ABSTRACT

Title of dissertation: HIERARCHICAL MAPPING TECHNIQUES FOR SIGNAL PROCESSING SYSTEMS ON PARALLEL PLATFORMS

Lai-Huei Wang, Doctor of Philosophy, 2014

Dissertation directed by: Professor Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering

Dataflow models are widely used for expressing the functionality of digital signal processing (DSP) applications due to their useful features, such as providing formal mechanisms for description of application functionality, imposing minimal data-dependency constraints in specifications, and exposing task and data level parallelism effectively. Due to the increased complexity of dynamics in modern DSP applications, dataflow-based design methodologies require significant enhancements in modeling and scheduling techniques to provide for efficient and flexible handling of dynamic behavior. To address this problem, in this thesis, we propose an innovative framework for mode- and dynamic-parameter-based modeling and scheduling. We apply, in a systematically integrated way, the structured mode-based dataflow modeling capability of dynamic behavior together with the features of dynamic parameter reconfiguration and quasi-static scheduling.

Moreover, in our proposed framework, we present a new design method called parameterized multidimensional design hierarchy mapping (PMDHM), which is targeted to the flexible, multi-level reconfigurability, and intensive real-time processing requirements of emerging dynamic DSP systems. The proposed approach allows designers to systematically represent and transform multi-level specifications of signal processing applications from a common, dataflow-based application-level model. In addition, we propose a new technique for mapping optimization that helps designers derive efficient, platform-specific parameters for application-to-architecture mapping. These parameters help to maximize system performance on state-of-the-art parallel platforms for embedded signal processing.

To further enhance the scalability of our design representations and implementation techniques, we present a formal method for analysis and mapping of parameterized DSP flowgraph structures, called *topological patterns*, into efficient implementations. The approach handles an important class of parameterized schedule structures in a form that is intuitive for representation and efficient for implementation.

We demonstrate our methods with case studies in the fields of wireless communication and computer vision. Experimental results from these case studies show that our approaches can be used to derive optimized implementations on parallel platforms, and enhance trade-off analysis during design space exploration. Furthermore, their basis in formal modeling and analysis techniques promotes the applicability of our proposed approaches to diverse signal processing applications and architectures.

HIERARCHICAL MAPPING TECHNIQUES FOR SIGNAL PROCESSING SYSTEMS ON PARALLEL PLATFORMS

by

Lai-Huei Wang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Professor Manoj Franklin
Professor Steven A. Tretter
Professor K.J. Ray Liu
Professor Yang Tao, Dean's representative

© Copyright by
Lai-Huei Wang
2014

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Shuvra S. Bhattacharyya for his academic guidance and financial support throughout my doctoral research work in the DSPCAD research group. He provided me great support to pursue and develop my desired research interests. During my PhD education, he has been the superb mentor whom I have learned a lot from. His expertise in the fields of signal processing and model-based design helps significantly to the critical thinking in my research work. Especially, I am grateful to him for his patience and thoughtful understanding during his guidance to my PhD work. I also thank him for giving me valuable research and teaching opportunities. Absolutely, the experience working with Dr. Bhattacharyya has been one of the most important career experience of my life.

I would also like to thank the members of my dissertation committee, Prof. Manoj Franklin, Prof. Steven Tretter, Prof. Ray Liu, and Prof. Yang Tao for their service and constructive comments and feedback.

The research underlying this thesis was supported in part by the Austrian Marshall Plan Foundation, the Laboratory for Telecommunications Sciences, and the US National Science Foundation. I sincerely thank for their support.

It has been a pleasure to conduct research with the DSPCAD members, including Chung-Ching Shen, William Plishker, Hsiang-Huang Wu, George Zaki, Zheng Zhou, Kishan Sudusinghe, Shuoxin Lin, Inkeun Cho, Scott Kim, and Yanzhou Liu. Among them, I specially thank Chung-Ching and Hsiang-Huang for their assistance and guidance during my beginning phase of research in the group.

I would like to thank my parents for their love, encouragement, and support during my PhD program. Also, I am grateful to my sisters for their encouragement during these years.

Last, and most importantly, I would like to give my deepest gratitude to my wife, Cindy. Without her constant encouragement, patience, and support throughout my doctoral studies, I would not have been able to successfully complete my PhD program. Her love is the biggest source of inspiration that enables me to overcome the challenges during my PhD pursuit.

Table of Contents

List of Figures

1 Introduction

Dataflow models are widely used for expressing the functionality of digital signal processing (DSP) applications, such as those associated with audio and video data stream processing, digital communications, and image processing (e.g., see [1]). Dataflow provides a formal mechanism for describing specifications of DSP applications, imposes minimal data-dependency constraints in specifications, and is effective in exposing and exploiting task or data level parallelism for achieving high performance implementations.

In recent years, a variety of computing architectures have been proposed for massively parallel processing (e.g., see [2, 3]). One of the most important classes of parallel computing platforms is the class of multicore processors. Such processors may use hundreds of lightweight cores to achieve application speedup. Graphics Processing Units (GPUs) form one major sub-class of multicore processors. GPUs provide large performance gains for certain types of regularly structured computations that have high degrees of parallelism (e.g., see [4, 5]). However, development of efficient implementations on GPUs requires careful attention to scheduling and resource mapping, and tedious fine tuning may also be required to extract performance gains.

When implementing a dataflow-based signal processing application model on a target platform, scheduling plays an important role (e.g., see [1]). Here, by *scheduling*, we refer to the process of determining which processing resource each actor executes on, and the ordering of execution among actors that share the same resource. By affecting key metrics that include performance, and memory usage, scheduling often has significant impact on implementation quality.

For efficient implementation on parallel platforms, it is useful for application developers to perform complex graph analysis, such as parallelism exploration and buffer size estimation, to map a platform-independent application graph into an efficient platform-specific realization. In the mapping process, parameters associated with the targeted parallel platforms, such as the degree of parallelism available for exploiting application components, are useful for scheduling of the applications. Here, by *scheduling*, we mean assigning application components (tasks) to processing resources and ordering the execution of the application components that share the same resources. The effective configuration of mapping parameters can be critical to optimizing performance. However, such parameter configuration often requires large amounts of human labor — e.g., to acquire profiling-related information for the application with respect to relevant platform properties.

In this thesis, we propose a systematic framework based on dataflow techniques to map signal processing applications onto parallel platforms. Our proposed framework is geared toward achieving optimized performance, and operates using platform-independent application graphs as its starting point. This use of platform-independent specifications helps to promote the retargetability of the proposed methods across different platforms, as well as the lasting utility across multiple generations of the same platform. Our mapping framework employs a form of dataflow modeling called *core functional parameterized synchronous dataflow* (*CF-PSDF*). In our work, CF-PSDF is applied to facilitate efficient scheduling techniques in dynamic systems, and to help optimize the exploitation of parallelism in the targeted platforms.

We also develop in this thesis a novel design method for hierarchical exploitation of parallelism in multi-dimensional signal processing applications. The method provides a formal linkage between hierarchical layers of parallelism in a targeted multicore platform and corresponding subsystems of the application. Scheduling can be optimized with the parallelism exposed from our proposed design method by using established techniques for DSP dataflow graph scheduling (e.g., see [1]). In addition, we also present innovative models for schedule representation. These models aid in the development of code generation techniques, as well as in the formal analysis and transformation of schedules to help meet design constraints.

1.1 Core Functional Parameterized Synchronous Dataflow

Due to the increased complexity of dynamics in modern DSP applications, such as wireless communication systems based on LTE and WiMAX, designers need significant flexibility in the types of functional behaviors that they can efficiently specify and implement. To model complex dynamic DSP systems, a variety of dataflow approaches have been proposed (e.g., see [1]). Some of these can model arbitrary dynamic behaviors, but may lead to inefficient schedules. Others allow powerful scheduling and mapping techniques by restricting the range of dynamic applications that they can accommodate.

Core functional dataflow (CFDF), is a dynamic dataflow model that provides highly expressive semantics for the design of applications with structured dynamic behavior [6]. However, this flexibility, especially when high levels of data-dependent dynamics are present, may result in significant run-time scheduling overhead and reduced predictability in scheduling performance.

On the other hand, *parameterized synchronous dataflow* (*PSDF*) is a modeling technique that provides for systematic integration of dynamic parameter reconfiguration into synchronous dataflow representations [7]. Such an approach enables flexible parameterized modeling as well as strong support for quasi-static scheduling, which allows efficient and predictable scheduling performance for many kinds of dynamic applications. Here, by *quasi-static scheduling*, we mean scheduling techniques that fix a significant portion of schedule structure at compile time, while allowing flexibility for run-time adaptation of this statically-constructed structure based on characteristics of input data and operating conditions [8]. To provide support for powerful quasi-static scheduling techniques, expression of dynamics in PSDF is restricted — in particular, dynamic changes to actor and subsystem dataflow properties are disallowed for some kinds of modeling structures [7].

