttt{main} will later contain the computation, inserted by the \textit{Integrator} (see Section \ref{sec:3.2.1}), to be triggered during the model execution period.

3.2.1.2 Integrator

The \textit{Integrator} inserts the computation extracted from the SEC generated application code into the prepared skeleton. To facilitate systematic code manipulation, we transform the abstract syntax tree of Simulink blocks into an Intermediate Representation (IR). For the work in this thesis, \textit{Algo2Spec} uses the Syntax Independent Representation (SIR) \cite{28} as the IR\textsuperscript{5}. In particular, the Integrator involves three major tasks: (1) preprocessor, (2) code insertion and localization and (3) port to variable connections.

As Simulink has a different set of reserved keywords, a number of SpecC reserved keywords may appear in the generated application code. For instance, ”in”, ”out” and ”range” are all reserved keywords in SpecC but can be used unrestrictedly in Simulink for model names, port names or embedded MATLAB functions. In order to obtain the code in the legitimate SpecC syntax, the \textit{Integrator} replaces the SLDL reserved keywords in the application code with extra suffixes at

\textsuperscript{5}SIR is primarily the internal representation of the SpecC SLDL, however producing SystemC have also been shown in \cite{119}.
CHAPTER 3. SPECIFICATION SYNTHESIS

the textual level. After the preprocessing, the model can be cleanly parsed into IR.

(a) SEC Generated Code for Single Block

(b) Single Behavior Example in SpecC

Figure 3.5: Leaf Synthesis Example

The major step of the Integrator is to insert the generated C code and localize the variables and procedures of the code into the SLDL leaf skeleton. The unprocessed application code shown in Figure 3.5a contains global user-defined types (line a:2-5) together with global variables (line a:7-12). These variables are used in global functions (line a:14-27). We omit the termination routine for brevity in this example as it shares similar principles. As mentioned earlier, in order to obtain a clean partitioning and synthesis design, an SLDL specification requires all computation routines to be contained in a behavior without referencing to global structures. Hence, global types and variables need to be localized.
Due to the type dependencies, the Integrator first transforms the global user-defined type (i.e. struct in line a:2-5) into a local user-defined type in the corresponding behavior (line b:5-8). After the type localization, the global variables (line a:7-12) that depend on those particular user-defined types are then migrated into the behavior scope (line b:9-13). They become the instances of the localized data types. Similarly, the global functions (line a:14-26) are relocated into the behavior scope (line b:14-16). Following this order ensures that the entire computation routines operate locally inside the behavior. Block_step is re-scoped into the behavior (line b:14) with the calling signature removed (light gray) since all parameters are now visible in the same scope. Likewise, Block_initialization is directly inserted into the function body of init in the behavior (line b:16, light gray). As demonstrated in Figure 3.5b all types, variables and functions that were previously in the global scope in the application C code are now localized to the behavior scope in the SLDL behavior. Note that the Integrator minimally affects the SEC generated computation code as the internal data structure is not altered during the insertion and localization. The intertwined global data structures and procedures in the generated application C code are carefully eliminated.

Although the computation routines are in place, variables used inside the behavior need to be connected to the proper ports of the behavior’s interface. This leads to the last step in the leaf synthesis, the port to variable connections.

SpecC does not allow any write access to an "in" port, and no read access to an "out" port. SEC generated code, however, does not enforce such strict semantics. To decouple the port and variable access, Algo2Spec opts for different ports and internal variables by inserting assignment statements at the appropriate location in the behavior to accessing ports. Meanwhile, the original signal names are maintained to ease understanding and debugging.

To connect the ports of a behavior to the corresponding internal variables, the Integrator examines variables across the step function and behavior ports to determine the access pattern upon data directions (in/out). It injects assignment statements before the step invocation to connect the input ports to the internal variables for passing in data. Similarly, new statements are inserted underneath the step function to connect the variables holding the results to outgoing ports. Shown as orange boxes in Figure 3.5, the local variables Block_U_In1, Block_U_In2, Block_Y_Out1, and Block_Y_Out2 were the previous global variables (line a:9-12) that carried in and out external data of the step function. In the SLDL, they are simply connected to In1, In2, Out1, and Out2 respectively.

By making such connections, Algo2Spec strictly separates the communication from computation. The block execution follows a sequence of communication (i.e. assignment from inport),
CHAPTER 3. SPECIFICATION SYNTHESIS

computation (i.e. `Block_step()`) and communication (i.e. assignment to outport). This fully
emulates the Simulink block execution semantics and more importantly, allows constructing a clean
block behavior for HW/SW partitioning in later exploration phases.

As a result, *Leaf Synthesizer* integrates the generated code into an SLDL behavior skeleton
and produces a valid and synthesizable leaf behavior. The process recursively iterates through the
total model tree to synthesize each encountered leaf block to behavior. Further scheduling and
connection of behaviors are addressed in generating hierarchical behaviors.

3.2.2 Hierarchy Synthesizer

To address the Simulink subsystem blocks and preserve the structural semantics of the
model, the *Hierarchy Synthesizer* involves four tasks:

1. Composing the communication ports of a hierarchical behavior
2. Signal synthesis within a hierarchical behavior
3. Child behavior instantiation
4. Child behavior scheduling

The first task is accomplished by re-using the skeleton generation procedure of the *Leaf Syn-
thesizer* to obtain communication ports of a hierarchical behavior. The other tasks are processed
step-wise in different modules as shown in Figure 3.3. To give an intuitive example of hierarchy
synthesis, Figure 3.6a shows a Simulink example. The corresponding synthesized SLDL model
is shown in Figure 3.6b. In this example, all hierarchical elements are preserved, and all original
ports with data type and port directions are maintained for each behavior. Nevertheless, some extra
blocks, inserted by the *Signal Synthesizer*, are necessary for the SLDL communication composition,
and these blocks are labeled with the prefix *TE* (darker color boxes).
CHAPTER 3. SPECIFICATION SYNTHESIS

3.2.2.1 Signal Synthesizer

Simulink signals must be synthesized into SLDL communication primitives to route traffic between SLDL behaviors. However, the SEC only generates code for individual blocks, not dealing with communication across blocks. For instance, the code snippet in Figure 3.5 only contains signal type and width. Therefore, the Signal Synthesizer realizes communication across sub-behaviors in a hierarchical behavior.

Simulink is a software-oriented approach with the semantics defining that every point of a signal holds the same value at each time step. SLDL, however, does not endorse such a liberal definition. Instead, it requires that port mapping and signal routing must be explicitly addressed for exploration and implementation requirements. Thus, the semantic differences exist between Simulink and SLDL in the case of signal connections. As Simulink does not enforce data directions,
CHAPTER 3. SPECIFICATION SYNTHESIS

Table 3.1: Signal Topology Categories

<table>
<thead>
<tr>
<th>Topology Type</th>
<th>Label in Figure 3.6</th>
<th>TE Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>block to block</td>
<td>c</td>
<td>No</td>
</tr>
<tr>
<td>block to multiple blocks</td>
<td>a</td>
<td>No</td>
</tr>
<tr>
<td>block to port</td>
<td>f</td>
<td>No</td>
</tr>
<tr>
<td>block to multiple ports</td>
<td>g</td>
<td>Yes</td>
</tr>
<tr>
<td>block to blocks + ports</td>
<td>e</td>
<td>Yes</td>
</tr>
<tr>
<td>port to block</td>
<td>b.branch</td>
<td>No</td>
</tr>
<tr>
<td>port to multiple blocks</td>
<td>b</td>
<td>No</td>
</tr>
<tr>
<td>port to port</td>
<td>d.branch</td>
<td>Yes</td>
</tr>
<tr>
<td>port to multiple ports</td>
<td>d.branch</td>
<td>Yes</td>
</tr>
<tr>
<td>port to blocks + ports</td>
<td>d</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Signal splits are permissible across block boundaries and data directions. An example is signal e in Figure 3.6a. It connects A6 with both out2 as well as A7. Such connection cannot be replicated in SLDL since Out requires an output port in SLDL on A5 which then could not be read by A7. This requires a semantic preserving translation.

The signal routing principles of Simulink models can be categorized into block-driving and port-driving. Block-driving is defined as the source of the signal is a block, whereas port-driving is defined as the source of the signal is a port. With this classification, 9 combinations can be derived as listed in Table 3.1. For example, the signal a in Figure 3.6a belongs to block to multiple blocks as block A propagates the signal a to block A2 and A5 simultaneously. Table 3.1 summarizes the signal topology categorization of Figure 3.6a.

For brevity, some topology categories are shown in the Figure 3.6a as a branch of a signal. An example is port to block can be viewed as the branch of the signal b connecting In1 to A3 or A4. SLDL supports most signal routing with communication primitives. However, for block to multiple ports, block to blocks + ports and other three categories, Transition Elements (TE) are introduced to maintain SLDL signal semantic correctness. To illustrate, the signal e cannot be presented identically in SLDL because it is synthesized to a shared variable between A6 and A7 and thus cannot be connected to out2. To resolve this, TE_e is inserted in the SLDL model to facilitate the SLDL routing semantics. Plotted in Figure 3.6b similar principles apply to TE_d and TE_g.

These topology categories serve as rule-based references for resolving signal intricacies during signal synthesis. Inserting extra structural elements does not increase the performance overhead, as during the back-end synthesis process, optimization tools in the compiler [122] will inline.
CHAPTER 3. SPECIFICATION SYNTHESIS

Figure 3.7: Hierarchy Synthesis Example: the SLDL Source Code

these blocks as they merely serve as a routing redirection without computation.

A source code example of hierarchy synthesis is given in Figure 3.7. Line 19 shows that the signal \( a \) in Figure 3.6 is instantiated in SLDL Figure 3.6b. The width and data type in ANSI C of the signal is analyzed and realized by the Signal Synthesizer. Shared variables are used in the current implementation. Channels will be deployed to exploit the parallelism of multi-rate systems in the future. After the signals are synthesized into SLDL primitives, they need to be connected to child behaviors within a hierarchical behavior.

3.2.2.2 Child Instantiation

The instantiated signal is used to connect interfaces among child behaviors in a composite behavior. Two different connections exist in Hierarchical Synthesizer: port-to-port or signal-to-port.

Port-to-port indicates that the external ports of the parent behavior are to be directly connected to the child behavior ports. Figure 3.7 shows a code snippet of the SLDL code for behavior A2 and Top - the top level block in Simulink, corresponding to the model visualized in Figure 3.6b. For the example of port-to-port connections, the inports of A3 and A4 in A2 are directly connected.
CHAPTER 3. SPECIFICATION SYNTHESIS

to their parent’s import (Line 3 and 4). For signal-to-port, the instantiated signal must be connected between a behavior that writes to the signal (producer) and other behaviors that read from it (consumer). Therefore, a direction analysis of port mapping of each child behaviors is conducted. The *Hierarchy Synthesizer* connects the signal between them accordingly. For example, Line 19 shows the instantiated signal as a shared variable array. Line 21 and 22 express that instantiated child behaviors, A1 and A5, with signal a connecting to both as A1 is the provider and A5 is the consumer of the signal (see Figure 3.6).

When child behaviors are instantiated with valid connections, multiple scheduling options exist. Different scheduling policies may vary in performance.

3.2.2.3 Scheduling

The *Hierarchy Synthesizer* utilizes the hierarchical block execution order in Simulink graphs for behavior scheduling, which is based on topological sort to resolve dependencies. Multiple scheduling policies can be selected without violating the block dependency. One policy: *faithful scheduling* emulates the simulation semantics of Simulink. All Simulink blocks are scheduled statically in a sorted dependency list and executed sequentially. Such policy enforces all SLDL behaviors to run sequentially in the same order.

A second scheduling policy: *pipeline scheduling* is a more aggressive policy to yield higher performance once the designer allows to re-time the model (i.e. inserting special delay blocks to construct a pipeline flow). For example, the Simulink HDL Coder generated code shows that blocks can be executed in a pipeline fashion with little overhead [88]. This scheduling policy (when chosen by the user) is synthesized into a pipelined SLDL model enabling block-level parallelism.

Another policy is *mixed scheduling*, which interleaves the scheduling of sequential, parallel and pipeline together guided by the designer as scheduling decision. As an example, once blocks are structurally independent, block parallelization becomes possible. The model in Figure 3.6b includes a split structure between A3 and A4. They have no dependency to each other and can be scheduled in parallel in the SLDL specification as shown textually in line 10-13 in Figure 3.7. For the purpose of this thesis, we focus on *mixed scheduling* with sequential and parallel explorations.

When the tree traversal reaches the top level, it concludes front-end model generation. The output of all intermediate hierarchical behaviors and leaf behaviors are passed into *Model Linker* (see Figure 3.5). It eventually produces a fully functional SLDL specification that captures hierarchy and communication structures of the original Simulink model.
CHAPTER 3. SPECIFICATION SYNTHESIS

Figure 3.8: Algorithm-Architecture Co-Design Flow
3.2.3 Granularity Discussions

Besides the scheduling policy, another optimization input controlled by the user is the granularity decision. Simulink blocks vary greatly in their amount of computation. A block may contain as little as a single operation (primitive functional blocks) or a complete algorithm (S-Function or MATLAB function). An important decision within the synthesis process is how much of the Simulink model hierarchy (i.e., granularity) should be retained in the SLDL specification. For example, it is not meaningful to expose single operations as individual SLDL behaviors (too little computation to be meaningful). Instead, a higher hierarchy level must be found to contain a meaningful amount of computation, which then is treated as a leaf.

The two extremes in granularity selection are the original (i.e., the finest) granularity, where all blocks are preserved in SLDL, and all merged (i.e., the coarsest), where only the top hierarchy is retained as one single "big" leaf. The finest-grained decision would miss the opportunity of block fusion as all blocks are generated separately and may cause unnecessary communication overhead in the SLDL. This in turn may downgrade simulation performance without offering impactful mapping alternatives. On the contrary, the coarsest-grained choice would cause the loss of hierarchy, which removes the possibility of heterogeneous exploration.

Algo2Spec, therefore, deploys a meet-in-the-middle decision with a threshold that represents a balance between overhead and exploration flexibility. In our current flow, Algo2Spec allows the granularity as a decision input (see Figure 3.3). The designer specifies the granularity as tunable parameters that can be iteratively improved by the designer given the fast feedback from the generated specification. In the experimental results, multiple iterations of identifying a proper granularity for a model are demonstrated to evaluate the granularity impact.

3.3 Algorithm-Architecture Co-Design Flow

By connecting Algorithm-Level Design and System-Level Design, Algo2Spec creates a new algorithm-architecture co-design flow. This section briefly summarizes the flow with illustrations of how Algo2Spec can assist current design process.

Following top-down design methodologies [45, 106] in modeling complex systems, the algorithm/architecture co-design flow also is embodied in a top-down approach expanding current system-level design methodologies to reach algorithm modeling context.

Figure 3.8 conceptually highlights the flow with added features such as multi-stage simu-
CHAPTER 3. SPECIFICATION SYNTHESIS

lation and multi-abstraction decisions. The pink annotated boxes and lines are novel contributions from our work, whereas black denotes existing components from Simulink or SCE. The dashed lines represent iterative improvement opportunities at various stages.