In this thesis, we develop a new dataflow modeling framework, which is based on careful integration of the CFDF and PSDF models. We refer to our proposed model as *core functional parameterized synchronous dataflow* (*CF-PSDF*). CF-PSDF provides useful trade-offs between dynamic modeling flexibility, and support for efficient quasi-static scheduling. By applying our proposed design methodology based on CF-PSDF modeling, designers can potentially enhance performance of dynamic applications by employing efficient static and quasi-static scheduling techniques locally, and reducing the overhead associated with more general dynamic scheduling strategies. We demonstrate the utility of our proposed CF-PSDF based modeling and design techniques using an application case study involving multi-input, multi-output (MIMO) detection.

1.2 Hierarchical Mapping Approach

Synchronous dataflow [9] has been popular in design of DSP applications because of its useful features, including compile-time, formal validation of deadlock-free operation and bounded buffer memory requirements, as well as support for efficient scheduling and buffer size optimization [1]. However, the SDF model is well suited only for one-dimensional DSP algorithms, such as those in the domains of speech, audio, and digital communication. Multidimensional synchronous dataflow (MDSDF) [10] is a generalization of SDF to multiple dimensions. MDSDF provides an effective model for a variety of multidimensional DSP systems that have statically structured dataflow characteristics.

In this work, we develop new methods for efficient implementation of parallel processing solutions for signal processing systems using MDSDF representations. Our proposed design methods apply dataflow transformations to exploit data parallelism hierarchically from multidimensional dataflow graphs. Our design methods provide a systematic approach for exposing and exploiting parallelism from multidimensional dataflow specifications across different levels of the specification hierarchy.

We demonstrate our proposed new modeling techniques and design methods by applying them to optimize implementations developed using the NVIDIA GPU programming environment [11]. Using our new MDSDF-based design techniques, we demonstrate efficient GPU implementations for integral histogram computations, which form an important class of image processing operations for surveillance and monitoring applications. The results of our experiments demonstrate concretely that our proposed design methods are effective in mapping formal design models for multidimensional DSP systems into efficient implementations on complex multicore processors.

1.3 Scheduling Representation

For dataflow models of large-scale DSP applications, the underlying graph representations often consist of smaller sub-structures that repeat multiple times. *Topological patterns* (*TPs*) have been shown to enable more concise representation and direct analysis of such substructures in the context of high level DSP specification languages and design tools [12]. Furthermore, by allowing designers to explicitly identify such repeating structures, use of TPs provides an efficient alternative to automated detection of such patterns, which entails costly searching in terms of graph-isomorphism and related forms of computation. A TP is inherently parameterized and provides a natural interface for parameterized scheduling, which enables efficient derivation of adaptive schedule structures that adjust symbolically in terms of design time or run-time variations.

In [13], a formal design method is presented for specifying TPs, and deriving parameterized schedules from such patterns based on a schedule model called the *scalable schedule tree* (*SST*). The method ensures deterministic behavior of the system based on compile-time analysis of its behavior, where the behavior may be expressed in terms of parameterizable patterns of actor and edge instantiations. However, this method enforces certain forms of regularity in executing schedules, which restricts the class of schedule structures that can be expressed, and hence the flexibility with which the method can be applied to the mapping of dataflow graphs.

In this thesis, we introduce a more general traversal method, which allows designers and automated schedulers to programmatically construct solutions from within a broad class of execution sequences. This allows for design and representation of a correspondingly broader range of schedules through the common framework of SSTs. To demonstrate our enhanced SST model, we present a case study involving optimized implementation of turbo decoders, which are important and widely used in wireless communication applications.

1.4 Organization of the Thesis

The remainder of this thesis is organized as follows. Background relevant to the research presented in the thesis is discussed in Chapter 2. Chapter 3 presents the CF-PSDF modeling approach for dynamic signal processing applications. Our model for hierarchical representation of DSP flowgraph parallelism is introduced in Chapter 4 through Chapter 5. In Chapter 6, we present our generalized method for managing and traversing dataflow graph schedules. Conclusions and directions for future work are discussed in Chapter 7.

2 Background

2.1 Dataflow Modeling

In dataflow modeling, an application is represented using a directed graph $G = (V, E)$, where V is a set of vertices and E is a set of edges [14]. In the form of dataflow that we employ in this thesis, each vertex (*actor*) $v \in V$ represents a computation of arbitrary complexity, while each edge $e = (v_1, v_2) \in E$ represents a first-in-first-out (FIFO) buffer that provides a logical communication link between actor v_1 and actor v_2. In dataflow graphs, an actor can be executed (*fired*) whenever it has a sufficient number of data values (*tokens*) available on each of its input ports. In the topology of a dataflow graph, an actor with no input edges is called a *source* actor; an actor that has no output edges is called a *sink* actor.

A *static schedule* for a dataflow graph $G = (V, E)$ is a sequence of actors in V that represents the order in which actors are fired during an execution of G. Each actor, when firing, consumes a certain number of tokens at each input port and produces a certain number of tokens at each output port. These numbers of tokens consumed or produced are referred to as dataflow *rates* of the associated dataflow actors or firings. Various types of dataflow models are formed based on the characterizations of the consumption and production rates of actors. For example, in synchronous dataow (SDF) [9], constant valued rates are used; in cyclo-static dataow (CSDF) [15], the rates are in the form of periodic patterns of constant values; in Boolean dataow (BDF) [16], data-dependent forms of rates are employed for the support of modeling dynamic behaviors.

2.2 Multidimensional Synchronous Dataflow

Synchronous Dataflow (SDF) [9] is a specialized form of dataflow that is used for an important class of DSP applications. In SDF, actors produce and consume data at fixed rates. Useful features of SDF include compile-time, formal validation of deadlock-free operation and bounded buffer memory requirements; support for efficient static scheduling; and buffer size optimization (e.g., see [1]).

However, SDF is ideally suited only for one-dimensional DSP algorithms. By expressing arrays in terms of 1-D streams, SDF modeling of multidimensional systems may hide potential data parallelism. Multidimensional synchronous dataflow (MDSDF) [10] generalizes SDF to multiple dimensions to provide an effective model for a variety of multidimensional DSP systems. In an MDSDF graph of dimension M, the number of tokens produced and consumed are given as M-tuples. For each edge, there are M *balance equations*, where the balance equations are used to determine the minimal numbers of actor firings required in each dimension to provide a periodic schedule (i.e., a schedule that can be executed iteratively, as many times as needed, with guaranteed bounded memory requirements for the dataflow graph edges) [10].

2.3 Parameterized Synchronous Dataflow

Parameterized dataflow is a meta-modeling technique that can significantly improve the expressive power of an arbitrary dataflow model that possesses a well-defined concept of a graph iteration [7]. Parameterized dataflow provides a method to systematically integrate dynamic parameter reconfiguration into such models of computation, while preserving many of the properties and intuitive characteristics of the original models. The integration of the parameterized dataflow meta-model with *synchronous dataflow* (*SDF*) provides the model of computation referred to as *parameterized synchronous dataflow* (*PSDF*). PSDF offers valuable properties in terms of modeling systems with dynamic parameters, supporting efficient scheduling techniques, and natural integration with popular SDF modeling techniques [7].

A PSDF specification (subsystem) is composed of three cooperating PSDF graphs, the *init*, *subinit*, and *body* graphs of the specification. The init graph is designed to configure the corresponding subinit and body graphs while the subinit graph can only change parameters in the body graph. The body graph, when executed, performs the main functionality of the subsystem based on the updated set of parameters. For more details on PSDF modeling, we refer the reader to [7].

2.4 Core Functional Dataflow

Core functional dataflow (*CFDF*) is a dynamic dataflow model that provides highly expressive semantics for the design of applications with structured dynamic behavior [17]. In CFDF, an actor is specified as a set of operational *modes.* In each mode, an actor consumes and produces fixed numbers of tokens on its input and output ports, respectively. These numbers of tokens consumed and produced are called the consumption and production *rates* of the associated input and output ports, respectively, and the associated modes. Consumption and production rates for CFDF actor modes can be arbitrary non-negative integers.