The flow starts with the designer modeling the algorithms in Simulink utilizing predefined primitives from a database or repository. For instance, the designer can experiment with various parameter settings to gain the desired detection accuracy in a vision algorithm. The focus at this stage is numerical and behavioral correctness, as well as design quality. Through immediate feedback (flow 1 in Figure 3.8), the designer is prompted to improve on algorithm details. As an example, a Simulink model is given on the left side in Figure 3.8.

Once the model is ready for system-level exploration, Algo2Spec performs the specification synthesis to produce the SLDL specification. During the synthesis, multiple decisions arise, such as granularity selection, block scheduling and the exposed parallelism. These decisions impact the suitability of mapping computation to different processing element classes. For example, mapping to custom hardware requires finer grained parallelism than mapping to a processor. Automating the generation of the system-level specification offers tremendous opportunities for optimization across layers and abstraction levels. This builds the foundation for optimizing specification synthesis using our framework.

After the specification is produced, the designer can inspect the results and, if necessary, reconfigure the Algo2Spec parameters such as granularity or scheduling (flow 2) or update the model from the top level (flow 3). In the middle left of Figure 3.8, the synthesized SpecC specification with the original granularity and the mixed scheduling is shown.

The specification enables the designer to proceed with system-level DSE, such as allocating PEs (e.g. processors, custom hardware components, or GPUs), selecting communication systems and memories, defining mapping of the computation to PEs, and system-level scheduling policies. The automatic specification refinement then actualizes these decisions in generating Virtual Platforms (VPs) at the level of TLM. The generated TLM is shown on the bottom left of the figure. With a CPU and a hardware (HW), the designer maps A1 and A2 (with A3 and A4) to the CPU while allocating the rest to HW.

Empowered by TLM, the designer can assess the overall performance, power and cost with fast yet accurate results. These results can direct different choices on DSE alternatives, such as architecture selection, computation allocation or different operating system scheduling choices, shown as flow 4 in the figure.

In addition to these DSE choices, the TLM performance analysis can also drive new al-
CHAPTER 3. SPECIFICATION SYNTHESIS

Figure 3.9: MJPEG Encoder Structural Hierarchy

Algorithm level decisions (flow 5 and 6). We have observed in the past often the need to change parallelism granularity and type depending on where algorithms are mapped to. Moreover, it is of critical importance to have platform-specific algorithm tuning such as converting from floating to fixed point computation or reducing memory bandwidth of a vision algorithm through the parameter compression. Again, by automating the path from algorithm to specification, these steps can be easily taken to ultimately yield more overall optimized designs on a given platform.

After completing both algorithm and system-level optimization and identifying a suitable algorithm/architecture combination, the generated VP serves as input to the back-end synthesis. The back-end synthesis produces implementations for custom hardware (through high-level synthesis), software (through software synthesis), and the glue interfaces in between.

3.4 Experimental Results

We demonstrate the effectiveness of Algo2Spec using a set of industrial strength applications: MJPEG encoder, MJPEG codec and Corner Detection. These models are taken from Simulink libraries in R2012a. First, we focus on a case study of the MJPEG video encoder highlighting exploration opportunities and DSE iterations to identify a suitable specification granularity. Afterwards, we generalize the observation by quantifying the productivity benefits for all applications.
CHAPTER 3. SPECIFICATION SYNTHESIS

3.4.1 MJPEG Encoder Synthesis Example

MJPEG [50] is a standardized video format in which each frame is compressed as a JPEG image separately. It is commonly used in video capturing applications and has high computation demands.

A partial view of the MJPEG model is shown in Figure 3.9a. The model consists of 140 blocks, 28 S-functions, and 35 subsystems. The model contains blocks with up to 8 levels in depth (we will later simply refer to as depth). For our experiments, the design encodes a 910-frame QVGA (320x240) video stream at 15 FPS. This demands an encoding time of less than 60.66 seconds in total.

Figure 3.9b views the structural hierarchy of the corresponding SLDL specification with granularity of depth 5 synthesized by Algo2Spec. For brevity, behaviors are arranged for better visualization and the details of blocks, signals and ports are omitted.

The specification retains the original model communication ports and hierarchy. Algo2Spec is capable of exploiting task-level parallelism. For instance, Tran1, Tran2 and Tran3 are scheduled in parallel as they are structurally concurrent in the algorithm model.

3.4.1.1 Early Estimations and Evaluations

During the DSE phase, architecture alternatives and specification combinations are explored (e.g. through behavior-to-PE mapping). Finding a suitable platform and mapping is simplified when the amount of blocks need to explored is relatively small. Otherwise, the effort for DSE explodes exponentially due to the sheer number of available blocks. Hence, it is necessary to identify the model granularity (i.e. number of blocks) to be synthesized, while meeting the system constraints such as timing or area. The joint design flow, with its feedbacks, enables designers to rapidly analyze design decisions to converge to an efficient granularity and mapping combination.

To estimate the model complexity, each generated specification is profiled using SCPROF [16] (part of the SCE environment) to determine computation and communication demands. Computation demands express the number of C-level operations executed in each leaf behavior. Communication demands show the total number of bytes in a leaf behavior including incoming and outgoing traffic. Note that the gathered demands at this point are independent of actual target selection and mapping (as they are yet defined). Instead, they are an indication of the complexity, but not necessarily the execution time, which may change depending on the execution target.
CHAPTER 3. SPECIFICATION SYNTHESIS

The early estimation analysis of MJPEG encoder is shown in Figure 3.10. Several design iterations are performed in order to select the proper model exploration granularity. The initial iteration has a depth of 2, and only the sub-blocks of Encoder_DUT are further synthesized (Figure 3.10a). BlkProc is the most computationally intense block with over 100K million operations, however, due to the coarse granularity this model is not suitable for mapping to multiple heterogeneous elements.

For the intermediate iteration, the designer directs to traverse into a depth of 3. The sub-blocks of all three blocks (Preproc, BlkProc and MatCon) are synthesized one level deeper. While this granularity increases mapping flexibility, it causes many trivial blocks as the model contains many blocks with trivial computations (see Figure 3.10b). The sub-blocks of PreProc and MatCon only minimally contribute to the total computation with less than 1K operations, such as SpIY, SpICb, and SpICr, which are not good candidates for heterogeneous mapping.

The performance estimations indicate that only computationally intensive blocks that are suitable for mapping need to be further decomposed if possible. For instance, the result of Figure 3.10b guides the designer to break SubBlk into several smaller blocks to increase mapping alternatives. Meanwhile, it is important to not alter other computationally non-demanding blocks, which otherwise would increase DSE complexity and extra communication overheads. Therefore, the designer can perform a customized depth synthesis of the model in the final design iteration where blocks are decomposed unevenly depending on their computation demands. For instance, SubBlk is explored to a deeper level while other blocks at the same depth (e.g. PreProc and MatCon) are left untouched. The demand analysis of the final iteration with varied depths is presented in Figure 3.10c.

At the same time, the final iteration exposes structural parallelism as Tran1, Tran2 and Tran3 have no dependency over each other. As shown in the early estimation (Figure 3.10c), Tran1 has significant computations (20K million operations) and mild communications (140KB), making it a candidate for concurrent execution.
(a) Initial Iteration: Depth 2

(b) Intermediate Iteration: Depth 3

(c) Final Iteration: Customized Depth

Figure 3.10: Early Estimation and Exploration of MJPEG Encoder
CHAPTER 3. SPECIFICATION SYNTHESIS

<table>
<thead>
<tr>
<th>Explorations</th>
<th>Components</th>
<th>Modules in HW1</th>
<th>Modules in HW2</th>
<th>TLM</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expl1</td>
<td>ARM7TDMI (50MHz)</td>
<td>none</td>
<td>none</td>
<td>81.2s</td>
<td>-</td>
</tr>
<tr>
<td>Expl2</td>
<td>ARM7TDMI (50MHz)</td>
<td>BlkProc</td>
<td>none</td>
<td>63.3s</td>
<td>-</td>
</tr>
<tr>
<td>Expl3</td>
<td>ARM7TDMI (50MHz)</td>
<td>SubBlk</td>
<td>none</td>
<td>62.7s</td>
<td>-</td>
</tr>
<tr>
<td>Expl4</td>
<td>ARM7TDMI (50MHz)</td>
<td>SubBlk</td>
<td>none</td>
<td>69.2s</td>
<td>-</td>
</tr>
<tr>
<td>Expl5</td>
<td>ARM7TDMI (50MHz)</td>
<td>PreProc Tran1</td>
<td>none</td>
<td>39.0s</td>
<td>OK</td>
</tr>
<tr>
<td>Expl6</td>
<td>ARM7TDMI (50MHz)</td>
<td>PreProc Tran1</td>
<td>Tran2</td>
<td>59.8s</td>
<td>OK</td>
</tr>
<tr>
<td>Expl7</td>
<td>ARM7TDMI (50MHz)</td>
<td>PreProc Tran1</td>
<td>Tran2</td>
<td>58.7s</td>
<td>OK</td>
</tr>
<tr>
<td>Expl8</td>
<td>ARM7TDMI (50MHz)</td>
<td>PreProc Tran1</td>
<td>Tran2 Mot</td>
<td>60.8s</td>
<td>-</td>
</tr>
</tbody>
</table>

(a)

Figure 3.11: DSE of MJPEG Encoder
CHAPTER 3. SPECIFICATION SYNTHESIS

The designer can perform the iteration process at arbitrary depth of the model until the model exploration produces a specification matching the underlying heterogeneous platform. The next section illustrates the execution results of these explorations of the MJPEG encoder example.

3.4.1.2 Design Space Exploration via Virtual Platforms

To demonstrate the benefits of DSE, this section reports the performance of the target platform as measured from the generated TLM. The timed TLM exposes PE mapping and scheduling on each PE, communication and synchronization across multiple PEs, and bus communication at the transaction level.

For simplicity, we limit our target architecture to a three PE template with one CPU: ARM7TDMI and two customized hardware (HWs), connected through an AMBA AHB bus. Our explorations over the platform are listed in Figure 3.11. Expl1 is a pure software solution for comparison in which all elements are mapped on the CPU. Expl2 is corresponding to the initial iteration discussed previously in Section 3.4.1.1 where only BlkProc can be mapped. Expl3 to Expl4 map the specification used in the intermediate iteration to several combinations of CPU and one HW. Likewise, the specification generated in the final iteration with customized granularity is used in Expl5 to Expl8 experimenting on all three PEs.

With the coarsest granularity, Expl1 can only use a single PE (CPU) as a target mapping. The execution time of 81.2s (see Figure 3.11) is far beyond the real-time requirement of 60.66s. The finer granularity of the specification from Expl2 to Expl4 allows more mapping flexibility. By mapping the computational heavy blocks, such as BlkProc, SubBlk or Chroma, onto HW1, it achieves better results up to 62.7s. Although it yields a nearly 25% speedup by combining hardware components on the platform, it still cannot meet the requirement of 60.66s.

The explorations based on the final iteration utilize the further decomposed blocks to seek the satisfactory performance. It also exposes the parallelism between Tran1 and Tran2 and Tran3. Splitting these behaviors enables more mapping opportunities. Figure 3.11b plots the execution time and PE utilizations from Expl5 to Expl8 with Expl1 as baseline. Expl5, Expl6 and Expl7 all satisfy the real-time requirement with runtime of 39.0s, 59.8s, and 58.7s, respectively. Nevertheless, mapping Tran2 to another HW (HW2) (Expl6-8) does not boost the performance meaningfully due to fairly large communication overhead. Although HW2 is load balanced in Expl8 by executing Mot, the overall performance does not increase much due to the additional traffic (420KB) between PEs. Subsequently, Expl8 fails to meet the deadline.
CHAPTER 3. SPECIFICATION SYNTHESIS

Expl5 achieves the best performance (39.0s) while others only yield a limited performance improvement over the pure software solution by Expl1. By splitting the two computational heavy blocks PreProc and Mot between CPU and HW1 without much traffic overhead, Expl5 keeps Tran1 (on HW1) running in parallel to Tran2 and Tran3 (on CPU). Moreover, it requires only one HW PE, consuming less computing resources than Expl6 and Expl7. Subsequently, Expl5 is chosen for the final implementation.

We have shown that the established flow can assist designers in early modeling, performance estimation and DSE with heterogeneous elements. The wealth of system-level evaluation and assessment toolchains are now made available to higher abstractions in Simulink. This makes it much easier for designers to move across abstraction boundaries and to select model-to-platform mappings with highly shortened concept-to-silicon time.

3.4.2 Productivity Gain

To measure the automation benefits of Algo2Spec, we generate system-level specifications of all examples: Corner Detection, MJPEG Encoder and MJPEG Decoder. These models stem from Simulink example library, exhibiting medium to high complexity in terms of blocks (57-315 blocks).

Table 3.2 summarizes the number of blocks in these input models, and the number of granularity tuning iterations required to reach the final SpecC specification (the similar procedure described in Section 3.4.1.1). The same target platform template from Section 3.4.1.2 is used here (one ARM7TDMI and two HW PEs). For the final specifications in each application, we measured the Lines of Code (LOC) of the SEC generated application C code (LOC of C), together with the total LOC of the synthesized specification (LOC of SpecC). The table also shows the synthesis time averaged over the total design iterations running on a CentOS 6.5 x86 Linux desktop of Intel i5 (Sandy Bridge), quad-core at 2.8GHz and 4GB memory.

To facilitate the productivity gain discussion, we consider the LOC that a designer would need to manually modify when transitioning from Simulink to SpecC. For this, we only count the changed lines between SEC generated code and SpecC specifications as SEC generates the application C code automatically. Furthermore, changing granularity level requires to manually compose design hierarchy and scheduling. In Table 3.2, only the LOC of the final design iteration in each example exploration is listed. However, as multiple iterations are necessary to determine the eventual specification granularity, the benefit of automatic synthesis multiplies throughout the total explorations. To normalize across the differently sized applications and as an application
Table 3.2: Design Characteristic and Productivity Gain

<table>
<thead>
<tr>
<th>Properties</th>
<th>Corner Detection</th>
<th>MJPEG Encoder</th>
<th>MJPEG CODEC</th>
<th>Per Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulink Blocks</td>
<td>57</td>
<td>140</td>
<td>315</td>
<td>1</td>
</tr>
<tr>
<td>Model Iterations</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>SpecC Behaviors</td>
<td>32</td>
<td>65</td>
<td>71</td>
<td>1</td>
</tr>
<tr>
<td>LOC of C</td>
<td>1.5K</td>
<td>7K</td>
<td>8K</td>
<td>88 (avg)</td>
</tr>
<tr>
<td>LOC of SpecC</td>
<td>1.8K</td>
<td>8K</td>
<td>9.3K</td>
<td>103 (avg)</td>
</tr>
<tr>
<td>Changed LOC</td>
<td>0.7K</td>
<td>3.1K</td>
<td>3.3K</td>
<td>38 (avg)</td>
</tr>
<tr>
<td>Estimated Manual Time</td>
<td>46 hrs</td>
<td>206 hrs</td>
<td>220 hrs</td>
<td>2.53 hrs</td>
</tr>
<tr>
<td>Avg. SEC Time</td>
<td>120.5s</td>
<td>178.1s</td>
<td>196.6s</td>
<td>3.08s</td>
</tr>
<tr>
<td>Avg. Algo2Spec Time</td>
<td>2.02s</td>
<td>3.04</td>
<td>3.27s</td>
<td>50ms</td>
</tr>
<tr>
<td>Avg. Synthesis Time (Total)</td>
<td>122.52s</td>
<td>181.14s</td>
<td>199.87s</td>
<td>3.13s</td>
</tr>
<tr>
<td>Productivity Gain</td>
<td>1351</td>
<td>4094</td>
<td>3962</td>
<td>2909</td>
</tr>
</tbody>
</table>

independent indicator, the last column indicates the average results per generated block across the investigated applications.