During execution, a CFDF actor operates in a unique *current mode* of the actor, which can be maintained as part of the actor state. Each actor has an associated *enable* function, which can be called by a run-time scheduler. The enable function returns a Boolean value indicating whether or not there is sufficient data available on the actor input ports to fire the actor in its current mode. The *invoke* function of an actor consumes data for execution based on the associated current mode. When an actor is invoked, it executes its current mode, produces and consumes data, and updates its current mode (i.e., sets the mode to be used in its next firing).

The enable function need not always be called before invoking an actor — in particular, it need not be called if static analysis of the graph can determine that the required data for the given actor mode will be available at the desired point of invocation. On the other hand, dynamic or quasi-static scheduling techniques may make use of the enable function to help ensure data availability in the absence of static guarantees [17].

2.5 Topological Patterns

For large-scale models of signal processing applications, the underlying dataflow graph representations often consist of smaller substructures that repeat multiple times. A method for scalable representation of dataflow graphs using topological patterns was introduced in [12]. Topological patterns, such as the *ring*, *butterfly*, and *chain* patterns, are pervasive in signal processing applications, including multi-dimensional signal processing systems, where processing of large scale dataflow structures is common. Topological patterns enable concise representation and direct analysis of sub-structures in the context of high level DSP specification languages and design tools. Modeling based on topological patterns also provides a scalable approach to specifying regular functional structures that is formally integrated with the framework of dataflow. This integration allows not only for specification of functional patterns, but also for their analysis and optimization as part of the larger framework of dataflow. For more details on modeling and design based on topological patterns, we refer the reader to [12].

2.6 Generalized Schedule Trees

The *generalized schedule tree* (*GST*) is a compact, tree-structured graphical format that can represent a variety of dataflow graph schedules [18]. In GSTs, each leaf node refers to an actor invocation, and each internal node n (called a loop node) is configured with an iteration count I_n for the associated sub-tree, where execution of the sub-tree rooted at n is repeated I_n times. The GST has been demonstrated to represent looped schedules for dataflow graphs effectively in the context of static, non-scalable schedules (e.g., see [18]).

2.7 General Purpose Graphics Processing Units

In recent years, graphics processing units (GPUs) have become increasingly popular in general-purpose computing applications due to their useful features, such as flexible programmability, tremendous computational ability, high memory bandwidth, and large amounts of parallelism [5, 4]. Compared to conventional microprocessors, GPUs are designed such that more hardware resources (transistors) are dedicated to data processing and less are dedicated to data caching and flow control. The result is that GPUs are effective on computations that involve large amounts of data-parallel computing and relatively small amounts of control flow. Significant levels of acceleration from GPUs have been demonstrated in many fields, such as physics, computer vision, signal processing, and wireless communications (e.g., see [19, 20, 21, 22]). Currently, parallel computing on GPUs is supported in a variety of programming models, including CUDA [23], OpenCL [24], and OpenACC [25]. In this thesis, we employ CUDA as the back-end environment for our GPU-targeted design methods.

CUDA (Compute Unified Device Architecture) is a software programming model for NVIDIA GPUs. In CUDA, a computational unit is wrapped in a *kernel* function, which, when invoked, is executed N times in parallel by N CUDA *threads* in a structure called a *grid*. The CUDA programming model features multi-dimensional and multi-level thread hierarchies. Threads are organized into multi-dimensional (up to three dimensions are supported) *thread blocks*, and multiple thread blocks are combined together to form grids, as shown in Figure 2.1. As with

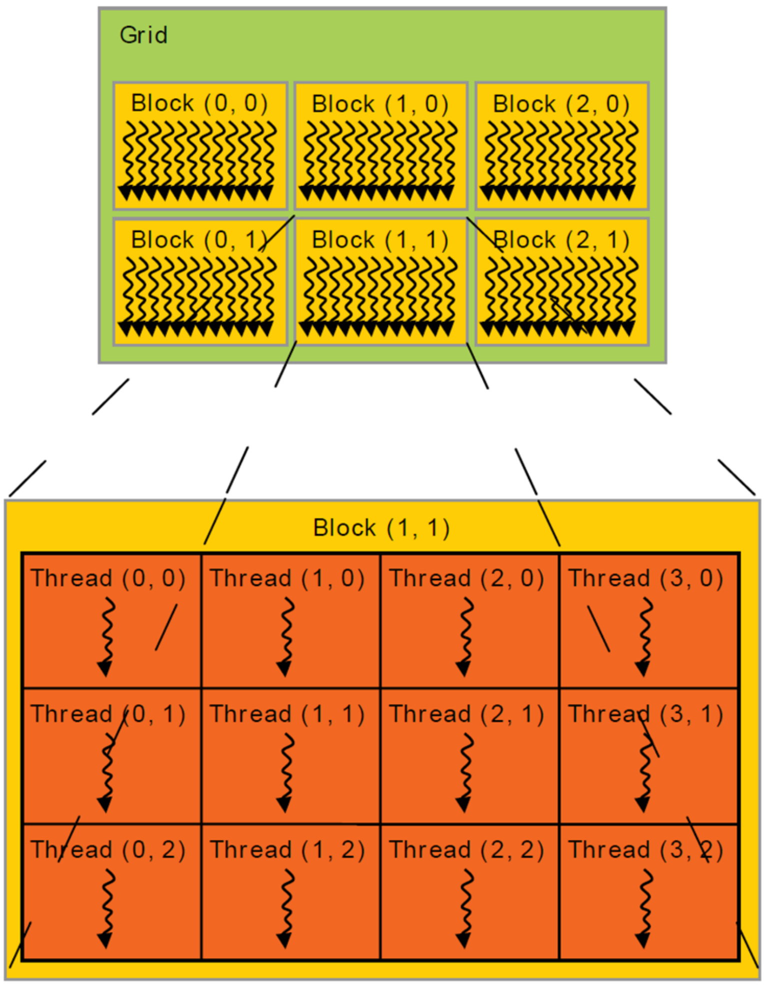


Figure 2.1: An example of thread hierarchy in CUDA.

thread blocks, up to three dimensions are supported for the construction of grids in CUDA.

3 CF-PSDF Modeling and Scheduling

In this section, we address challenges pertaining to modeling and scheduling of dynamic signal processing applications. Here, by dynamic signal processing applications, we mean applications in which the underlying dataflow graphs contain actors whose characteristics, such as production and consumption rates and execution times, can exhibit significant run-time variation [26]. Development of efficient scheduling techniques for dynamic signal processing applications is challenging because of the limited information that is available at compile time about actor characteristics, and because of the potential performance overhead and decreased predictability involved in making significant scheduling decisions at run-time.

In this chapter, we address these challenges for a specific class of dynamic signal processing applications. In the targeted application class, actors that exhibit significant dynamics are controlled by common sources that control the actor dynamics at run-time. These *control sources* can be viewed as specific actors whose outputs are used to control the behavior of other actors. We present a new dynamic dataflow model that groups dynamic actors based on their control sources to enable efficient static and quasi-static scheduling approaches for the associated actor groups (subsystems).

In this chapter, we present an application modeling approach called *core functional parameterized synchronous dataflow* (CF-PSDF), which integrates the CFDF and PSDF models. In CF-PSDF, an application is represented with a two-level hierarchy, as illustrated in Figure 3.1. In the top level (e.g., Figure 3.1(a)), each node is a CF-PSDF actor (with *enable* and *invoke* functions as a CFDF actor) to model data dependent dynamic behaviors that may change dataflow. The bottom level (e.g., Figure 3.1(b)) is composed of three subgraphs (as in PSDF) to provide flexible dynamic parameter reconfiguration.

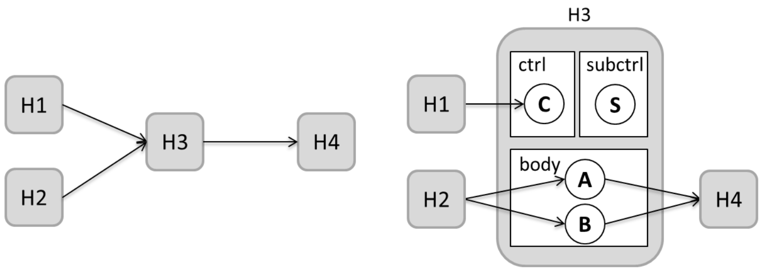
This chapter is based on work presented in [27].

3.1 CF-PSDF Model

In CF-PSDF, a DSP application is modeled through a *CF-PSDF specification*, which is also called a *CF-PSDF subsystem*. A hierarchical actor that encapsulates a CF-PSDF subsystem S (i.e., for instantiation in a higher level subsystem) is called the *CF-PSDF actor* associated with subsystem S. A CF-PSDF actor H can be viewed at its interface as a CFDF actor that has a set of modes, and *enable* and *invoke* functions, which are fundamental components of the CFDF model [6]. When H is executed in a given mode, a fixed number of tokens is consumed and produced at the input and output ports of H, respectively. Across different modes of H, however, the production and consumption rates at the ports of H can vary.

In a CF-PSDF actor H, the encapsulated specification, which we denote by $\sigma(H)$, is decomposed into three cooperating graphs, which we refer to as the *ctrl* (ϕ_c), *subctrl* (ϕ_s), and *body* (ϕ_b) graphs of $\sigma(H)$ (here, "ctrl" is used as an abbreviation for "control"). The actor `H3` in Figure 3.1(b) shows an example of a CF-PSDF actor.

As in PSDF modeling, the body graph of a CF-PSDF specification is intended for use in modeling the core functional behavior of the associated subsystem, while



(a) The top-level graph. (b) The bottom-level graph of H3.

Figure 3.1: An example of a CF-PSDF graph.

the ctrl and subctrl graphs, which are analogous in some ways to the init and subinit graphs of PSDF, control the dynamic behavior of the body graph. This dynamic body graph control is achieved by appropriately configuring selected body graph parameters. As in PSDF, the subctrl graph of a CF-PSDF specification $\sigma(H)$ can configure the parameters in the associated body graph in ways that do not change the production and consumption rates at the interfaces (ports) of H.

The ctrl graph of a CF-PSDF subsystem $\sigma(H)$ is executed once during each firing of H and is allowed to update parameters in the associated subctrl and body graphs. Such parameter configurations may depend on parameters of the enclosing system as well as on run-time data generated from other CF-PSDF actors (i.e., data-dependent parameter updates).

On specific parameter that is configured in the ctrl graph of $\sigma(H)$ is a special parameter $\mu(H)$ that controls the execution mode of H. The ctrl graph is the basic mechanism in CF-PSDF for determining this execution mode. After the ctrl graph of $\sigma(H)$ executes and $\mu(H)$ is updated, the production and consumption rates at the interfaces of H are fixed until the next execution of the ctrl graph. Furthermore, the control information processed in the ctrl graph of $\sigma(H)$ can be shared by "exporting" tokens (through dedicated dataflow graph edges) to ctrl graphs in other CF-PSDF actors within the enclosing application model. This mechanism of "sharing" control information, which represents a departure from the parameterized dataflow meta-model, can facilitate local scheduling and mapping optimization, and help avoid repetitive computation of control information. Additionally, the ctrl graph can process data from input ports of $\sigma(H)$ and produce data onto the output ports of $\sigma(H)$. This is more flexible compared to the init graph of PSDF, where such linkages to the ports of the enclosing PSDF actor are not allowed.

The modeling flexibility of CF-PSDF compared to PSDF is illustrated in Figure 3.2. In the PSDF subsystem shown in Figure 3.2, the production rate of actor A is independent of the output of actor X. However, in some applications it can be useful to model behaviors where the production rate of actor A in this kind of a subsystem structure is *dependent* on the output of actor X. This kind of data-dependent dynamics, which is not expressed in the more predictable, PSDF-style specification of Figure 3.2, can be useful to model precisely when developing DSP applications. For example, in wireless communications, a turbo decoder with a dynamic iteration count may or may not execute one more iteration according to the run-time output of the current iteration [22]. Another example of this kind of dataflow dynamics is discussed in Section 3.5.

On the other hand, in CF-PSDF, designers can pass control tokens from actor X to the ctrl graph of a CF-PSDF subsystem, as shown in Figure 3.2(b). Then, the ctrl graph can configure the dataflow (production and consumption) rates of A

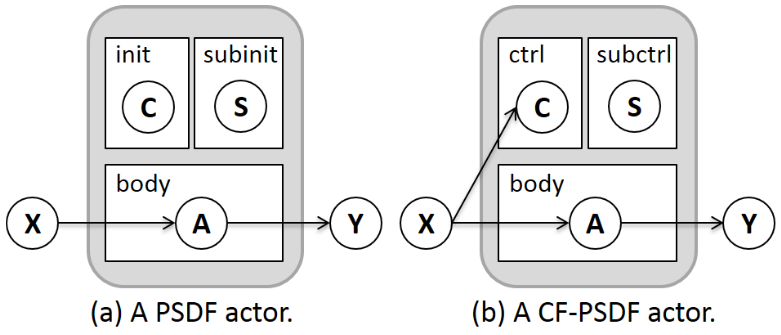


Figure 3.2: Examples of PSDF and CF-PSDF actors.

through the CF-PSDF mechanism of parameter reconfiguration based on subsystem input tokens (input tokens arriving from actor X in this case). A disadvantage in supporting this kind of dynamics is that the efficient quasi-static scheduling techniques that have been developed for PSDF models (e.g., see [7]) are in general not applicable to CF-PSDF specifications. However, in Section 3.4, we develop new scheduling techniques that exploit the structure of CF-PSDF models, and permit derivation of efficient schedules from such models.

3.2 Multi-Mode Actors

In this section, we develop scheduling techniques for mapping CF-PSDF graphs into efficient implementations.

The hierarchical, mode-oriented structure of CF-PSDF modeling allows designers to specify complex applications with more concise graphs representations, where related functionality can be grouped together naturally under common actors or subsystems. For example, a P-QAM mapper, which maps blocks of log_2P input bits to P-QAM symbols, consumes P tokens (bits) and produces one token (QAM symbol). An SDF representation of this functionality would typically require three separate actors for 4-QAM, 16-QAM, and 64-QAM processing, while this entire functionality can be encapsulated within a single, multi-mode CFDF actor. However, in a CFDF graph, additional control modes may be needed to provide for correct transitioning between operational states (e.g., see [6]), which may increase design effort and introduce scheduling overhead. In CF-PSDF, we alleviate these problems by applying a central control mechanism, through ctrl and subctrl graph execution, and a modeling approach based on designer-specified sets of practical mode combinations (functional modes) across body graph actors. This concept of functional modes is discussed next, in Section 3.3.

3.3 Subsystem Modes

In CF-PSDF, it is not possible for one body graph actor to have direct control over the dataflow rates of another actor in the same body graph. For example, in the graph of Figure 3.1(b), the dataflow rates of actors A and B can be varied based on output from `H1`, but the dataflow rates of A are prohibited from depending on outputs of B, and vice versa. This condition ensures that the mode transitions of all actors in a body graph ϕ_b can be configured centrally from the ctrl graph based on system parameters and run-time data. This centralized, ctrl-graph based control of actor modes can help to eliminate certain local (actor-level) modes and transitions that are employed in pure CFDF models to ensure proper transitioning between processing states.

Each CF-PSDF actor (subsystem) H has a special mode called the *control mode* of H, which is used to execute the ctrl graph of H, and update parameters, including the next mode parameter $\mu(H)$. The control mode can in general consume and produce data at the interface ports of H, as described previously.

Apart from the control mode, a CF-PSDF actor H may have any number of additional modes, which are referred to as the *functional modes* of H. Execution of H proceeds based on alternating sequences of the control mode and a functional mode (i.e., between each pair of successive functional mode executions, there is exactly one execution of the control mode). Each functional mode of H corresponds to a unique set of modes for all actors that are contained in the associated body graph, ϕ_b. That is, for each functional mode m of H and each actor α in ϕ_b, there is a unique mode $z(m, \alpha)$ of α that governs the execution state of α whenever H executes in functional mode m. Thus, in each functional mode m, ϕ_b can be viewed an SDF graph $G_{sdf}(m)$, which can be analyzed and scheduled by leveraging the large body of existing techniques for SDF (e.g., see [1]).

Note that in CF-PSDF, the set $F(H)$ of functional modes of H is defined explicitly by the designer. An alternative approach would be to derive $F(H)$ by enumerating all possible mode combinations across the actors within ϕ_b. However, this approach is clearly not scalable since, for example, there is no polynomial bound on such mode combinations.

Indeed, in practical DSP applications, many mode combinations may be uninteresting (e.g., redundant or simply not useful). In Figure 3.1(b), for example, suppose that actors A and B are both P-QAM mappers with three modes each for

$P = 4$, 16, and 64. The set of all mode combinations for `H3` contains $3 \times 3 = 9$ combinations. However, at any given time during actual execution of the system, the values of P will be identical for both A and B — only three mode combinations are relevant in the design of `H3`. Thus, `H3` is designed such that $F(H3)$ contains only three modes.

In previous work, methods have been developed to detect and eliminate unreachable mode combinations in CFDF graphs [6]. However, in practical scenarios, such as the example of Figure 3.1(b), it can be difficult to detect all unused modes without designer guidance. Automated techniques, such as those developed in [6], can be used in a complementary fashion to the designer-specified approach in CF-PSDF (e.g., to remove unused modes from the specified functional mode set). Integrating such automation into the CF-PSDF framework is an interesting direction for future work.