To estimate the manual development time, we use development performance reported in previously published works [121, 65, 17]. They reported 15 lines of correct code per hour [121], 3-30 LOC/hour (from concurrent to serial programming [65]), and 14-20 LOC/hour (statistics from software projects [17]). For the purpose of this article, we assume 15 LOC/hour to approximate the manual development time.

To obtain the SLDL specification, 0.7K to 3.3K lines needed to be changed manually from the SEC generated code (see Table 3.2). This translates to 46 to 220 hours of coding effort. Conversely, using Algo2Spec this process only takes between 123 to 200 seconds depending on the number of blocks and their complexity (e.g. complex kernel blocks or simple operations). Note that the synthesis time includes the runtime of the SEC code generation (120.5s to 196.6s). The runtime of Algo2Spec is orders of magnitude faster (2s-3.3s).

Based on the estimated manual coding times and the measured tool running time, Algo2Spec synthesizes the system-level specification 1351, 4094, and 3962 times faster for each evaluated application, respectively. The productivity gain is higher for models containing more blocks. Overall, Algo2Spec yields three orders of magnitude speedup in productivity over manually re-authoring the specification.

To compare across examples, we consider the average LOC to be changed to wrap the SEC generated code of a single Simulink block into SpecC. Shown in the last column of Table 3.2.
CHAPTER 3. SPECIFICATION SYNTHESIS

on average 38 lines need to be altered for one block. This translates to an estimated 2.53 hours of manual development. In comparison, Algo2Spec synthesizes a block in 3.13s on average including 3.08s for SEC and 50ms for Algo2Spec itself.

We acknowledge the difficulty of quantifying productivity gain as it involves manual work depending on factors such as designer’s experience, tools and familiarity to the design [21]. However, as the automatic synthesis by Algo2Spec takes a few seconds for each block versus hundreds of hours in manual transformation, the productivity gain would not be affected significantly even by more precise measured manual time or experienced designers.

3.5 Summary

In this chapter, we identified the specification gap between algorithm-level and system-level designs. We proposed Algo2Spec to automatically synthesize Simulink algorithm models into SLDL specifications to close the gap. With the automatically generated system-level specification, early estimation profiling, rapid platform re-targeting and heterogeneous VP explorations are available to vastly enrich the algorithm designer’s decisions. Algo2Spec creates a new joint co-design flow seamlessly expanding system-level tool-chains and refinement steps to algorithm designs in Simulink.

We have evaluated the efficiency of our approach by exploring several real-world examples. The joint co-design flow offers tremendous opportunities for algorithm designers to explore the platform suitability of algorithms and to tune algorithms to better match platform requirements (e.g. in terms of parallelism). The new methodology avoids the tedious and error-prone process of manual effort. Algo2Spec synthesizes a single block in 3.13s on average, compared to an estimated manual re-authoring in 2.53 hours. With this, Algo2Spec offers three orders of magnitude gains in design productivity for generating the system-level specification.

Furthermore, the presented synthesis framework builds the foundation for optimizations of block synthesis and simplifies the exploration of parallelism in the future.
Chapter 4

Specification Granularity

With the automatic generation of specifications by Algo2Spec in place, new optimization and synthesis opportunities emerge. In this chapter, we explore the new synthesis options. In particular, we look into Specification Granularity and examine how it impacts the DSE complexity and specification performance. This chapter starts with introduction and problem definition, followed with the proposed heuristic solutions. Lastly, we discuss the results.

4.1 Introduction

Design Space Exploration (DSE) is an instrumental ingredient in System-Level Design to identify suitable execution platforms with design configurations including PE allocations, mapping, scheduling and etc. Finding an optimal solution (i.e. performance, area, or cost) using heterogeneous platforms is still an NP-Complete problem in today’s codesign research [51, 55, 107]. The ability of generating specification automatically by Algo2Spec brings new aspects to the DSE complexity issue. The DSE complexity of a synthesized specification can be influenced by its granularity, in this context defined as how many blocks (from Simulink) are retained in the specification.

The number of mapping alternatives grows in combinatorial exponential order with the amount of blocks (assuming constant number of PEs). Previously shown in Figure 1.2, although the high degree of mapping alternatives offers high flexibility, such flexibility is only beneficial in a platform with more independent components, which is PEs in our context. In typical hardware-software co-design area, the amount of PEs are fairly small, within tens of elements. Thus, a large space of the flexibility is in fact overhead, which unnecessarily overburdens the DSE. Conversely,
Chapter 4. Specification Granularity

Flexibility vs. Efficiency Trade-off

A certain flexibility must be retained in order to map blocks to different PEs for the purpose of codesign.

Meanwhile, Algo2Spec offers optimization opportunities using block fusion techniques provided by Simulink Embedded Coder (SEC) [86] as a higher granularity of specification (fewer blocks) may output more efficient code due to block grouping. Nevertheless, synthesizing into a single block (zero flexibility) is DSE design-prohibitive as it removes any Hardware/Software Codesign opportunities although it may produces the most efficient specification.

Selecting granularity has two extremes: (a) the finest and (b) the coarsest granularity. The finest granularity (a) does not merge any blocks. Thus the generated SLDL specification retains all blocks of the Simulink model (1-to-1 block mapping). Conversely, the coarsest granularity merges all blocks into a single SLDL module, capturing the entire Simulink model’s functionality. Both extremes yield the same functionality through specification synthesis internally using SEC but differ significantly in computation demand. We observed a computation reduction between 5% and 12%[1] between the extremes.

Figure 4.1 visualizes the trade-off between the number of blocks and efficiency. As we have discussed, the number of blocks inversely affects mapping flexibility as the coarsest offers no exploration freedom while the finest drowning the DSE with too many options. The challenge is to identify a suitable granularity for the synthesized specification with a balanced the trade-off.

---

1Optimization through block fusion depends on the application and the Simulink Embedded Coder optimization capabilities. With continuous advances in compiler techniques, the maximal optimization is likely to increase in the future releases.
CHAPTER 4. SPECIFICATION GRANULARITY

to reduce the DSE complexity (fewer mapping alternatives) without jeopardizing the performance
(higher efficiency).

We introduce two design phases observation for the synthesis. One is DSE-Phase, is a
region where synthesized specification is platform-agnostic, meaning the granularity choice is inde-
pendent from the choice of the underlying architecture. The goal is to produce a specification with
reduced amount of blocks (yet not zero) to simplify the processing of identifying a suitable platform
for the specification by pruning off some DSE space. Therefore, the overwhelming flexibility (blue
in Figure 4.1) should be avoided.

The second phase, Synthesis-Phase, is defined as platform-specific specification synthe-
sis. After settling on allocation and mapping in the DSE phase, in the Synthesis-Phase any re-
mainig flexibility becomes overhead since no more exploration is necessary. Thus, matching the
mapping decision with the particular granularity choice is ideal (i.e. creating the exact amount of
blocks for a certain mapping decision). It maximizes the block-merging optimization potential for
the given mapping.

In this chapter, we first analyze the granularity trade-off for a set of media applications.
Then, we propose an automatic approach that, as part of the Algo2Spec tool-chain, selects certain
algorithm blocks for fusion and thus tunes the effective granularity of the specification.

Our automated granularity tuning approach, Demand-/Mapping-aware Granularity Tun-
ing (DMGT), bases the merging decision on computational/communication demands of algorithm
blocks, and mapping decisions of the specification to the underlying platform to navigate the trade-
off curve. DMGT employs granularity tuning heuristics to help navigate the DSE phase. DMGT-
DSE controls granularity through both vertical (inter-hierarchy) merging and horizontal (intra-
hierarchy) block clustering. DMGT-DSE identifies the set of meaningful blocks based on computa-
tion, substantially reducing DSE complexity to reach the orange area in Figure 4.1. After mapping
decision is made, in the synthesis phase, DMGT-Synthesis generates a mapping-aware specification
with exactly those blocks matching the selected mapping. By this, it maximizes the possible
efficiency for a particular mapping (green area in Figure 4.1).

We demonstrate the benefits through industrial strength multimedia applications. DMGT
synthesized specifications achieve up to 84% of the maximally achievable optimization (see more
detailed definition in Chapter 4.4) in the DSE-Phase with 86% block reduction. Synthesis-Phase
generates specifications with up to 96% achievable optimization with 95% block reduction and
up to 50% traffic reduction. This chapter is structured as following: Section 4.2 examines the
background of specification characteristics. Section 4.3 discusses the granularity tuning. Section 4.4
CHAPTER 4. SPECIFICATION GRANULARITY

demonstrates the benefits on experimental studies. Section 4.5 summarizes the chapter.

4.2 Background: Specification Performance Analysis

This section first analyzes the specification performance of a video MJPEG Encoder to illustrate the background and the motivation. It then introduces a metric to formalize metrics of our tuning approach.

4.2.1 Demand Analysis of MJPEG Encoder

To highlight the distribution of computation and communication, we chose a MJPEG Encoder which directly stems from the Simulink demo library (R2012a). The model includes 164 blocks in total, with 94 leaf blocks (non-dividable) and 70 hierarchical blocks. It has up to 12 levels of block hierarchy. Figure 4.2 analyzes model performance at the leaf level. We characterize blocks as follows:

**Definition 1 BlockSize** is the computation demand (number of C operations) of a block during entire execution.

**Definition 2 BlockTraffic** is amount of data transferred (Bytes) on all input and output ports of the block.

Figure 4.2 shows the histogram of leaf block characteristics with blocks sorted into bins by BlockSize on the logarithmic x-axis. For each BlockSize bin, it shows the relative number of blocks in yellow (left), the relative contribution to total computation in blue (middle) and the relative contribution to total traffic in red (right).

Figure 4.2 shows that more than 83% of all leaf blocks have BlockSize less than 1 Million operations performing only simple and individual operations. While largely contributing to the total number of blocks, these do not contribute to the total computational cost. Only a few blocks (less than 17%) are of 1 Million operations or larger in size. In our model, these blocks include kernel computations such as template matching and DCT (Discrete Cosine Transform) for processing arrays of pixels. Often, computationally intense blocks are predefined library blocks in Simulink with un-explorable internal structures as encrypted MATLAB code or binary distributions. In this model, essentially only the large blocks contribute to the overall computational demands.

\[ \text{We assume an x86 host architecture for traffic word size.} \]
CHAPTER 4. SPECIFICATION GRANULARITY

The Pareto principle (i.e. 80-20 rule) is clearly observable in the MJPEG Encoder analysis; where about the 20% of blocks contribute to 80% of the total computational load. At the same time we observe a strong drop in contribution to the total computation as the blocks get smaller in \( \text{BlockSize} \). We therefore chose \( \text{BlockSize} \) as a metric to decide which blocks should be merged together during SLDL specification synthesis. Merging blocks controls the SLDL specification granularity and leverages the inter-block code optimization potential for synthesizing SLDL leaf blocks. This will result in a reduced overall computational demands of the generated SLDL specification while still offering sufficient number of blocks for heterogeneous DSE at system-level. At the same time this exposes only blocks with meaningful computational contributions, simplifying the DSE by reducing the total number of blocks to be mapped.

Moreover, Figure 4.2 highlights that 40% of blocks have only 100s of operations almost not contribution to total computation. They are clear merging candidates. Starting with blocks of 1M size, the computation contribution becomes significant – even though only 10% of the total number of blocks are in that category. Given this distribution, block blocks smaller than 100K operations should be merged.

Looking at traffic, blocks of 10K size contribute to almost 30% of traffic (but without significant computation). Merging these blocks to their most communicated to neighbors will significantly reduce traffic. Conversely, the traffic of blocks of 1M size cannot be much optimized, unless a small neighbor can be merged into a large block.\(^3\) Overall, large optimization potential exists and heuristics have to be identified to utilize it.

\(^3\) The block demand measurement is gathered through dynamic profiling and we consider the operations as a heaviness factor. However, the concept can be extended easily to architectural-correlated values.


4.2.2 Block Demand Matrix

Each block can be characterized by its computation and communication demands. We define a block demand matrix to categorize blocks with respect to the orthogonal measures of Block-Size and BlockTraffic as shown in Figure 4.3.

The computation lower bound threshold \( P \) vertically splits the block demand space into light-weight and heavy-weight blocks. The computation upper bound \( U \) identifies super heavy blocks (region 5). The region bound by \( P \) and \( U \) contains blocks with desirable computation (\( P < \text{BlockSize} < U \)). They express meaningful impact in DSE to be mapped on Processing Elements (PE). In contrast, light-weight blocks are too little in computation to offer meaningful alternatives. Meanwhile, too heavy blocks will impede the DSE as well because computation intensive applications need to split across multiple PEs as a single embedded processor or hardware (HW) cannot handle in order to meet budget constraints such as deadline or power efficiency.

On the y-axis, the BlockTraffic identifies how much a block communicates. We define “loud” and “quiet” with much, or very little traffic. The threshold is a percentile (determinable by the user) to the total traffic in the model. With these defined regions in place, the blocks in region 4 are most suitable for DSE. These blocks have a good balance on BlockSize and small TrafficSize, which are ideal choices for DSE mapping alternatives. The DMGT in DSE-Phase leverages these two tunable thresholds \( P \) and \( U \) in order to merge and cluster the blocks in the specification into region 4 as much as possible, meanwhile, avoiding the super heavy blocks.
4.3 Granularity Tuning Approach

With the large number of blocks and high model complexities, automated methods are required for identifying algorithm blocks to be merged in the specification synthesis. Motivated by the strong correlation of BlockSize to the total computational demands, we propose the Demand-Mapping-aware Granularity Tuning (DMGT) approach. This section first overviews the specification tuning/synthesis flow and then introduces the algorithms in detail.

4.3.1 Specification Granularity Tuning Flow

Figure 4.4 illustrates the overall flow. The input to the flow is an algorithm model captured in Simulink. The algorithm model is dynamically profiled to retrieve the computational and communication demands (BlockSize, BlockTraffic). It employs a run-time profiling method to gather the demand information, which therefore avoids the more pessimistic Worst-Case Execution/Communication Time (WCET/WCCT). DMGT utilizes the demands of each block in the model to determine the leaves for the SLDL specification. Finally, Specification Synthesis module (see Chapter 3) realizes the granularity decision and generates accordingly the SLDL specification as an input to the system-level design space exploration.