3.4 Scheduling Techniques

In CF-PSDF, dynamically parameterized and dynamic dataflow subsystems are represented with two-level hierarchies, as illustrated in Figure 3.1. In the top level (e.g., Figure 3.1(a)), each actor is a CF-PSDF actor with associated enable and invoke functions, which have similar roles as in the pure CFDF model. The lower level of the subsystem design hierarchy (e.g., Figure 3.1(b)) is composed of three subgraphs to provide flexible dynamic parameter reconfiguration. This structured decomposition into three subgraphs is based on a similar kind of decomposition provided in PSDF, but with significant adaptations to make the modeling approach more flexible and more coupled to CFDF design techniques.

CF-PSDF provides a natural framework for quasi-static scheduling based on the decomposition of a CF-PSDF subsystem H in terms of it functional modes $F(H)$ and the associated set of SDF graphs

$$S(H) = \{G_{sdf}(m) \mid m \in F(H)\} \tag{3.1}$$

that characterizes the body graph ϕ_b. Each graph $\{R \in S(H)\}$ can be scheduled using SDF techniques, and based on specific operational constraints (e.g., constraints on throughput, latency, or buffer memory requirements) that are associated with the corresponding functional mode of H. The resulting set of SDF schedules $S(R) \mid R \in S(H)$ can then be integrated in a quasi-static, dynamic control-driven manner using CFDF techniques for scheduling H as a component within its enclosing subsystem or application graph model.

For example, for the dataflow graph G_{outer} that contains H, one can readily apply a *CFDF canonical schedule*, which is a standard type of schedule for CFDF graphs that can be constructed quickly and is suitable for rapid prototyping and bottleneck identification [6]. Alternatively, existing techniques for CFDF schedule optimization (e.g., see [6]) can be applied to G_{outer} to help improve system performance or satisfy operational constraints.

3.5 Case Study: MIMO Detection

We demonstrate the utility of CF-PSDF-based implementation with a case study of soft multiple-input, multiple output (MIMO) detection.

3.5.1 Application Model based on CF-PSDF

MIMO technology has been adopted in many modern wireless communication standards, such as LTE and WiMAX, due to the significant capacity increases that can be achieved by using multiple antennas in transmitters and receivers [28]. In this case study, we implement an application of $M \times M$ MIMO detection with a P-QAM constellation. We apply an efficient soft MIMO detection algorithm called the *list fixed-complexity sphere decoder* (LFSD), which is a list-based version of the fixed-complexity sphere decoder (FSD) [29]. The LFSD generates a list of candidates around the maximum likelihood (ML) solution that can be used to calculate soft-output information for each transmitted bit b_k in the form of log-likelihoods (LLRs), $\{L_k\}$.

An $M \times M$ MIMO system is commonly decomposed into M processing layers (in our experiments, we use $M = 4$). In our design, the vector-valued parameter λ, consisting of M-elements, specifies the number of optimal detected results that are generated at each layer. If $\lambda = (n_1, n_2, \ldots, n_M)$, then the list size can be expressed as $N_L = \prod_{i=1}^{M} n_i$.

In our implementation, the soft MIMO detector takes the received symbol vector y and the channel matrix C as inputs, finds the N_L candidates for each y and

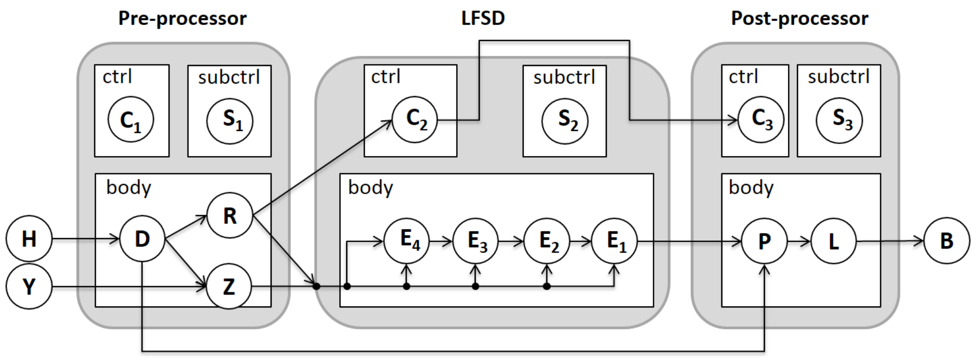


Figure 3.3: CF-PSDF model of soft MIMO detection application.

C, and generates the soft information L_k. We model the application with our proposed CF-PSDF framework, as illustrated in Figure 3.3. Here, the dataflow graph is composed of three CF-PSDF actors (`Pre-processor`, `LFSD`, and `Post-processor`), two source actors `Y` (source of y) and `H` (source of C), and one sink actor `B` (sink of L_k). The soft MIMO detector is divided into three parts: (1) the preprocessing component (`Pre-processor` actor), which applies QR decomposition on the channel and least squares estimation of the input symbols; (2) the LFSD component (`LFSD` actor), which generates a list of candidates according to the FSD algorithm; and (3) the postprocessing component (`Post-processor` actor), which computes the LLRs with the list generated by the LFSD component. On the subsystem corresponding to each CF-PSDF actor, the quasi-static scheduling technique developed in [7] is applied.

In our implementation, the list size N_L is determined dynamically for each realization (i.e., for each y and C) based on the channel quality. Usually, a large value for N_L improves the bit error rate (BER), but at the cost of increased complexity. In our design, a realization with better channel quality is processed with a smaller list to reduce computational complexity. On the other hand, a large list is used for realizations in poor channel states to improve the detection accuracy. We consider three different settings of λ in our MIMO system implementation: $(1, 1, 1, P)$, $(1, 1, 2, P)$, and $(1, 2, 2, P)$. These settings result in $N_L = P$, $N_L = 2P$, and $N_L = 4P$, respectively.

In our CF-PSDF-based design, the `LFSD` actor includes three modes, `MODE-P`, `MODE-2P`, and `MODE-4P` to output $N_L = P$, $N_L = 2P$, and $N_L = 4P$ tokens (candidates), respectively. The associated control actor `C2` of the `LFSD` actor, when fired, computes the channel quality (instantaneous channel capacity of C, denoted ρ_C) with the input data exported from the `Pre-processor` actor, and then configures the subsystem mode parameter μ (i.e., selects a list size) based on the current channel quality indicator ρ_C. In the cases of $\rho_C > \rho_{TH1}$ ("good quality") and $\rho_C < \rho_{TH2}$ ("bad quality"), `MODE-P` and `MODE-4P` are selected, respectively, while in all other cases, the mode is set to `MODE-2P`. Here, ρ_{TH1} and ρ_{TH2} are two system parameters that determine the thresholds to use for determining good and bad channel quality, as described above.

The designer-provided specification of functional modes in CF-PSDF provides significant streamlining in the space of mode combinations that need to be handled during the implementation process. The body graph of the `LFSD` actor contains four actors — E_1, E_2, E_3, and E_4 — which, respectively represent the FSD processing elements for layers 1 through 4. Each E_i has four operational modes — a `LOAD` mode for reading input tokens, and three processing modes, denoted `M-1`, `M-2`, and `M-P`, to process data for $n_i = 1$, $n_i = 2$, and $n_i = P$, respectively. The total number of actor mode combinations in the body graph is therefore $4^4 = 256$. However, it is easy for the designer to understand and specify that only three of these combinations, which correspond to `MODE-P`, `MODE-2P`, and `MODE-4P` of the LFSD subsystem, are relevant. Thus, including the required control mode, the total number of operational modes for the LFSD subsystem is reduced from 256 to only 4 using the CF-PSDF convention of designer-specified functional modes.

3.5.2 Experimental Results

Our experiments on this MIMO detector case study are performed on a PC with an Intel 3GHz CPU and 4GB RAM. First, we compare the performance of the detector modeled in pure CFDF and CF-PSDF for a 4×4 MIMO system with QPSK, 16-QAM, or 64-QAM modulation. In the experiments, both implementations apply the canonical CFDF scheduler [6]; however, for the CF-PSDF-based implementation, the results of this scheduler are integrated with SDF schedules for individual functional modes, as described in Section 3.4.

Table 1 lists experimental results for this comparison. From the results, we see that compared to CFDF, CF-PSDF modeling can significantly reduce the number of average visited actors per realization (shown in the row labeled *Visited node count*). Here, by a “visit”, we mean a basic dataflow scheduling operation that involves assessing whether an actor has sufficient input data, firing the actor if it has sufficient data, or both. This reduction in visited node count, which can be viewed as a reduction in schedule execution overhead, arises because of the novel

Table 1: Performance comparison between CFDF- and CF-PSDF-based implementations. Run time is in microseconds.

Modulation	4-QAM		16-QAM		64-QAM	
Dataflow model	CF	CF-PS	CF	CF-PS	CF	CF-PS
Visited node count	272.6	68.7	1012	151.9	3972	484.4
Improvement	74.8%		85.0%		87.8%	
Run time	0.11	0.10	0.19	0.15	0.50	0.33
Gain	9.1%		21.1%		34.0%	

support in CF-PSDF for efficient quasi-static scheduling (i.e., in terms of local SDF schedules for individual functional modes). The overall performance of the CF-PSDF implementation is correspondingly improved as well. As we see from the row labeled "Run time", the average execution time is improved by 9.1%, 21.1%, and 34.0%, respectively, for QPSK, 16-QAM, and 64-QAM.

As P increases, the run time improves more since in CFDF, more non-firing node visits occur while in CF-PSDF, such overhead is avoided through efficient quasi-static scheduling.

Next, we compare the performance of our dynamic MIMO detector against a conventional static detector with a fixed list size $N_L = 4P$ for a 64-QAM 4x4 MIMO system (i.e., $P = 64$). To evaluate the coded BER performance, we feed the soft output of the detectors to a length 3600, rate 1/2 turbo decoder with eight iterations [30]. Our experimental results show that to achieve the target BER

Table 2: Experimental comparison between the SS and DS. Run time is in microseconds.

SNR (dB)	19.0	19.5	20.0	20.5	21.0
Run time (static)	0.425	0.425	0.424	0.426	0.425
Run time (dynamic)	0.370	0.353	0.339	0.327	0.318
Gain	12.9%	16.9%	20.0%	23.2%	25.2%

(assume 10^{-4}), the static system (SS) and dynamic systems (DS) require at least 19.84dB and 19.90dB signal to noise power ratio (SNR), respectively. In exchange for this small (0.06dB) degradation, the DS provides a significant improvement in run time (RT), as shown in Table 2. As expected, the RT of the SS at various SNRs is almost uniform. By contrast, the RT for the DS improves as SNR increases. This is because higher SNR provides more opportunities for use of smaller list sizes, which results in lower computational cost.

3.6 Related Work

In addition to CFDF and PSDF, there is a variety of other models that support dynamic dataflow modeling, design, and implementation. Wiggers, Bekooij, and Smit [31] present *variable rate dataflow* (*VRDF*) to model systems with data-dependent communication, and develop techniques to compute buffer sizes for VRDF specifications for given throughput constraints. Eker et al. [32] present a hierarchical approach for modeling of heterogeneous embedded systems, including systems that incorporate dataflow behaviors. In this approach, designers employ modeling constructs called *directors* to control the communication and execution schedules for associated application subsystems. The *stream-based functions* (*SBF*) model of computation combines the semantics of dataflow and process network models for design and implementation of embedded signal processing systems [33]. An actor in SBF contains a set of operational functions, along with a controller, state, and a transition function. The use of operational functions and the transition function in SBF is analogous in some ways to the modes and next mode determination functionality in the CFDF model. Given this relationship, an interesting direction for further study is the adaptation of the techniques introduced in this thesis to SBF specifications (i.e., an integrated SBF-PSDF modeling framework).

The dataflow-based modeling and design techniques presented in this chapter differ from the related work discussed above in that our framework generalizes the CFDF and PSDF models to provide systematic, mode-based, dynamic modeling together with flexible dynamic parameter reconfiguration. Our emphasis on support for localized use of optimized static and quasi-static schedules further distinguishes our contribution in this chapter from related work in this area.

Furthermore, the CF-PSDF modeling approach proposed in this chapter can potentially reduce scheduling overhead and also provide opportunities for powerful dataflow analysis and transformation techniques. We discuss these advantages further in Chapter 4.

4 Hierarchical Mapping for Parallel Architectures

In this chapter, we present a structured design method based on MDSDF graphs for hierarchical mapping of DSP systems onto parallel architectures. Material in this chapter was published in partial, preliminary form in [34] and [35]. In various forms of data parallel programming, programmers can define functions, and have multiple calls to the functions execute in parallel on different data sets (e.g., see [11, 24]). Recent data parallel programming environments emphasize support for exploiting multi-level or *hierarchical* parallelism, where parallelism is exploited programmatically at multiple levels of granularity. For example, CUDA [11] provides a two-level thread hierarchy, where a set of threads makes up a *thread block*, and multiple thread blocks form a *grid*.

Such hierarchical support for representing parallelism is important for multidimensional signal processing applications, where parallelism exists in different forms at different levels of the *design hierarchy* (*DH*) (e.g., inter-frame, inter-block, and inter-pixel parallelism in video processing). In this chapter, we build on the MDSDF and CF-PSDF models of computation, and develop a design method to represent and apply parallelism hierarchically for multidimensional dataflow graphs. We refer to the proposed method as the *parameterized multidimensional design-hierarchy mapping* (*PMDHM*) framework.

Currently, the class of directed acyclic graphs (DAGs), i.e., directed graphs with no directed cycles [36], is targeted in this work. In other words, we assume that application dataflow graphs that are provided as input to the PMDHM framework are DAGs. A large class of useful signal processing applications conforms to the structure of DAGs. The extension of the PMDHM framework to handle classes of graph structures that include cycles is a useful topic for further investigation.

4.1 PMDHM Framework

In this section, we present our proposed PMDHM framework for dataflow-based design, which is targeted to the flexible, multi-level reconfigurability, and intensive real-time processing requirements of emerging dynamic signal processing systems.

A CF-PSDF specification is composed of three cooperating CF-PSDF graphs, the *ctrl*, *subctrl*, and *body* graphs of the specification. Actors and edges in CF-PSDF graphs can be annotated with arbitrary parameters, which can be changed at runtime. Such actors and edges correspond, respectively, to functional components and intra-component connections in signal processing flowgraphs (see Chapter 3).

The ctrl graph of a CF-PSDF subsystem $\sigma(H)$ is executed once during each firing of H and is allowed to perform data-dependent parameter updates in the associated subctrl and body graphs as presented in Chapter 3. The subctrl graph of $\sigma(H)$ can configure the parameters in the associated body graph in ways that do not change the dataflow rates at the interfaces of H.

For selected subsystems in a CF-PSDF-based system design, a new design transformation called the *multi-level hierarchical dataflow transformation* (*MHDT*) can be employed to efficiently map the subsystem to a given target platform that employs parallelism at multiple levels of platform architecture. Designers can thus select subsystems that have critical constraints (e.g., on performance, energy efficiency or resource utilization) for application of the MHDT.

For each alternative body graph that results from different sets of parameter configurations (e.g., application or subsystem modes) in the ctrl graph, the MHDT approach transforms an application graph with parameterized production and consumption rates (i.e., dataflow rates that are represented as functions of system parameters) into a hierarchical organization of graphs such that the structure of the hierarchy helps the designer to map the design onto the hierarchical parallel structures in the target platform.

4.1.1 Multi-level Hierarchical Dataflow Transformation

Let $G = (V, E)$ denote an MDSDF graph where $V = \{v_1, v_2, \ldots, v_L\}$ is a set of vertices (*actors*), and $E = \{e_1, e_2, \ldots, e_K\}$ is a set of directed edges, which represent communication between actors according to MDSDF semantics. In MDSDF graphs, actor firings are indexed (in their associated "firing spaces") by n-dimensional vectors, where the values of n depend on the dimensions of the data that are produced and consumed ($n = 1$ corresponds to conventional single-dimensional, SDF-like firing sequences) [10].

Suppose that v is an MDSDF actor with a firing space of M dimensions, and let $r_{v,i}$, for $i = 1, 2, \ldots, M$, denote the size of the ith dimension of the firing space for v in a given periodic schedule S for G. A periodic schedule is a sequence of

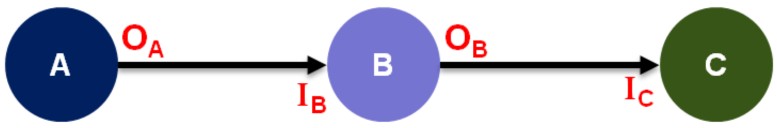


Figure 4.1: An example of a three-actor MDSDF graph.

actor firings that executes each actor at least once and produces no net change in the numbers of tokens queued on the edges of G [9, 10]. We refer to the M-vector $r_v = [r_{v,1}, r_{v,2}, ..., r_{v,M}]$ as the *firing vector* for actor v associated with S. The product of the M elements of this vector gives the total number of firings of v within S. For a properly constructed MDSDF graph, r_v can be computed by solving a system of equations called the *balance equations* for the graph [10].