Profiling: In order to gather the demand information of the model, profiling is used as the first step of estimating the algorithm model characteristics. Both the number of operations (e.g. add, subtract, multiply, divide) and the volume of data transfers (e.g. in, out, inout) are estimated for computation and traffic demands respectively as a target-independent metric for measuring the
CHAPTER 4. SPECIFICATION GRANULARITY

We deployed for the scope of this work a profiling phase using SCPROF [15] that instruments at the basic block level and gathers demand information through simulation and analysis process. For this, the algorithm model is synthesized into the SLDL specification at the original granularity and analyzed through dynamic profiling to avoid overhead in static analysis. Other methods for determining computation demand, such as static analysis, are feasible, however, do not change our overall proposed approach.

**Specification Generation:** Given the selection of leaves identified by DMGT, the specification synthesis generates the SLDL specification converting the decision into reality. Each designated leaf is converted into ANSI-C code using the Simulink Embedded Coder (SEC). For this, inter-block optimizations are enabled, reducing the overall computation demand if a designated leaf contains multiple blocks. In addition, Specification Synthesis transposes the generated leaf ANSI-C code into SLDL and generates SLDL hierarchical blocks according to the algorithm specification. The resulting SLDL specification is then input to a top-down system-level design space exploration flow evaluating heterogeneous platform and specification mapping alternatives.

**DMGT: DSE Phase:** Blocks in the algorithm model are distinguished between leaf and hierarchical blocks. DMGT-DSE identifies the suitable leaves (i.e. selecting hierarchical blocks to be leaf blocks) using gathered demands. DMGT-DSE incorporates two major routines to leverage the demands information.

- **Vertical merging.** It base the merging on computation utilizing only BlockSize. The approach compares the BlockSize against a computational lower bound threshold $P$ and an upper bound threshold $U$. If the BlockSize of a block falls between these two bounds, it will be marked as a leaf for later synthesis. Details of this heuristic are discussed in Section 4.3.2.

- **Horizontal clustering.** This scheme clusters blocks within the same hierarchical level together bounded by $P$ and $U$. This further affects the number of blocks that cannot be merged vertically ($\geq U$). This part is explained in Section 4.3.3. As its output, DMGT defines the Leaf Selection.

**DMGT: Synthesis Phase:** As the result from DSE, a set of mapping decisions for the system specification is deducted. The mapping decide thereby feeds back to the DMGT to trigger the plat-
form optimization stage. DMGT-Synthesis generates a mapping-aware specification that is tailored to the platform mapping choice which maximizes AFP. Details of the algorithm are discussed in Section 4.3.4. This final specification is then used for refinement and back-end synthesis tools.

4.3.2 DMGT-DSE Phase: Vertical Merging

This chapter looks into the vertical merging routine in DMGT-DSE phase. We define the algorithm as Computation-Guided Granularity Tuning (CGGT) [124]. To more formally define CGGT, we define the following notations: an algorithm model hierarchy can be viewed as a spanning tree graph \( G(v, e) \) where a vertex \( v \) is a block and tree edges \( e \) represent containment relationship. The example in Figure 4.5 shows a simple Simulink model structure with top-level block \( A \) containing \( B \) and \( C \), which in turn contain other sub-blocks (e.g. \( \text{Child}(A) \) returns \( B, C \) and \( \text{Parent}(C) \) returns \( A \)). For simplicity of explanation, we assume unique block instances; hence blocks have a single parent. Only tree edges are allowed which describe hierarchical relationships and therefore back edges, forward edges or cross edges do not exist in our formulation. Each leaf block is annotated with the BlockSize reflecting the computational demands obtained by the profiling – see Table 4.1 for our example. The BlockSize of a hierarchical block (i.e. a block with children) is defined as the sum of all its children.

To identify blocks based on the BlockSize, CGGT employs a lower bound threshold \( P \). \( P \) denotes the minimal BlockSize of a block to be considered as a leaf (i.e. BlockSize\( (v) > P \)). Otherwise, the leaf identification needs to propagate up the hierarchical tree to combine more blocks.
CHAPTER 4. SPECIFICATION GRANULARITY

Table 4.1: LeafBlockSize and Tuning Threshold

<table>
<thead>
<tr>
<th>BlockSize</th>
<th>H</th>
<th>I</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>F</th>
<th>G</th>
<th>P</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockSize</td>
<td>100</td>
<td>40</td>
<td>20</td>
<td>30</td>
<td>50</td>
<td>120</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>90</td>
<td>130</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

together to meet $P$. Meanwhile, the merging needs to be stopped with an upper bound threshold $U$ which controls the largest tolerable BlockSize of a block. Not using the threshold $U$ would lead to a single root block as all the blocks will be merged. In essence, a block is declared as a leaf, if:

$$P < \sum_{v \in \text{Child}(\text{block})} \text{BlockSize}(v) < U$$  \hspace{1cm} (4.1)

The CGGT algorithm (see Algorithm 1) builds upon the Depth-First Search (DFS) to cover all nodes in the graph (i.e. all blocks in the model) using a marker labeling approach for the leaf block selection. During the tree traversal by the algorithm it declares a node (block) as either leaf or hierarchical block by $\text{Mark}(\text{block}) \leftarrow \text{LEAF} \lor \text{HIER}$. To illustrate the operation, Figure 4.5 together with Table 4.1 shows an example. Prior to the algorithm running, all blocks in the model have no markers. CGGT thus marks all blocks that are without children (i.e. leaves in the algorithm model) as leaves (line 13) by recursively reaching to the deepest blocks. In our example, these are $H, I, K, L, M, F, G$.

Upon encountering a leaf block $v$ with $\text{BlockSize}(v) > P$, such as $H$ and $F$ (highlighted in gray), it will mark its parent as $\text{HIER}$ (line 19). As $H, F$ are leaves with sufficient computation, they label their parents as a hierarchical block ($\text{HIER}$) because CGGT works heuristically by separating blocks of big BlockSize as much as possible within a parent block. By setting the $\text{HIER}$ marker, it signals that there are heavy blocks within such block. Thus, this parent block cannot be overwritten by the $\text{LEAF}$ maker anymore to preserve the earlier identified leaf blocks ($H, F$). On the other hand, these light leaf blocks are only able to claim their parents to be a leaf unless there is no other blocks have marked it as $\text{HIER}$. The $\text{LEAF}$ label of a block can be overwritten by other sibling blocks if any has a $\text{BlockSize} > P$ (line 14 - line 20).

Nevertheless, if none of the block’s children has BlockSize larger than $P$, the block (with clean makers) will be marked as a $\text{LEAF}$ (line 16). Essentially, $v$ propagates the leaf to one level above based on computational compositions. The BlockSize of the recent marked leaf block is again examined to decide if it needs to be further aggregated up in the MERGE procedure (line 1). In our example shown in Figure 4.5b, $K, L, M$ are all pruned off by marking their parents.
CHAPTER 4. SPECIFICATION GRANULARITY

to be a leaf, yet their parent \( J \) satisfies the threshold \( P \) (i.e. \( \text{BlockSize}(K) + \text{BlockSize}(L) + \text{BlockSize}(M) > P \)), which makes it a leaf that should be preserved in the hierarchy.

Once a hierarchical block is reached in the tree traversal, \( MERGE \) procedure (line 2 - line 9) examines the \( \text{BlockSize} \) of it against \( U \) to check if the block can be still considered as a leaf to further prune off undesired computational demands. The input block to \( MERGE \) has a marker of either \( \text{LEAF} \) or \( \text{HIER} \) labeled by its child blocks. If the \( \text{BlockSize} \) of this hierarchical block is less than \( U \), it will be marked as a leaf and propagate up the leaf marker. Otherwise, it remains as \( \text{HIER} \) and sets its own parent to be \( \text{HIER} \). In our example, \( F, G \) are merged to their parents because the \( \text{BlockSize} \) of \( C \), which is the combined size of \( F \) and \( G \), is less than \( U \), and similarly \( J \) is merged because it is the only block, which apparently satisfies the requirement of \( U \).

The CGGT algorithm recursively walks through the entire model tree. Once the root node is visited, the leaf identification process completes. Figure 4.5c shows the output specification tree. Most of the trivial blocks (\( K, L, M, G \)) are pruned off as their \( \text{BlockSize} \) is smaller than \( P \). The merging threshold \( U \) contributes further to trim off combinations of heavy and small blocks. \( J \) and \( F \) are merged to a level up without impeding their parent \( \text{BlockSize} \) too much. However, \( I \) remains in the final specification as the \( \text{BlockSize} \) of its parent block \( D \) is above the merging threshold (i.e. \( U < \text{BlockSize}(D) = \text{BlockSize}(H) + \text{BlockSize}(I) \)). Therefore, it stays as a leaf. This kind of leaf blocks are considered as \textit{forced leaf} blocks. The number of forced leaf blocks is an important metric in order to evaluate the efficiency and performance of the tuning parameter selection. The goal is to minimize the number of forced leaves since they have small computation, thus are not desirable for later DSE.

4.3.3 DMGT-DSE Phase: Horizontal Clustering

DMGT extends the CGGT algorithm to determine granularity in DSE-Phase. To ease the explanation of DMGT, we first intuitively overview CGGT. In a nutshell, CGGT performs vertical merging through the hierarchical tree following a bottom-up DFS search. Consider an example in Figure 3.6. The upper part shows a partial input model tree of a Simulink example, where ”M” is the hierarchical block containing a number of sub-blocks. All children of ”S” are light-weight (below \( P \)), therefore they are merged to the parent (”S”). Leaf selection is propagated up vertically. In this example, ”N”, ”Q” and ”S” (in gray) are heavy-weight (above \( P \)). However, as their sum exceeds the upper bound threshold \( (U) \), would otherwise create a super heavy block, they (and the other siblings) are not merged into ”M”. In result, the small siblings (here ”O”, ”P”, ”R”, ”T”) remain
Algorithm 1 Computation-Guided Granularity Tuning

1: procedure MERGE(block, U)
2:   if BlockSize(block) < U then
3:     Mark(block) ← LEAF
4:     if Mark(Parent(block)) ≠ HIER then
5:       Mark(Parent(block)) ← LEAF
6:     end if
7:   else
8:     Mark(Parent(block)) ← HIER
9:   end if
10: end procedure

11: procedure CGGT(block, P, U)
12:   if Child(block) = ∅ then
13:     Mark(block) ← LEAF
14:     if BlockSize(block) < P then
15:       if Mark(Parent(block)) ≠ HIER then
16:         Mark(Parent(block)) ← LEAF
17:     end if
18:   else
19:     Mark(Parent(block)) ← HIER
20:   end if
21:   else
22:     for all v ∈ Child(block) do
23:       CGGT(v, P, U)
24:     end for
25:     MERGE(block, U)
26: end if
27: end procedure
 Despite there are many small sibling blocks to these heavy leaves, the CGGT algorithm disallows to merge them to ”M” because that would create a leaf too big exceeding the .

DMGT following the usage of the threshold $P$ and $U$ for leaf identification. Controlled by $P, U$, block distribution can vary from one region to another, mostly to region 4, namely quiet, heavy-weight blocks. DMGT operates horizontally for all sub-blocks in a hierarchical block. It is implemented as a merging step extension in CGGT.

The lower part of Figure 4.6 shows the processed tree produced by CGGT (also see the later more detailed Figure 4.7a which annotates BlockSize in blue). As a result, $N, Q$ and $S$ are marked as real leaf blocks (colored in gray). Their BlockSize meets the tuning threshold ($P = 90, U = 130$) and the sub-tree of $Q$ and $S$ are pruned due to too little computation. $M$ remains as the hierarchical block as it would turn otherwise into a super heavy block (exceeding $U$). Hence, the light-weights $O, P, R$ and $T$ are forced leaves. In the following discussion, we will focus on the horizontal relationship among the sub-blocks of $M$ shown in Figure 4.7a.

A direct graph notation $G(v, e)$ is used to describe the relationship among sub-blocks under a hierarchical block with communication dependency and computation/traffic volumes. Each vertex ($v$) in the graph is a block and each edge ($e$) is the communication link with the traffic as weight between two blocks. Figure 4.7a continues the example with the communication links shown together with leaf blocks (both forced and real).

Upon visiting a parent block, DMGT routine is then triggered. Algorithm 2 outlines the DSE-Phase algorithm. The inputs are a hierarchical block and the upper bound threshold $U$. 

Figure 4.6: Example Model Tree Input
The algorithm first populates the LinkRank table containing all the links in the block sorted in descending order (line 10). The result of LinkRank is a list of links sorted by BlockTraffic also capturing source (src) and sink (sink) of each link. The algorithm starts from the heaviest link (l). It examines source (Src(l)) and sink (Sink(l)) blocks of l and ensures that both are leaves (line 14). If merging two blocks would not exceed the upper bound, a new leaf block is created (line 15). Lastly, the graph is updated to reflect the new connections and update the traffic weight (line 18) in LinkRank. For brevity, we omit the details of UPDATE-GRAPH which is standard.
CHAPTER 4. SPECIFICATION GRANULARITY

Table 4.2: Example Model Demands

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>O</th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>P</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockSize</td>
<td>100</td>
<td>40</td>
<td>10</td>
<td>135</td>
<td>10</td>
<td>120</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>90</td>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.7 briefly illustrates the working principles clustering small blocks ("O", "P", "R", "T") to their heavy siblings ("N", "Q", "S") to form new heavy blocks with reduced traffic (from 350 to 115). It shows how blocks are merged to create quite, heavy-weights (region 4 of Figure 4.3).
CHAPTER 4. SPECIFICATION GRANULARITY

For the discussion, assume $\text{TrafficSize} > 100$ as loud. Initially, "P" and "T" (in Figure 4.7a) are loud, light-weights due to small $\text{BlockSize}$ (10, 5) with yet large $\text{BlockTraffic}$ (130, 105). Similarly, "O", "R" are in region 3 and "N", "Q", "S" are in region 2. The heaviest link, $N, P$, is merged creating new block "NP" (see Figure 4.7b). "Q" and "S" cannot be merged together due to a combined $\text{BlockSize}$ exceeding $U$ although the link $Q, S$ is heavy. "S" and "T" are merged to eliminate the traffic in between (Figure 4.7c). The overall result of the algorithm is show in Figure 4.7d. Only three blocks remain: "QR", "ST" (in region 2) and "NPO" (in region 1). Due to the nature of the block, the traffic between $Q$ and $S$ cannot be reduced as they are both computational heavy blocks. However, by clustering with their small neighbors, their $\text{BlockTraffic}$ decreased from 115 and 200 to 113 and 105, respectively.

With the heuristic approach, DSE-Phase algorithm is sufficient to reduce the overall model traffic by effectively suppressing blocks that are computational trivial, which is beneficial to reduce DSE complexity. The merging further enables the opportunities for leveraging the cross-block optimizations that can ultimately boost the model efficiency.