Consider, for example, the 3-node graph illustrated in Figure 4.1. The firing vectors r_A, r_B, and r_C can be found by solving the following balance equations for $i = 1, 2, \ldots, M$:

$$r_{A,i}O_{A,i} = r_{B,i}I_{B,i}, \quad r_{B,i}O_{B,i} = r_{C,i}I_{C,i}, \tag{4.1}$$

where $I_X = [I_{X,1}, I_{X,2}, \ldots, I_{X,M}]$ and $O_X = [O_{X,1}, O_{X,2}, \ldots, O_{X,M}]$ are the M-dimensional consumption and production rates, respectively, for actor X.

Now suppose that we have an N-level hierarchical parallel programming model (*platform hierarchy*) P, which we want to use to implement a given MDSDF graph G. For example, such a parallel programming model could be used as a target for code generation or could be used for an implementation that is derived from hand based on a functional reference ("golden model") that is based on the MDSDF specification. We develop an N-level hierarchical dataflow graph transformation approach to achieve such a mapping from an MDSDF-based application graph to

P. We refer to N in this context as the *platform depth*.

First, we introduce some definitions and notation related to hierarchical dataflow graphs. For a dataflow graph $G = (V, E)$, let $P_i(V)$ and $P_o(V)$ be the sets of input and output ports of all actors in V, respectively. A *supernode* s in G is an actor (i.e., $s \in V$) that is associated with a "nested dataflow graph" $H(s)$, where execution of s in G corresponds to execution of $H(s)$. In general, not all actor ports in $H(s)$ are connected in $H(s)$ (i.e., not all of them connect to edges within $H(s)$). The "unconnected actor ports" are referred to as the *interface ports* of $H(s)$, and these ports are in one-to-one correspondence with ports of actor s.

If G is the "top" of the design hierarchy (i.e., G is not encapsulated by a supernode in another graph), then we say that the *nesting level* (or simply *level*) of G, denoted $\lambda(G)$, is 1. Similarly, for each supernode s in G, $\lambda(H(s)) = 2$; for each supernode t in any of these $H(s)$'s, $\lambda(H(t)) = 3$, and so on.

The design hierarchies in our model are non-overlapping, which means that for all supernodes within a design hierarchy (i.e., across all levels), their corresponding nested dataflow graphs do not share any actors or edges. Furthermore, we assume that these design hierarchies are finite, which means that the levels (λ values) are all bounded.

We refer to the maximum λ value in a design hierarchy D as the depth δ of D. For each $i \in \{1, 2, \ldots, \delta\}$, we denote by L_i the set of all actors that are "at level i". That is, $L_1 = V$, and for $i = 2, 3, \ldots, \delta$,

$$L_i = \cup\{V_h(s) | \lambda(H(s)) = i\}, \tag{4.2}$$

where $V_h(s)$ denotes the set of actors in the nested dataflow graph $H(s)$.

Design hierarchies in our decomposition approach can be constructed by designers as they explore alternative methods to structure the hierarchies such that they map efficiently into the parallelism hierarchy supported by the targeted platform. The key constraint in the construction of a design hierarchy D is that the depth of each candidate design hierarchy should equal the platform depth. In Section 5.1 and 5.2, we illustrate how a design hierarchy can be constructed naturally from understanding of the flowgraph structure of an application. However, design hierarchies can also be targeted by automated tools. Exploration of such automated design hierarchy construction tools is a useful topic for future work.

We have developed a systematic method, called the *multi-level hierarchical dataflow transformation* (*MHDT*), to specify and map design hierarchies into hierarchies of smaller graphs, which can in turn be mapped to successively lower levels of the targeted platform hierarchy. Figure 4.2 illustrates this approach for an MDSDF graph. The designer can construct the design hierarchies in a bottom-up or top-down fashion. At each ith level ($i > 1$) of the design hierarchy, one or more groups (*clusters*) of connected actors are combined into units that are viewed as individual supernodes from level ($i - 1$). Groups of actors, including supernodes, that are contained within such clusters are then scheduled together by adapting techniques for SDF- and MDSDF-based clustered graph analysis and scheduling [37, 10].

When applying the MHDT, each supernode s at each level i is transformed into a corresponding "standalone" dataflow graph, which is referred to as a *mapping cluster*. The transformation of a supernode into a mapping cluster is performed through the following process.

In each mapping cluster, two special interface actors, `ii` (interface input) and `io` (interface output) actors, are inserted. These actors represent interfaces to the enclosing supernodes and serve to inject data from input edges and to output edges of the supernodes, while providing standalone dataflow graph representations for each level of the design hierarchy. Using these standalone representations, buffer management and scheduling can be performed to ensure correct, consistent execution while mapping the actors in each design hierarchy level L_i into the corresponding ith level of the targeted parallel platform.

Each mapping cluster, when executed, is assumed to fire the interface actors only once. The derivation of the production and consumption rates associated with the interface input and output actors, in general, depends on the characteristics of supernodes (subsystems). Nevertheless, the valid values of the rates should satisfy the condition that there exists a non-trivial solution of the balance equations for the associated mapping cluster given that the `ii` and `io` actors only fire once. This condition is important since it guarantees that a valid period schedule can be found for the mapping cluster.

Presently, we compute these rates by hand, as our emphasis in this work is on demonstrating the overall design methodology and its utility on practical case studies. However, the process can readily be automated since it is based on formal dataflow principles. Development of automated tool support for the design methodology proposed in this chapter is a useful direction for further work.

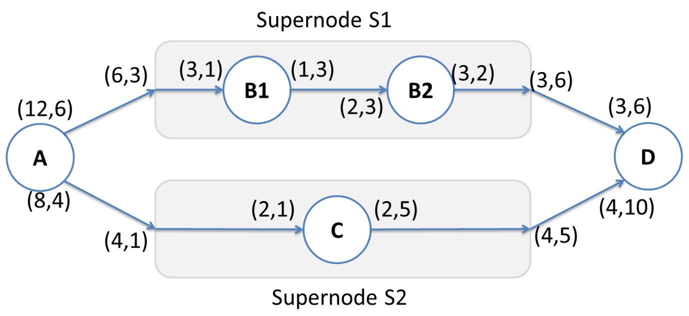
For a mapping cluster C, *virtual edges* (edges with zero rates of consumption and production) are added between source (sink) actors and the `ii` (`io`) actor to connect these actors with the interface actors, where the source (sink) actors are those actors without input (output) edges. For each source actor `src` in C, a virtual edge is created to connect `src` from the `ii` actor; Similarly, for each sink actor `snk` in C, a virtual edge is created to connect `snk` to the `io` actor. The original mapping cluster together with the actors `ii` and `io`, and the corresponding set of virtual edges is referred to as the *augmented mapping cluster graph* (*AMCG*). Given a mapping cluster C, the corresponding AMCG is denoted as $AMCG(C)$.

The virtual edges are added to augment the mapping cluster such that for each actor α in C, there exists a path in the AMCG from `ii` to `io` that traverses α. This condition is important for further partitioning and transformation techniques, which are introduced in Section 4.1.2.

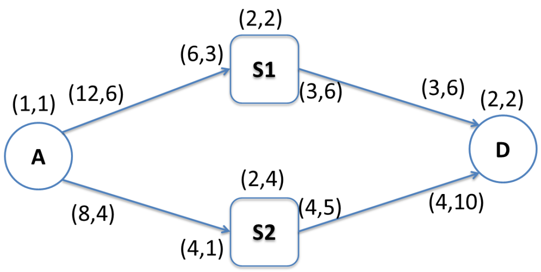
The mapping clusters constructed using this process are used for efficient mapping of flowgraph structures into architectures that employ multi-level parallelism. Such architectures, such as graphics processing units (GPUs) and CBEA-compliant processors [38], are becoming increasingly important in the realization of computationally-intensive signal processing systems.

4.1.2 Partitioning of Mapping Cluster Graphs

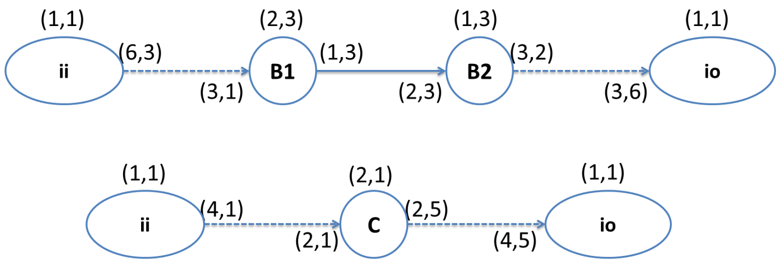
In this section, we present techniques for graph partitioning that use the mapping clusters generated by the MHDT approach described in Section 4.1.1. These partitioning techniques provide further transformations to the application dataflow



(a) The overall graph.