DMGT addresses merging as a clustering problem. Graph partitioning and clustering is an intensively studied topic [77], with algorithms such as K-way partitioning [20], or Kernighan-Lin [70, 13]. DMGT avoids the left-over light-weight siblings of CGGT by horizontal block clustering (across siblings). The clustering is controlled by a modified BlockSize-centric node collapsing algorithm based on link ranks similar to [13]. However, unlike other approaches which minimize the overall cross edge sum, DMGT is controlled by BlockSize to collapse nodes together bounded by lower bound ($P$) and upper bound ($U$). As DMGT is a heuristic giving priority to BlockSize rather than solely minimizing the traffic, it reaches a status in which small blocks are fused away. Compared to Kernighan-Lin, our algorithm has a liner complexity ($O|E|$) where $E$ is edges. Moreover, Simulink graphs have a strict semantic over algebraic loops, and current partitioning algorithms do not in particular consider such cases. Our algorithm avoids algebraic loops during the clustering by checking that source and sink are placed on the same direct feedthrough path [112]. The merging strictly complies with the Simulink simulation semantics, which is a serialized composition [112] to ensure a validated numerical results. Meanwhile, cyclic edges are allowed only with a consistent Simulink semantic checking avoiding algebraic loop [112].
CHAPTER 4. SPECIFICATION GRANULARITY

4.3.4 DMGT-Synthesis Phase: Mapping-aware Synthesis

Using the output of DMGT-DSE, designers can perform design space exploration in a System-level Design Environment (SLDE). Once designers reached a suitable mapping decision, synthesizing a new specification where granularity exactly matches the mapping decisions further increases performance. We propose the DMGT-Synthesis to select leaf blocks to enforce the user mapping decision while optimizing the efficiency of the overall specification for the chosen platform.

Algorithm 3 DMGT: Synthesis-Phase

```plaintext
1: procedure PE-MERGE(b)
2:   if \( \forall x, y \in PE(b), x = y \) then
3:     Mark(Parent(b)) ← LEAF
4:   else
5:     Mark(Parent(b)) ← HIE
6:     Mark(b) ← HIE
7:   end if
8:   \( \forall i, j \in Child(b) \), merge i, j if \( PE(i) = PE(j) \)
9:   PE(Parent(b)) ← PE(Parent(b)) ∪ PE(b)
10: end procedure

11: procedure DMGT-SYNTHESIS(b)
12:   if Mark(b) = LEAF then
13:     Mark(Parent(b)) ← LEAF
14:     PE(Parent(b)) ← PE(Parent(b)) ∪ PE(b)
15:   else
16:     for all \( v \in Child(b) \) do
17:       DMGT-Synthesis(v)
18:     end for
19:     PE-MERGE(b)
20: end if
21: end procedure
```

The DMGT Synthesis-Phase algorithm contains two parts: Tree coloring based on Processing Element (PE) selection as the prerequisite and the Synthesis-Phase itself to produce the mapping-aware specification. First, before Synthesis-Phase algorithm, a subtree coloring scheme
is used in prior to colorize all subtrees of a block when a block is chosen to be mapped to a PE. For simplicity, the detail of the coloring scheme is omitted. The Synthesis-Phase is shown in Algorithm \[3\]. The Synthesis-Phase approach realizes a multi-endpoint merging goal, where leaf blocks need to be merged to different endpoints (i.e. chosen PEs). At the core of the algorithm, it is built upon DFS search for a bottom-up block coverage. In order to overlay PE mapping decisions to the model tree, the algorithm facilitates a two dimensional block space associated with PE. I.e. Besides original block attributes (first dimension), such as \textit{LEAF} or \textit{HIER}, another attribute space, such as \textit{PE}(v), is associated with each block \(v\) in the model graph \(G\).

We define that if \(Mark(v) = \text{LEAF}\), then \(PE(v)\) is a scalar, which means a leaf block can only be assigned to a single PE. Whereas if \(Mark(v) = \text{HIER}\), then \(PE(v)\) is an array containing all assigned PEs under its subtree. The subtree coloring scheme is triggered in prior to Synthesis-Phase to reflect PE selections as colors associated with blocks. In result, each block \(v\) is annotated with a new attribute that denotes to which PE (color) it belongs and can be returned by \(PE(v)\). The algorithm recursively visits to the leaf blocks that are identified by the DSE-Phase previously and mark the parent of it to be a leaf and adding its own PE choices to the PE sets (line \([14]\)) of its parent. Once a hierarchical block is reached, PE-MERGE is called to escalate the leaf marks to one level above if all elements in its PE set belong to the same PE (line \([3]\)). Otherwise, this block must be a \textit{HIER} block as it contains blocks assigned to multiple different PEs. This causes the leaf escalating stops on this block by marking parent of it to be hierarchical (line \([5]\)). A horizontal merging is then conducted on all the sub-blocks of the block. Blocks that belong to the same PE are merged together (line \([7]\)). The PE sets are propagated up to ensure all hierarchical blocks are aware
CHAPTER 4. SPECIFICATION GRANULARITY

of the mapping decisions (line 9). The algorithm has complexity of \( O(E) \).

Figure 4.8 provides an example to explain the operation principles of the algorithm. Figure 4.8a shows the input model tree that is tuned by previous DSE-Phase. After PE mapping and analysis, the system designer selected \( A \) to run on PE1, \( C \) to run on PE2 and \( H, I \) to run on PE3 (see Figure 4.8b). After the subtree coloring scheme, all sub-blocks under each PE are marked with corresponding color (i.e. orange for PE1, yellow for PE2 and blue for PE3).

The Synthesis-Phase procedure is therefore called to output the final mapping-aware specification. Traversing down to the bottom blocks \( H, I, J \), they add their PE sets to their parent \( E \) (line 14). Once \( E \) is visited, since it contains multiple PEs, it remains a hierarchical block and merge its sub-blocks if any are mapped to the same PE, which causes \( H \) and \( I \) to be merged as \( HI \).

Likewise, all blocks below \( C \) are converged and the subtree of \( D \) is optimized away. The final specification for this example is illustrated in Figure 4.8c. Apparently, the tree depth of PE allocation affects the overall merging possibilities. The higher level blocks of the tree PEs are mapped to, the more merging opportunities exist.

With the heuristic merging for multiple endpoints, the DMGT-Synthesis offers the maximum possible vertical and horizontal merging in the model tree given a PE mapping. The resulting specification can be directly input to refinement process for TLM verification and back-end synthesis.

4.4 Experimental Results

To demonstrate the benefits of our heuristics, we have integrated them into the Specification Synthesis approach [125] and applied to several real world examples. In this section, we first outline the realization of our approach and describe the experimental setup. We then introduce our examples - three media applications: MJPEG Encoder, MJPEG CODEC and Corner Detection Chapter 3.4.

We will separate the analysis of DMGT on different phases. We first evaluate DMGT-DSE: CGGT on MJPEG Encoder in details and subsequently generalize it over all examples. Afterwards we look into DMGT-DSE: Clustering on MJPEG in details comparing against CGGT. Lastly, we analyze DMGT-Synthesis to generalize the discussion over three examples.
CHAPTER 4. SPECIFICATION GRANULARITY

Table 4.3: Benchmark Application Characteristics

<table>
<thead>
<tr>
<th>Model</th>
<th>Num. of Blocks</th>
<th>Num. of Leaves</th>
<th>Comp. of original [MOps]</th>
<th>Comp. all merged [MOps]</th>
<th>Traffic original [bytes]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>MJPEG Encoder</td>
<td>140</td>
<td>93</td>
<td>83M</td>
<td>75M</td>
<td>172M</td>
<td>10%</td>
</tr>
<tr>
<td>MJPEG Codec</td>
<td>315</td>
<td>202</td>
<td>113M</td>
<td>99M</td>
<td>312M</td>
<td>12%</td>
</tr>
<tr>
<td>Corner Detection</td>
<td>57</td>
<td>45</td>
<td>195M</td>
<td>184M</td>
<td>640M</td>
<td>5%</td>
</tr>
</tbody>
</table>

4.4.1 Experiment Setup

Table 4.3 lists the model characteristics of our three media applications. MJPEG Encoder and CODEC consist of 140 and 315 blocks respectively. Out of these, 93 and 202 blocks are leaf blocks. The Corner Detection is smaller, containing only 57 blocks, among which 45 are leaves. Many of the core blocks of the Corner Detection are described in isolated units (S-Function or library modules) that express a large computational demands. With the limited visibility, these blocks cannot be decomposed into smaller blocks. For run-time validation, MJPEG operates on a VGA-sized video stream of 900 frames, and the corner detection on 1000 VGA-sized frames.

In order to simplify the comparison across examples, we define Achieved Fusion Potential (AFP) as a relative metric. It shows how much of the best possible intra-block optimization (i.e. when all blocks are merged into one) is achieved by the model. AFP is defined as follows:

\[
AFP(\text{Current}) = \frac{\text{Comp.}(\text{Coarsest}) - \text{Comp.}(\text{Current})}{\text{Comp.}(\text{Coarsest}) - \text{Comp.}(\text{Finest})}
\]

where Finest and Coarsest denote the finest and coarsest granularity representation of the model, respectively. AFP is defined as the computation demands of the model over the computation reduction budget between the maximum (coarsest) and the minimum (finest) granularity selections. For instance, in the MJPEG Encoder the optimization potential is \(83.8M - 75.2M = 8.6M\). With \(P = 80\%\), the total computational demands is 76.7M. In absolute terms, 7.1M operations were avoided, yielding an AFP of 82\%. Table 4.4 lists the AFP in the last column. The following paragraphs analyze how much the optimization potential can be retained while merging computationally non-contributing blocks.

4.4.2 DMGT-DSE Phase: Vertical Merging

This sections closely examines the performance of CGGT. First, it investigates the impact of choosing the threshold \((P, U)\) in MJPEG encoder. Then, it generalizes the trade-off observation
CHAPTER 4. SPECIFICATION GRANULARITY

4.4.2.1 CGGT: Granularity Tuning Evaluation

As described in the CGGT algorithm, the lower bound threshold $P$ is used for leaf identification. As such, it impacts the resulting number of blocks and the inter-block optimization potential. In order to make our measurements independent of the example size, we seek an example-relative definition of $P$. For this, we propose the $BlockSize$ cumulative contribution percentile for $P$. For instance, $P = 80\%$ indicates the threshold has the value such that 80\% of all blocks in an example model has equal or smaller $BlockSize$. In the MJPEG Encoder example, $P = 80\% = 1.1MOps$ (see Figure 4.2). For simplicity, we define the upper bound merging threshold $U$ as $U = 1.1P$. The overall effect of both parameters on a resulting specification can be observed while tuning $P$ to a larger value yielding more coarse-grained model hierarchy. Figure 4.9 visualizes the results when varying the threshold $P$ from 0\% to 100\% at three intervals: 20\%, 40\% and 80\% using the MJPEG Encoder example. The plot shows how leaf blocks with increasing $BlockSize$ (X-axis) contribute to the cumulative computation (Y-axis). Note that both axes are in log scale. The cumulative computation indicates that blocks of a certain $BlockSize$ or smaller have contributed to the total computational demands. To give an example: in the original granularity (blue line), blocks with a $BlockSize$ of 100 operations or less attribute to total computational demands of 10000 operations. Given in Table 4.4, we observed that tuning $P$ from a lower to a higher threshold introduces a block removal associated with $P$. Both Figure 4.9 and Table 4.4 show that a higher $P$ produces fewer total blocks with bigger $BlockSize$ as small blocks have been merged together. Due to the model hierarchical limitations, it is not possible to remove all trivial blocks while only keeping

Figure 4.9: CGGT: MJPEG Block Reduction in all benchmark applications.
CHAPTER 4. SPECIFICATION GRANULARITY

Table 4.4: CGGT: Tuning Effect on MJPEG Encoder

<table>
<thead>
<tr>
<th>Threshold $(P)$</th>
<th>Num. of Blocks</th>
<th>Num. of leaf Blocks</th>
<th>Num. of Forced Leaves</th>
<th>Model Comp. Demands</th>
<th>Speed-up</th>
<th>AFP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P = 0$</td>
<td>140</td>
<td>93</td>
<td>0</td>
<td>83.8M</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>$P = 20%$</td>
<td>94</td>
<td>71</td>
<td>17</td>
<td>81.4M</td>
<td>2%</td>
<td>28%</td>
</tr>
<tr>
<td>$P = 40%$</td>
<td>46</td>
<td>31</td>
<td>9</td>
<td>78.6M</td>
<td>6%</td>
<td>60%</td>
</tr>
<tr>
<td>$P = 80%$</td>
<td>19</td>
<td>12</td>
<td>5</td>
<td>76.7M</td>
<td>8%</td>
<td>82%</td>
</tr>
<tr>
<td>$P = 100%$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>75.2M</td>
<td>10%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Figure 4.10: CGGT: MJPEG Granularity vs. AFP

heavy blocks. The number of forced leaves shows this effect. By increasing $P$, the forced leaves are reduced from 17 blocks to 5 blocks as some of these trivial blocks exist at a higher hierarchical level and they cannot be merged without merging other extra heavy blocks together. The most effective block reduction is obtained with $P = 80\%$, which produces a specification with 12 leaf blocks and only 5 forced leaves. The total model speedup is around 8% comparing to the original model with full block specification synthesis. As previously highlighted in Figure 4.2, blocks between 0.1M and 1M size contribute over 80% to the total computational demands. Only 20% of blocks have a $BlockSize$ larger than 1.1M, yet they constitute 80% of the total model computational cost.

Figure 4.10 illustrates the effect of increasing the threshold $P$ in finer detail. It shows the threshold $P$ on the X-axis. The left (blue) Y-axis indicates the resulting number of blocks, and the right (green) Y-axis the AFP. The MJPEG Encoder starts out with many blocks and low AFP. As $P$ increases, number of SLDL blocks decreases and AFP increases. Both correlations are almost linear.
4.4.2.2 Block Fusion Optimization and Mapping Trade-off

In order to generalize our observations, we have expanded to include all three media applications. Figure 4.11 illustrates the trade-off between number of blocks, which impact mapping flexibility, and AFP (utilizing the inter-block optimizations). In order to normalize across all applications, the y-axis shows the number of blocks relative to the maximum number (i.e. original granularity). The x-axis denotes AFP as defined earlier. Each application is shown as a line obtained by sweeping the threshold $P$ at multiple intervals.

All three models converge to higher AFP with a higher threshold $P$. The Corner Detection, with its predominantly large blocks, increases very little in AFP with small $P$ ($P < 50\%$), showing a limited block reduction. With larger $P$ ($P > 50\%$), it follows the same trend as the other examples, with linearly increasing AFP.

Figure 4.11 numerically validates the trade-off introduced in our motivating discussion (see Section 4.1). It serves a tremendous value for designers to determine the leaf selection for a synthesized specification. The CGGT approach enables designers to navigate the granularity / AFP trade-off, and to balance the number of blocks for exploration with overall performance.

4.4.3 DMGT-DSE Phase: Horizontal Clustering

To evaluate the DMGT algorithm on reducing DSE complexity, we analyze how much it can preserve the optimization potential by merging computationally non-contributing blocks on
CHAPTER 4. SPECIFICATION GRANULARITY

both vertical and horizontal span. Yet it still needs to retain a sufficient degree of flexibility. The thresholds $P$ and $U$ are essential in influencing the granularity in the model, the number of blocks and computation/traffic demands. Thus, the selection of $P$ and $U$ impacts the design flexibility as well as optimization potentials.

An application-relative definition of $P$ is necessary to make our measurements independent of the application size. For the purpose of this chapter, we define $P$ as a BlockSize cumulative contribution percentile. As an example, $P = 70\%$ will yield a threshold $BlockSize (P_{abs})$ so that the sum of computation demands of all blocks with $BlockSize \leq P_{abs}$ equals to 70\% total computation demand. For the MJPEG Encoder, $P = 70\% = 1Mops$. For simplicity, we define the merging threshold $U$ as $U = 1.1P$. Varying the threshold affects the resulting granularity (larger $P$ yielding a coarser grained model). Table 4.5 summarizes the impact and compares the computation-only CGGT with DMGT.

Figure 4.12 visualizes the demand matrix of the MJPEG application tuned by two threshold $P$ of 30\% and 70\% illustrating the block characteristic optimization. The CGGT result is plotted as a baseline to assess the DMGT algorithm’s improvement.

When $P = 30\%$, the CGGT blocks (black squares) are largely located in region 1) (55\%) and region 3) (33\%), whereas DMGT primarily migrates these blocks (blue circles) into region 3)(84\%). This is due to the traffic reduction enforced by the DMGT algorithm.

Given $P = 70\%$, blocks volume is significantly shrunk. Compare to CGGT, where 40\% blocks (cyan diamonds) are still in region 1) and 3), DMGT shifted most blocks (90\%) towards region 4) (red rings). Due to the compositional nature of the model, it is impossible to produce a model with all blocks in region 4). A few blocks (10\%) are still exist in region 2) and 3) after DMGT tuning.
CHAPTER 4. SPECIFICATION GRANULARITY

Table 4.5: DMGT Vertical Clustering: Tuning Effect on MJPEG Encoder

<table>
<thead>
<tr>
<th>Tuning Scheme (P)</th>
<th>Num. of Blocks</th>
<th>Num. of Leaves</th>
<th>Num. of Forced Leaves</th>
<th>Model Comp. Demands</th>
<th>Model Traffic Demands</th>
<th>Speed-up</th>
<th>AFP</th>
<th>Flexibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>P = 0%</td>
<td>140</td>
<td>93</td>
<td>0</td>
<td>83M</td>
<td>172M</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>P = 30% CGGT</td>
<td>89</td>
<td>68</td>
<td>15</td>
<td>80M</td>
<td>133M</td>
<td>3%</td>
<td>44%</td>
<td>63%</td>
</tr>
<tr>
<td>P = 30% DMGT</td>
<td>65</td>
<td>41</td>
<td>5</td>
<td>79M</td>
<td>106M</td>
<td>4%</td>
<td>55%</td>
<td>46%</td>
</tr>
<tr>
<td>P = 70% CGGT</td>
<td>33</td>
<td>21</td>
<td>9</td>
<td>77M</td>
<td>92M</td>
<td>7%</td>
<td>79%</td>
<td>23%</td>
</tr>
<tr>
<td>P = 70% DMGT</td>
<td>18</td>
<td>12</td>
<td>4</td>
<td>76M</td>
<td>89M</td>
<td>8%</td>
<td>84%</td>
<td>13%</td>
</tr>
<tr>
<td>P = 100%</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>75M</td>
<td>0</td>
<td>10%</td>
<td>100%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Figure 4.13: DMGT Vertical Clustering: MJPEG Tuning Result

For brevity, we only demonstrate the impact of P = 30% and P = 70% for DMGT algorithms with CGGT as a comparison, together with two extremes P = 0% and P = 100% as boundaries. Yet, swiping P in between would yield a similar impact bound by the results of these P values. To visualize the impact of the four tuning schemes (two extremes of P are omitted for simplicity), Figure 4.13 plots them as to the overall cumulative computation and traffic of the model. The plot shows how leaf blocks with increasing BlockSize (X-axis) contribute to the cumulative computation (left Y-axis) and cumulative traffic (right Y-axis). Note that all axes are in log scale. The cumulative computation (or traffic) indicates that blocks of a certain BlockSize or smaller have contributed to the total computation (or traffic) demand. The solid lines denote the computation growth and the dashed curves show the traffic cumulation.

Together with Table 4.5, we observed that DMGT offers both total traffic reduction, block reduction, forced leaf reduction and AFP gain comparing to the previous established CGGT results. Given P = 30%, total blocks are reduced from 89 to 65, the total traffic reduced from 133M to
CHAPTER 4. SPECIFICATION GRANULARITY

Table 4.6: DMGT-Synthesis Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Tuning Scheme (P)</th>
<th>Num. of Blocks</th>
<th>Num. of PE</th>
<th>Model Comp. Demands</th>
<th>Model Traffic Demands</th>
<th>Traf. Reduction</th>
<th>AFP</th>
<th>Flexibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>MJPEG Encoder</td>
<td>P = 70%</td>
<td>18</td>
<td>3</td>
<td>76M</td>
<td>89M</td>
<td>48%</td>
<td>84%</td>
<td>13%</td>
</tr>
<tr>
<td>MJPEG Encoder</td>
<td>DMGT-Synthesis</td>
<td>5</td>
<td>3</td>
<td>75M</td>
<td>86M</td>
<td>50%</td>
<td>96%</td>
<td>0%</td>
</tr>
<tr>
<td>MJPEG CODEC</td>
<td>P = 80%</td>
<td>21</td>
<td>3</td>
<td>103M</td>
<td>173M</td>
<td>45%</td>
<td>81%</td>
<td>6%</td>
</tr>
<tr>
<td>MJPEG CODEC</td>
<td>DMGT-Synthesis</td>
<td>8</td>
<td>3</td>
<td>102M</td>
<td>170M</td>
<td>46%</td>
<td>94%</td>
<td>0%</td>
</tr>
<tr>
<td>Corner Detection</td>
<td>P = 60%</td>
<td>14</td>
<td>3</td>
<td>186M</td>
<td>520M</td>
<td>19%</td>
<td>79%</td>
<td>24%</td>
</tr>
<tr>
<td>Corner Detection</td>
<td>DMGT-Synthesis</td>
<td>7</td>
<td>3</td>
<td>185M</td>
<td>510M</td>
<td>21%</td>
<td>82%</td>
<td>0%</td>
</tr>
</tbody>
</table>

106M and AFP increased from 44% to 55%. The similar benefits is observable when P = 70%, where the AFP achieves up to 84% comparing to 79% in CGGT and traffic shrinks from 92M to 89M with a block reduction from 33 to 18 blocks in total.

Shown in both Figure 4.13 and Table 4.5, a higher P produces less total block numbers with bigger blocks sizes as small blocks have been emerged together. Due to the model hierarchical limitations, it is not possible to remove all trivial blocks while only keeping heavy blocks. This effect can be examined by the forced leaf blocks. In DMGT, with the same P tuning, the forced leaves are reduced comparing to the previous CGGT algorithm benefited from the vertical merging. The forced leaves numbers are reduced from 15 to 5 and 9 to 4, respectively. The overall preferred tuning is chosen with P = 70%, as it produces a specification with 12 leaf blocks and only 4 forced leaves. The total model speedup is around 8% and the traffic demand is reduced around 48% comparing to a base model specification with full block specification synthesis.

4.4.4 DMGT-Synthesis Phase: Mapping-aware Specification

For the discussion of the mapping-aware specification, we have expanded to include all three media applications to generalize our observations. Table 4.6 gives the tuning result of these applications to demonstrate the Synthesis-Phase impact. Each application is tuned by DSE-Phase and selected for the mapping-aware specification with a user specified PE mapping. For simplicity, we only show the final P that is used for the final specifications and assume all PEs are a combination of 1 CPU (ARM7TDMI) and 2 Hardware.

Given a tuned specification as input for all three applications, we observed the Synthesis-Phase algorithm produces a mapping-aware specification with a further block reduction. Blocks in the Encoder have reduced from 18 to 5. CODEC has 8 blocks and Corner Detection only has 7 blocks. Traffic reduction also increases to 50% for Encoder, 46% for CODEC and 21% for Corner
Figure 4.14: DMGT: Flexibility vs. AFP

Detection. Likewise, the AFP gain boosts to 96%, 94% and 82% respectively for each application. The flexibility is normalized to 0 as the synthesized specifications are no longer intended for exploration. Due to the majority of the blocks in Corner Detection are large, the traffic reduction is not as obvious as the Encoder examples yet the AFP still increased.

To generalize all our experimental results, Figure 4.14 and Table 4.6 quantify the trade-off and indicates the design phases. Our DGMT provides for the DSE-phase early design space pruning, reducing the flexibility to a meaningful 38% - 12% (only selected values shown in table) and simultaneously improves AFP (≥79%). Adjusting $P$ gives a knob to control DSE freedom. Once a mapping is known, i.e. for the Synthesis-Phase, DGMT yields the absolutely necessary flexibility achieving almost all of the AFP (≥82% depending mapping diversity). In result, our DGMT improves AFP of the design process by eliminating meaningless mapping flexibility. At the same time, it improves AFP of the design itself by facilitating block fusion.

### 4.5 Summary

In this chapter we have identified the trade-off between model granularity and intra-block optimization in a specifications synthesis process. Model granularity impacts the specification mapping flexibility as well as AFP. Many small blocks with negligible contribution to computation may only introduce overhead, may not lead to meaningful design space exploration alternatives, and
CHAPTER 4. SPECIFICATION GRANULARITY

thus only artificially complicates any DSE. Conversely, few large blocks reduce the overall computational demands through inter-block optimizations during the specification synthesis, but will limit the mapping flexibility. In this chapter, we have analyzed and quantified the trade-off through three real-world media examples.

To aid the designer in navigating the trade-off, we introduced Demand- and Mapping-aware Granularity Tuning (DMGT) approach. It identifies leaf blocks with block fusion to produce an overall specification model in desired granularity as specified by the user. Our approach enables designers to navigate the trade-off to produce specifications with over 84% AFP with 90% block reduction simplifying DSE complexity. Specifications for the DSE-Phase expose only meaningful blocks reducing the flexibility to meaningful 12% (of the original block numbers) while already obtaining optimization potential and reducing traffic up to 50%. When, mapping decision are known, in the Synthesis-phase, DMGT maximizes AFP by generated the exact blocks needed for the desired target mapping. Overall, DGMT strikes the balance between AFP and flexibility for exploration, and can custom match the required flexibility for synthesis maximizing AFP.
Chapter 5

Flexibility Overhead on Software Synthesis

5.1 Introduction

Component-oriented principles are frequently employed in today’s Multiprocessor System-on-Chip (MPSoC) design providing advantages such as hierarchical composition, clearly defined interfaces and dependencies, as well as multiple abstractions. Following these principles, complex models can be described for both software and hardware through System-level Design Languages (SLDL), e.g. SystemC \cite{57} and SpecC \cite{49}, which are widely used in modern Electronic System Level (ESL) flows. In particular, SLDLs incorporate component-oriented principles by supporting the concept of interfaces and ports separating computation from communication. This separation of concerns allows to orthogonally capture multiple design aspects and enables a flexible model composition. Modules implementing an interface (regardless of actual realization) can be connected to ports that require the interface.

This flexible composability empowers, for example, multi-level communication abstractions. While providing the same interface, communication may be realized from a simple user-level abstract channel at the specification level down to a Transaction Level Models (TLM) emulating complex interconnect structures. It allows to separate functional models from timing models and enables dedicated refinement approaches to transition from specifications to detailed implementations. However, this flexibility comes at cost of some simulation execution speed. During run-time, the connectivity has to be evaluated to invoke the implementing component, which constitutes the
Dynamic Dispatch Overhead (DDO).

As Figure 5.1 shows, the component-oriented flexibility is very beneficial during the specification phase all the way through TLM-based system exploration – allowing efficient model construction via module-reuse and function encapsulation. However, such flexibility is no longer required after the platform and mapping decisions have been identified when the target synthesis phase starts. In other words, the model composition (by definition) is fixed in synthesizing embedded software. The previously beneficial flexibility now becomes meaningless, and the cost of DDO may dominate in resource-constrained environments.

At the C++ language level (SystemC is based on C++), an interface is expressed as an abstract class with pure virtual methods. A derived class then implements these methods realizing the actual functionality. Different classes can implement the same interface. The class to be called at run-time is determined through port binding. At run-time, the actual method of the derived class
is resolved through a virtual function table (VTABLE) also referred as *dynamic dispatch*.

Generally, the overhead for a virtual function table call itself is low (2 cycles \[32\]). More importantly, however, this indirection hinders inlining optimizations. Especially when the callee function has only minimal computation, the DDO becomes quite significant on embedded platforms. Measurements on a JPEG encoder show a slowdown of 12% to 16% on an ARM9 processor due to virtual function tables and the lost opportunity of inlining. Furthermore, the slow-down has occurred in several SLDLs and was observed in SystemC, SpecC and component-oriented C code.

Current SW synthesis approaches, however, do not address the DDO. They either use C++ polymorphism allowing for interchangeability between host model and target implementation \[64, 73\], or generate embedded C code retaining the flexibility through explicit virtual function tables \[122\]. To improve the performance and quality of SLDL synthesized embedded SW, new approaches are needed to eliminate or reduce the DDO.

In this chapter, we propose a DDO-aware SLDL to embedded C code synthesis approach achieving DDO reduction/elimination through dispatch type analysis. It utilizes the fact that the complete model (with all connectivity) is known during the SW synthesis phase, which creates optimization opportunities via static dispatch type analysis to determine the call type during compile-time. For each call to a statically determinable type, our approach emits a direct function call, instead of the more general (yet slower) polymorphic method call (through a VTABLE). We implement the solution in the current available SLDL compiler: sc2c \[122\] to synthesize embedded C code from SpecC SLDLs.\footnote{We chose the SpecC SLDL as a compiler is readily available. The concepts are applicable to other SLDLs, such as SystemC. To simplify the description, we use SystemC terminology.} Our results with synthetic benchmarks show that the DDO can be completely eliminated saving 14 cycles per port/interface indirection compared to SystemC on an x86 host and 16 cycles on an ARM9 target. The DDO-reduction approach shows significant benefits on a real-world example of the JPEG encoder. When synthesized to an ARM9-based HW/SW platform, static dispatch conversion yielded a 12% to 16% speedup in execution time.

The chapter is organized as follows. Section 5.2 examines on important background of virtual function realization and language concepts. Section 5.3 describes our DDO software synthesis approach. Section 5.4 quantifies the benefits and offers analytical insights. Lastly, Section 5.5 summarizes the chapter.
CHAPTER 5. FLEXIBILITY OVERHEAD ON SOFTWARE SYNTHESIS

5.2 Background

This section introduces relevant language concepts to help explain our approach. Component-oriented principles are commonly used in languages such as SLDL, Verilog, or VHDL, for separating concerns in system-level and digital designs. Importantly, the definition of an API is separated from its implementation. Different components can provide the same API and yet the inter-component connectivity regulates which component may be actually called. As a result, the interface/port concept simplifies component reuses and allows building loosely coupled independent modules. In the context of SLDLs, interface/port concept is intensively utilized for separating the concerns of computation and communication in multiple design and refinement layers.

Figure 5.2 visualizes an example illustrating the use of an interface (sc_interface) and ports (sc_port) in SystemC. The channel of type C1 provides the interface iCall. Module M1 defines a port of the type iCall, which thus requires the interface iCall. Within M1, the channel implementing iCall is invoked as a port method call: portVar.method(). When instantiating M1 as m1, any channel or module implementing such interface iCall can be mapped to m1’s port, which is referred to as port map. In the example, instance m1 of type M1 is connected to channel instance c1 of type C1. Note that both modules and channels in SystemC can implement interfaces and have ports for connectivity. As their distinction is not relevant for the purpose of this article, we thus only focus on modules for simplicity.

Since SystemC is based on C++, we now focus on the C++ realization of the interface/port concept. A SystemC interface is realized as an abstract class with pure virtual methods. A module implementing such interface inherits from the abstract class. Since multiple classes can implement the same interface differently, this is commonly known as polymorphism. A SystemC port is accomplished as a reference to the base class that defines the interface. The connectivity between modules is determined during the instantiation (i.e. the elaboration phase). As the type (class) implementing the interface is unknown at compile time, run-time type resolution is required to resolve

\[\text{Identical concepts hold true for SpecC. For SpecC simulation the SpecC Compiler (SCC) generates C++ using the same C++ language constructs as SystemC.}\]
the implementing class through **dynamic dispatch**. Dynamic dispatch in C++ is actualized through virtual function tables (VTABLE). Essentially, each method in the interface class is identified by an entry in the VTABLE containing a pointer to the actual method. A module implementing the interface (i.e. class inherited from base class) sets the VTABLE for the port (i.e. reference to base class).

Each call to a virtual method incurs an overhead due to the indirection through the VTABLE. By itself, this overhead is small (2 cycles [32]) outweighed by the flexibility benefits during specification and model construction. However, with VTABLE indirection the compiler cannot inline methods, which then aggravates the performance penalty. We measured the overhead of 14 cycles (16 cycles) per call indirection (see Section 5.4) versus inlining on x86 (ARM).

The flexibility offered by such method visualization is useful during specification exploration and thus the DDO is acceptable. However, for target-specific SW synthesis, this flexibility is no longer needed, and DDO unnecessarily slows down performance. In practice, models with layer-based communication particularly suffer as they have many methods with little computation, which are ideal inline candidates for performance gain. Since the complete model with all connectivity is known during the synthesis stage, optimization opportunities arise. If effectively only static connectivity exists between modules and interfaces, dynamic dispatch can be replaced by static dispatch (direct function calls) removing the DDO penalty for the synthesized applications.

Figure 5.3 intuitively visualizes the distinction between the dispatch types of static and dynamic. Dynamic dispatch is needed when multiple instances of the same module map to different module types implementing the same interface. On the contrary, if all instances of the same module type map to instances of the same interface implementing module type, static dispatch is determinable and producible. Figure 5.3a requires the dynamic dispatch as the instances m1a and m1b of M1 are mapped to instances of different types (m1a to M2:m2 and m1b to M3:m3). Nevertheless, static dispatch is sufficient for module M3 in Figure 5.3b as all of its instances map to instances of the same module type M4:m4a, M4:m4b. In our compiler implementation, the DDO can be eliminated by replacing the VTABLE indirection with a direct method call as the static dispatch is sufficient.

### 5.3 DDO-aware Software Synthesis

This section describes our approach for DDO-reduction. We first overview the software synthesis putting our optimization into context and then describe the static dispatch analysis which
CHAPTER 5. FLEXIBILITY OVERHEAD ON SOFTWARE SYNTHESIS

5.3.1 Software Synthesis Overview

The software synthesis [105] produces embedded code from either an SLDL specification or a refined TLM model. The latter case requires synthesis being invoked once for each Processing Element (PE). For this, the synthesis approach implements the application captured in an SLDL, including SLDL-specific keywords, such as modules, tasks, channels, and port mappings, on a target processor. Instead of compiling the SLDL (i.e., its C++ representation) directly onto the target software platform, our software synthesis generates embedded C out of the SLDL to achieve compact and efficient code.

We divide software synthesis into code generation and Hds generation, shown in Figure 5.4. Code generation deals with the code inside each task and generates flat C code out of the hierarchical model captured in the SLDL. Meanwhile, Hds generation creates code for processor internal and external communication, adjusts for multi-tasking, and eventually generates configura-

Figure 5.3: Dispatch Type Examples

is at the heart of the approach.
CHAPTER 5. FLEXIBILITY OVERHEAD ON SOFTWARE SYNTHESIS

Figure 5.4: Software Generation Flow

The code generation facility [122] produces sequential C code for each task within a programmable component. It translates the hierarchical composition of behaviors in the SLDL into flat C-code consisting of functions and data structures. For SLDL features not natively present in the targeted C language (e.g. port/interface concept, hierarchical composition), code generation realizes these SLDL features out of available C constructs. ANSI-C does not provide an encapsulation for behavior local storage. Therefore, all behaviors’ local variables are added to a module-representing structure. A port becomes a member of the module-representing structure and includes the VTABLE pointing to the implementing methods. In summary, embedded code generation solves similar issues as C++ to C compilers that translated a class hierarchy into flat C code.

Listing 5.1: Dynamic Dispatch C Code

```
void M3_main(struct M3 *This) {
    This->c = *((
        // Get port and VTABLE list
        (This->p1.port->_VPTR
        // index VTABLE
        )[This->p1.offset]
        //call get, pass called module’s *This
    ))->get( This->p1.port );
}
```

Listing 5.1 gives a simplified excerpt for M3 in Figure 5.3b (type casting omitted for
simplicity). The interface iCall has one method: `int get(void)`, assuming that M3 defines the port variable for the iCall port as `p1`. Within the SLDL code of M3, the port method is invoked by: `c = p1.get()`. Listing 5.1 shows how that port call is realized in the generated C code. The typical characteristic is the indirection chain that first gets the list of VTABLEs; then indexes into it to get the desired interface, followed by calling the de-referenced function pointer (pointing to M4.get()). Besides the performance penalty, the indirections in the generated code significantly complicate the debugging process and obscure the code understandability.

In case static dispatch is possible (i.e. when the connected module is the same for all instances of the calling module), then the code can be dramatically simplified as outlined in Listing 5.2. Here, the actual method M4.get() is called directly from calling module, improving the code readability and potentially increase performance.

Listing 5.2: Static Dispatch C Code

```c
void M3_main(struct M3 *This) {
    This->c = M4_get(This->p1.port);
}
```

To identify the dispatch type for each port module type, we thus expand our software synthesis tool-chain with the *static dispatch type analysis* to obtain DDO-aware code generation.

5.3.2 Static Dispatch Type Analysis

The goal of the static dispatch type analysis is to determine whether static dispatch optimization is possible. As such this analysis stage is essential for the optimization potential. To resolve the dispatch type, we use the following rules:

- **Rule 1**: Static dispatch for port method calls of a module is permissible, if all instances of this module type are port mapped to instances of only one module type implementing the interface.

- **Rule 2**: Multi-core systems may execute the same applications on different PEs. Only instances and types within the same PE have to be taken into account for the port mapping analysis.

These rules are implemented as an essential part of *Static Dispatch Analysis* shown in Figure 5.4 where connections between modules are known and the model is complete. More specifically, the dispatch type analysis is split into two algorithms. At first, *ColorInstAndType* colors all
module instances and their types that are to be synthesized for the selected PE (rule 2). Second, 
DispatchTypeAnalysis determines for each port and each module type occurring in the PE whether 
static dispatch is possible (rule 1).

Algorithm 4 Module Type and Instance Coloring

1: procedure COLORINSTANDTYPE(modInst, PE)
2:     if modInst ∈ PE then
3:         for all child ∈ Children(modInst) do
4:             ColorInstAndType(child, PE)
5:             child.Color ← RED
6:             Type(child).Color ← RED
7:         end for
8:     end if
9: end procedure

Algorithm 4 outlines the coloring algorithm. It gets as input parameters the starting in-
stance as well as the PEs. It traverses all children of a PE coloring both the module instances as
well as their types with the color RED as an example. The coloring routine stops at the PE bound-
aries. Afterwards, Algorithm 5 illustrates our dispatch type analysis. The algorithm iterates (line 3)
through all module types of the design that are at least once instantiated in the PE. For each module
type $T$, it goes (line 5) through each port $P$. If the port type is an interface (line 5), it then analyzes
the dispatch type. For this, the algorithm loops through all instances of that module type instantiated
within the PE (line 8). For each module instance $i$, it determines to which module type $MT$ the port
$P$ is mapped through calling GetMappedType($i$, $P$). GetMappedType($i$, $P$) follows the port
mapping to identify the final mapped module type, traversing the hierarchy if needed. If a different
$MT$ is found for one instance of $T$, the annotation staticMappedType is cleared (line 11) indicating
dynamic dispatch is required, and the search stops. Conversely, if all instances of module type $T$
inside the PE are mapped to the same module type $MT$, staticMappedType is set to $MT$ indicating
that static dispatch can be deployed.

According to staticMappedType, code generation emits the appropriate dispatch code.
Port method calls requiring dynamic dispatch are generated with the VTABLE indirection as out-
lined in Listing 5.1. Port method calls with static dispatch are generated with a direct method call
as outlined in Listing 5.2.

To illustrate the working principle of the dispatch type analysis, consider the example in
Figure 5.5a and the simplified corresponding intermediate representation in Figure 5.5b. Within
Algorithm 5 Static Port Type Analysis

1: procedure DispatchTypeAnalysis(Design, PE)
2:   for all $T \in Design.ModuleTypeList$
3:     $\land T.Color = RED$
4:     for all $P \in T.PortList$
5:       if $P.TypeType = Interface$ then
6:         $P.staticMappedType \leftarrow \emptyset$
7:         for all $i \in Design.GetInstance(T)$
8:           $\land i.Color = RED$ do
9:           $MT = GetMappedType(i, P)$
10:          if $P.staticMappedType \neq \emptyset$
11:             $\land P.staticMappedType \neq MT$ then
12:                $P.staticMappedType \leftarrow \emptyset$
13:                break
14:          end if
15:          $P.staticMappedType \leftarrow MT$
16:       end for
17:     end if
18:   end for
19: end procedure
the design, the interface \textit{iCall} is implemented by two module types: \textit{M4} and \textit{M5}, each of which is instantiated once. The module types \textit{M1}, \textit{M2} and \textit{M3} each has a port of the type \textit{iCall}. \textit{M2} is instantiated once, and its port is mapped through its parent’s (\textit{M1}) port to \textit{M4}. \textit{M3} has two instances \textit{m3a} and \textit{m3b}. While \textit{m3a} is connected through the parent’s port to an instance of \textit{M4}, \textit{m3b} is mapped to an instance of \textit{M5}. Hence, static dispatch is possible for \textit{M2}, but dynamic dispatch is required for \textit{M3}.

To give an example of the algorithm operation, consider Algorithm 5 just entering the loop in line 8, investigating module type \textit{T} = \textit{M2}, with its first port \textit{P} = \textit{P1} of type \textit{iCall}. Calling \texttt{Design.GetInstances()} then returns the instance \textit{m2}. Subsequently, \texttt{GetMappedType()} in line 9 searches for the mapped type of the first port. The port map for \textit{m2} is captured at its parent \textit{m1} indicating that the first port of \textit{m2} is mapped to the first port of \textit{M1}. \texttt{GetMappedType()} therefore traverses up in the hierarchy to find the parent of the \textit{M1:m1} instance. The parent of \textit{M1} is the root \textit{Main}. Since \textit{Main} contains the \textit{M1:m1} instance, it also contains the port map that connects to the instance \textit{M4:m4}. Hence, \texttt{GetMappedType()} returns the type \textit{M4}. In line 15 \texttt{P.staticMappedType} is set to \textit{M4}. Since, no other instances of \textit{M2} are found within the design, \texttt{P.staticMappedType} = \textit{M4} remains set, which indicates static dispatch.

After analyzing the only instance of \textit{M2}, the search continues (line 3) with the next type \textit{M3}, yielding instances \textit{m3a} and \textit{m3b} (line 8). The search proceeds with \textit{m3b}. For this, \texttt{GetMappedType()} returns the type \textit{M5} (found through port mapping at \textit{M1}). As the just found type (\textit{M5}) differs from the earlier found type (\textit{M4}), this triggers the condition in line 15. Hence, \texttt{P.staticMappedType} is reset indicating that dynamic dispatch is required and the search for type \textit{M3} port \textit{P1} terminates. The algorithm moves forward with the types \textit{M4} and \textit{M5} and both do not contain any ports (they only implement an interface). Consequently, \texttt{DispatchTypeAnalysis} finishes. It found that any call to methods on the first port of \textit{M2} can be statically dispatched. All other calls require to be dispatched dynamically.

### 5.3.3 Compilation Performance Considerations

We have separated analysis (ie. \texttt{DispatchTypeAnalysis}) from generation to maintain compilation performance. The findings of the analysis is captured in the \texttt{P.staticMappedType} annotation. Then, each time a port method is reached during SW generation, the result lookup occurs in linear time (directly looked up at the port). The analysis phase contains a triple nested loop through all types, ports, and instances. At it’s innermost loop, \texttt{GetMappedType()} traverses up
the hierarchy of port maps. To reduce the search time, GetMappedType() uses intermediate results. In the example of Figure 5.5, the port $P_1$ of M1 is first investigated when analyzing all M1 instances. Hence, when the mapped types for port $P_1$ of $m_2$ and $m_{3a}$ are investigated, GetMappedType() can directly return the type stored at $m_1$ port $P_1$. Note that analysis order depends on the sorting order of the type list in Design.

### 5.4 Experimental Results

To evaluate and demonstrate the benefits of our dynamic dispatch avoidance approach, we have implemented it using SpecC [49] as an SLDL. SpecC was chosen as a parser and intermediate representation (Syntax Independent Representation (SIR) [119]) are available with visibility to all SLDL and ANSI-C operations of the model. SIR provides a rich set of API that simplifies analyzing model composition and connectivity (important for dispatch type analysis, Section 5.3.2). As the SIR exposes all operations in the model, it also captures port method calls. The code generation of the SW synthesis tool sc2c [122] always generates embedded C code with dynamic dispatch. As
part of this work, sc2c was expanded to include the dispatch type analysis and to conditionally emit static (Listing 5.2) or dynamic dispatch code (Listing 5.1). The added DDO-reduction in sc2c has minimally increased the C-code generation time by less than 0.5% on average.

To indicate generality of our approach we compare the Dynamic Dispatch Overhead (DDO) in four languages: **SystemC** (v2.3.0), **SpecC** (v2.2.1) generated C++ for simulation, embedded C generated by the unmodified sc2c, and embedded DDO-reduced C generated by the improved sc2c. The code is then compiled using GCC (g++) version 4.1.2, and executed on RHEL 5.7 on an Intel Core2 Duo, E8500 with 3.16GHz. Additionally, we analyze the target execution on an ARM926ejs (bare-c, no OS) when cross compiled with GCC (v3.4.3).

We first measure the DDO using synthetic benchmarks and then assess the real-world benefits using a JPEG encoder.

### 5.4.1 Synthetic Benchmarks

To quantify the DDO impact, we evaluate the DDO using two synthetic benchmarks as shown in Figure 5.6. In both benchmarks, the module \( M_1 \) calls the port method \( \text{get()} \) 10^5 times (5K times on ARM). The channel \( C_n \) implements the \( \text{get()} \) method and returns a simple math operation’s result. In the model chaining (Figure 5.6a), an increasing number of half channels is employed. Each half channel implements the \( \text{get()} \) method with just another interface method call. The chaining repeats until the final channel \( C_n \) is reached. This results in an increasing number of channel calls (2, 4, 8, 16) in total. This emulates the cost of a layer-based communication implementation with many half channels produced in step-wise refinements. We expect an increasing delay with the increasing chaining length. The model hierarchy (Figure 5.6b) contains an increasing number of hierarchy levels leading to the final map to the implementing channel. This setup is indicative of a model with many hierarchy layers. These two effects are typically observable in SLDL incremental refinements and the back-end synthesis in current ESL design flow [30].

Figure 5.7a shows the cost for the chaining benchmark (using O2 compiler optimizations). On the x-axis, it indicates the chaining length, meaning the number of half channel calls before getting to the actual implementing channel (see Figure 5.6a). The y-axis records the execution time in the number of cycles on the simulation host. All four languages show a linear increase of execution delay with the growing chaining length. This indicates a constant dynamic dispatch overhead per indirection. SystemC and SpecC both share the highest cost with about 200 MCycles with 16 indirections. The generated embedded C is about 30% more efficient, needing 129 MCycles.
Our proposed DDO reduction improves by an additional 25%. Overall, the DDO reduced C is twice as fast as the C++ based counterpart.

Figure 5.7b uses the same benchmark but executes the generated embedded C cross compiled with GCC O2 on the ARM9. Identical to the host execution, the delay linearly increases with the chaining length. Since the ARM9 does not run any operating system, the graphs are perfectly linear. The original code executes for 1.37 MCycles at 16 indirections (5K iterations). Indirections on the target are much more costly than on the host. With our DDO reduction, the optimized code executes four times faster with only 0.33 MCycles.

Figure 5.7c illustrates the costs for hierarchy levels on the host. All four languages have a flat trend. Although the hierarchy depth increases, the execution time remains constant. This indicates that the hierarchical indirections do not increase the cost. From a language perspective, this is expected as regardless of the hierarchy depth, always the local VTABLE is used and there is no hierarchy traversal (e.g. to get to a parents VTABLE). Next, we investigate the impact of compiler optimizations. We focus on the chaining benchmark since the overhead in the hierarchical case is constant.
CHAPTER 5. FLEXIBILITY OVERHEAD ON SOFTWARE SYNTHESIS

Figure 5.7: Synthetic Benchmarks Execution Time
CHAPTER 5. FLEXIBILITY OVERHEAD ON SOFTWARE SYNTHESIS

The dynamic dispatch overhead (DDO) over different languages and compiler optimizations is plotted in Figure 5.8. The DDO is expressed in cycles per port/interface indirection, meaning how many cycles of overhead are added due to each individual half channel call, which basically is the cost of one call through VTABLE. The DDO captures the incremental cost (i.e. the slope in Figure 5.7a) excluding constant operations) for each call. The results are grouped on the x-axis by compiler optimizations. SpecC and SystemC show similar results with a DDO of 12 cycles for O0 and O1. With O2 or O3, the DDO drops to 8 cycles for SpecC and 9 cycles for SystemC.

The generated C code is more efficient, with 10 cycles for O0, dropping to 5 cycles for O2 and O3. The DDO-reduced C is about 1 cycle faster for O1 and O2. A much more dramatic result is visible for O3 when DDO drops to 0 cycles. The compiler has inlined all calls regardless of chaining length. In the result, only the computation in the final channel actually causes execution delay whereas the port/interface indirections are now overhead free.

The benefits of DDO reduction are more pronounced considering the target micro-architecture of an ARM9. First, the VTABLE overhead for non-optimized code is much larger in comparison to the host execution. With 39 cycles, it requires 2.6 times more cycles than host execution of the same code. This dramatic overhead is due to the simpler memory hierarchy (lower chance of cache hit) and the more conservative code generation when optimizations are disabled. Still with the DDO-reduced code, the binary without compiler optimizations shows 24 cycles per indirection. Continuing with the optimizations from O1 through O3, the dynamic dispatch code requires 16 cycles in these cases. Nevertheless, the DDO-reduced C increasingly benefits. With O1, the overhead drops to 9 cycles, and further to 3.5 cycles in O2. Similar to the execution on the host when using O3 optimization, the DDO is completely eliminated (hierarchical inlining of the functions). This highlights the tremendous benefits by DDO reduction for embedded targets.

DDO reduction is highly desirable as it can eliminate the cost for port/interface indirection once the compiler inlining is triggered. Our DDO-reduction in addition leads to more readable embedded C code significantly simplifying code modifications/debugging after synthesis. The actually observable performance benefits will depend on the model. The chaining length and the amount of computation performed at each level impact the achievable speedup. To give an indication of the opportunities, the next section investigates into a real-world example of a JPEG encoder.
5.4.2 JPEG Encoder

To highlight the benefits of our DDO-reduction in a real-world scenario, we utilized a JPEG image compression [12] benchmark whose encoder is captured in SpecC. The mapping decisions and target architectures are outlined in Figure 5.9. The JPEG encoder executes on an ARM926EJS at 100MHz. BMP input and JPEG output are mapped to own custom hardware components and communication occurs through the AMBA AHB. We use SCE [30] to generate a TLM implementing the JPEG encoder over the selected architecture realizing the mapping decisions. SW synthesis (sc2c) then generates the application and the required Hardware-dependent Software (HdS), such as boiler-plate software and drivers. A polling-based synchronization is chosen for simplicity. Since both input and output HW blocks are always available, no synchronization overhead is expected.

We anticipate the most optimization potential in the driver code. The HdS drivers are generated in a layered approach using the component-oriented separation. To gain more insight into the influence of communication onto the DDO reduction, we vary the communication patterns of the JPEG encoder model. In particular, the input and output communication paths (BMP In to JPEG,
and JPEG to JPEG Out) are configurable in data granularity with a coarse and fine grained option for each. The input can either operate on individual pixels, or coarser on a whole pixel row. The latter will reduce the number of transactions by using larger transactions. The output is configurable to operate on individual bytes or more efficiently by using a queue for buffered communication (queue size 256 bytes). If selected, the queue is mapped to the processor. This increases efficiency as the processor writes with small operations (mostly bytes) and the JPEG output retrieves larger blocks. Similar to the input, this reduces the number of transactions by using fewer, yet larger transactions. Larger transactions yield fewer calls into the layered drivers. Thus, we anticipate a lower speedup through DDO-reduction versus using smaller transactions.

Table 5.1 quantifies the DDO-reduction potential for the JPEG encoder example in its four configurations. In addition to the execution time, it shows the speed up as well as the number of dispatches converted to static dispatches.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>C [ms]</th>
<th>DDO-red. C [ms]</th>
<th>Improvement</th>
<th># Static Dispatch</th>
<th># MAC drv. calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td>queue</td>
<td>39.64</td>
<td>34.74</td>
<td>12.4%</td>
<td>89</td>
<td>11469</td>
</tr>
<tr>
<td>row</td>
<td>byte</td>
<td>42.69</td>
<td>36.89</td>
<td>13.6%</td>
<td>85</td>
<td>16560</td>
</tr>
<tr>
<td>pixel</td>
<td>queue</td>
<td>54.01</td>
<td>46.31</td>
<td>14.3%</td>
<td>85</td>
<td>36351</td>
</tr>
<tr>
<td>pixel</td>
<td>byte</td>
<td>57.06</td>
<td>47.86</td>
<td>16.1%</td>
<td>81</td>
<td>41442</td>
</tr>
</tbody>
</table>

For the JPEG encoder, 81-89 port method calls could be converted from dynamic to static dispatch in the code. With the coarser communication, the number increases slightly, as additional driver code (MAC driver calls in Table 5.1) is generated for the buffered communication. Despite the additional code, the JPEG encoder with coarser communication is significantly more efficient when comparing the total execution time. It executes for 40ms in its coarsest configuration (row, queue), while taking 57ms with the finest grained communication (pixel, byte).

All configurations significantly benefit from DDO-reduction. The speedup through DDO reduction ranges from 12.4% for the coarsest to 16.1% for the finest grained communication. Although fewer port method calls are converted to static dispatch in the fine-grained case (81 instead of 89), the drivers are called much more frequently (41K calls instead of 11K calls) leading to the larger speed up through our optimization. Irrespective of communication granularity, our approach offers tremendous benefits as it eliminates DDO for the generated embedded SW while also improving code readability.
CHAPTER 5. FLEXIBILITY OVERHEAD ON SOFTWARE SYNTHESIS

5.5 Summary

This chapter analyzed the dynamic dispatch overhead (DDO) associated with modern SLDLs utilizing component-based design principles (e.g. for separating computation from computation). The overhead occurs through virtual table indirections, but also hinders aggressive inlining by the compiler. The overhead is measurable even with maximum compiler optimizations (-O3). Per indirection, it requires 15 cycles for a SystemC model running on an x86 host. Even generated C with dynamic dispatch takes 8 cycles per indirection (16 cycles on an ARM9 target). During specification and exploration, the benefits of component-based design (e.g. module reuse, separation of communication / computation) clearly outweigh the cost in DDO. For embedded SW synthesis, however, only limited flexibility is needed as target mapping is known and thus DDO becomes performance detrimental.

We introduced DDO-aware SW synthesis. Given an SLDL model for target synthesis, it eliminates any un-necessary flexibility avoiding the DDO whenever possible. For this, it analyzes the module connectivity in an SLDL model and emits simpler, more efficient static dispatch code if resulting final port mapping can be statically determined.

We have demonstrated significant benefits of our DDO-reduction for both an x86 host as well as an ARM9 target. Our DDO-reduction sped up a real-world example of a JPEG encoder (executing on an ARM9-based HW/SW platform) by 12 to 16% depending on communication granularity. In addition to performance improvement, the readability of the generated C is dramatically enhanced. Therefore, our DDO-aware SW synthesis allows benefiting from the component-based flexibility during specification and exploration phases, without incurring the DDO performance penalty for synthesized embedded software.
Chapter 6

Conclusion and Future Work

In this thesis, we proposed a specification synthesis approach that joins Algorithm-Level Design (ALD) and System-Level Design (SLD). With this, we established a new Algorithm/Architecture Co-design flow. We designed and implemented a synthesizer: Algo2Spec, which synthesizes Simulink models into SpecC specifications. We have evaluated the efficiency of our approach by exploring several real-world benchmarks. The joint co-design flow offers tremendous opportunities for algorithm designers to explore the platform suitability of algorithms and to tune algorithms to better match platform requirements (e.g. in terms of parallelism). The new methodology avoids the tedious and error-prone process of manual effort. Algo2Spec synthesizes a single block in 3.13s on average, compared to an estimated manual re-authoring in 2.53 hours. With this, Algo2Spec offers three orders of magnitude gains in design productivity for generating the system-level specification.

Furthermore, specification synthesis enables new optimization opportunities with regard to model granularity and intra-block fusion. We analyzed and quantified the trade-off through three real-world multimedia benchmarks. To aid the designer in navigating the trade-off, we introduced Demand-/Mapping-aware Granularity Tuning (DMGT) approach. It identifies leaf blocks with fusion based on computation and communication to produce an overall specification model in desired granularity as specified by the user. Specifications for the DSE-Phase expose only meaningful blocks reducing the flexibility to meaningful 12% (of the original block numbers) while already obtaining 90% of the maximum optimization potential and reducing traffic up to 50%. Overall, DGMT strikes the balance between efficiency and flexibility for exploration, and can custom match the required flexibility for synthesis maximizing efficiency.

Lastly, we introduced DDO-aware SW synthesis. Given an SLDL model for target synthesis, it eliminates any unnecessary flexibility avoiding the DDO whenever possible. By analyzing
CHAPTER 6. CONCLUSION AND FUTURE WORK

the module connectivity in an SLDL model, our tool emits simpler, more efficient static dispatch code if resulting final port mapping can be statically determined. The DDO-reduction accelerated real-world examples up to 16% depending on communication granularity. In addition to performance improvement, the readability of the generated C is dramatically enhanced. Therefore, our DDO-aware SW synthesis embraces the benefits from the component-based flexibility during specification and exploration phases, without incurring the DDO performance penalty for synthesized embedded software.

Our work greatly introduced new tools, methodologies, and optimizations to deeply enhance current design flows. Particularly, algorithm designs are empowered to effortlessly utilize the readily available heterogeneous computing power to achieve an ideal algorithm and platform solution.

One direction is new automatic methods for guiding Parallelism Exploration of Simulink models. Given a Simulink model, we can explore four types of parallelism: a) Apparent Parallelism (structurally non-dependent blocks), b) Task-level Parallelism (e.g. loop unrolling in a loop block), c) Pipeline Parallelism. d) Multi-rate Parallelism. These opportunities exist on the entire model hierarchy and thus not require extra manual model re-construction or modification. The proposed Granularity Tuning can be used to preserve only computationally meaningful blocks in the design to reduce design space exploration complexity. Moreover, inter-block communication is taken into account for block merging for total traffic reduction. Another future path is Multi-objective Tuning. Currently, we investigated the granularity tuning based on computation and communications. Nevertheless, the clustering and merging criteria can be enhanced to include more objectives such as module closeness, characteristics, or underlying platforms. The richness of Simulink models have not been fully exploited, such as multi-rate systems, continuous systems, and complex networks. To fully embrace all Simulink semantics is a continuous work to expand the established co-design framework.
Bibliography

[1] Omg systems modeling language. \url{http://www.omgsysml.org/}


BIBLIOGRAPHY


BIBLIOGRAPHY


[34] Rui-ling Duan, Qing-xiang Li, and Yu-he Li. Summary of image edge detection. Optical Technique, 3(3):415–419, 2005.


[40] PETRU ELES, ZEBO PENG, KRZYSZTOF KUCHCINSKI, and ALEXA DOBOLI. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. 1997.


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