(b) The top-level graph.



(c) The level-2 mapping clusters.

Figure 4.2: An example of a design hierarchy for an MDSDF specification.

graph that facilitate analysis and exploitation of parallelism across different levels of the targeted platform hierarchy.

Let $V(C), E(C)$ denote, respectively, the sets of actors and edges in $AMCG(C)$. The graph $AMCG(C)$ is partitioned through the following process.

First, $AMCG(C)$ is partitioned into n *channels* $X_1, X_2, \ldots, X_n$, where $n \geq 1$. Each channel X is an ordered pair $X = (\mathit{actors}(X), \mathit{edges}(X))$, where $\mathit{actors}(X) \in V(C)$ and $\mathit{edges}(X) \in E(C)$, and $\mathit{actors}(X), \mathit{edges}(X)$ are *mutually connected.* A set P of actors and a set Q of edges are mutually connected if for each $e \in Q$, we have that $\mathit{src}(e) \in P$ *and* $\mathit{snk}(e) \in P$, where $\mathit{src}(e)$ represents the source actor of edge e and $\mathit{snk}(e)$ represents the sink actor of edge e. For the partition into channels $X_1, X_2, \ldots, X_n$ to be *valid* in this context, the edge sets $\mathit{edges}(X_1), \mathit{edges}(X_2), \ldots,$ $\mathit{edges}(X_n)$ must be disjoint. Furthermore, in a valid partition $X_1, X_2, \ldots, X_n$, each channel contains *both* `ii` and `io` (i.e., each $\mathit{actors}(X_i)$ contains both of these actors), and beyond these two common actors, no other actor in $AMCG(C)$ is contained in multiple channels.

Thus,

$$\cap_{i=1}^{n} \{\mathit{edges}(X_i)\} = \{\emptyset\}, \tag{4.3}$$

and

$$\cap_{i=1}^{n} \{\mathit{actors}(X_i)\} = \{\texttt{ii}, \texttt{io}\}. \tag{4.4}$$

We refer to this kind of graph partition as a *channel partition* of the associated

AMCG. By definition, a channel partition ensures that, except through the interface actors, there is no path that connects any pair of distinct channels, and therefore, all channels can be executed in parallel once the input tokens of the associated supernode are injected into the associated mapping cluster. We refer to each X_i as a *channel* of the mapping cluster C (or of $AMCG(C)$).

Algorithm 1 outlines a partitioning process, which we refer to as the *AMCG partitioning algorithm*, that we have developed for strategic derivation of channel partitions.

After channel partitioning, the actors in each channel X are further partitioned into $\rho_1(X), \rho_2(X), \ldots, \rho_{parts(X)}(X)$ in such a way that all actors in each $\rho_i(X)$ are *ready to fire* (i.e., all input tokens are available for a given iteration of the associated mapping cluster) immediately after all actors in the preceding partition components $(\rho_1(X), \rho_2(X), \ldots, \rho_{i-1}(X))$ have *fired completely.* In this context, by firing completely, we mean that the associated actor A has finished $\nu(A)$ firings (i.e., a single- or multi-dimensional "volume" of firings as represented by the vector $\nu(A)$) in the current iteration of the enclosing mapping cluster, where $\nu(A)$ represents the firing vector of the actor A. At the beginning of this partitioning process, the interface input actor `ii` is initialized as being fired completely.

This process results in a unique partition $\rho_1(X), \rho_2(X), \ldots, \rho_{parts(X)}(X)$ of each channel X, which we refer to as the *pipeline partition* of the channel. The pipeline partition provides a decomposition of a channel into a flowgraph pipeline consisting of $parts(X)$ "stages", where all actors within a given pipeline stage $\rho_i(X)$ can be fired simultaneously for a given mapping cluster iteration once the previous stages

Algorithm 1 Outline of the AMCG partitioning algorithm.

```
// The AMCG partitioning process for a mapping cluster C = (V,E).

// This process generates a finite sequence of channels {X(i) = (V(i),E(i))}.

Vb = V.Remove(ii,io);   // The set of all non-interface vertexes in V.

Eb = E;

i = 1;

while (Vb is non-empty) {

    v = Vb[1];          // The first vertex in Vb.

    V(i) = {v};         // Initialize the vertex set for a new channel.

    Vb.Remove(v);       // Remove v from Vb.

    E(i) = {};

    EC = {};

    EC =  The set of all edges in Eb that are incident to any vertex in V(i);


    while (EC is non-empty) {

        VC = {all vertexes in Vb that are incident to edges in EC};

        V(i).Add(VC);       // Add all vertexes in VC to V(i).

        Vb.Remove(VC);      // Remove all vertexes in VC from Vb.

        E(i).Add(EC);       // Add all edges in EC to E(i).

        Eb.Remove(EC);      // Remove all edges in EC from Eb.

        EC = The set of all edges in Eb that are incident to any vertex in V(i);

    }

    V(i).Add(ii,io);    // Add the interface actors to the channel.

    i++;                // Next partition.

}
```

(stages $1, 2, \ldots, (i-1)$) have been completed.

This partitioning process, which we refer to as *channel pipeline partitioning*, is illustrated by the pseudocode shown in Algorithm 2. Here, a vertex a_1 is said to be a *successor* of a vertex a_2 if there is an edge that is directed from a_2 to a_1.

Algorithm 2 Outline of the channel pipeline partitioning process.

```
// Pipeline partitioning process for channel X = (V,E).
// This process divides V into V(j), j = 0, 1, ... where
// V(j) is the j-th pipeline stage in the pipeline partition of channel X.


VC = V;
VF = {ii};       // The set of all vertices in VC that are ready-to-fire.


V(0) = {ii};     // The initial stage.
VC.Remove(ii);   // Remove the partitioned vertexes from VC.


j = 1;
while (VC is non-empty) {
    V(j) = {};          // A new pipeline stage.
    VR = The set of vertices in VC that are successors of vertices in VF;
    V(j).Add(VR);       // Add the vertices in VR to the current stage.
    VF.Add(VR);         // Add vertices in VR (ready-to-fire) to VF.
    VC.Remove(VR);      // Remove vertices in VR from VC.
    j++;                // Next stage.
}
```

Through these partitioning steps, a mapping cluster C is transformed to an intermediate MDSDF graph representation that we call the *pipelined AMCG*. The pipelined AMCG provides a compact, graphical representation of application structure that exposes data parallelism, and facilitates further analysis for mapping and performance optimization. In the construction of the pipelined AMCG, each channel X is transformed into a pipelined AMCG actor $\delta(X)$. Each such actor $\delta(X)$ has a single input edge e_{in}, which is directed from the interface input actor `ii`, and a single output edge e_{out}, which is directed to the interface output actor `io`. This is illustrated in Figure 4.3(b).

In a pipeline partition $\rho_i(X)$ of a channel X, since all actors are ready to fire immediately after the execution of the previous pipeline stages, the degree of parallelism (the maximum number of parallel actor firings) in each dimension k is given by

$$\hat{r}_{i,k}(X) = \sum_{v \in \rho_i(X)} \mathit{fvect}_{v,k}, \tag{4.5}$$

where $\mathit{fvect}_{v,k}$ represents the firing vector component associated with the kth dimension for actor v.

The kth entry of the firing vector for the pipelined AMCG actor $\delta(X)$ (transformed from channel X) can be represented as

$$\mathit{fvect}_{\delta(X),k} = \phi(\hat{r}_{1,k}(X), \hat{r}_{2,k}(X), \ldots, \hat{r}_{\mathit{parts}(X),k}(X)), \tag{4.6}$$

where $\phi : Z_{pos}^{\mathit{parts}(X)} \rightarrow Z_{pos}$ is a mapping from the set of $\mathit{parts}(X)$-tuples of positive integers to the set of positive integers (Z_{pos} represents the set of positive integers). The function ϕ is a design parameter in this formulation.

An optimal setting for ϕ depends on various factors, including the application dataflow, as well as characteristics of the target platform. A simple heuristic method for configuring ϕ is to derive ϕ in terms of the maximum degrees of parallelism across all of the pipeline stages encapsulated by X. That is,

$$\phi(a_1, a_2 \ldots, a_m) = \max_{1 \le j \le m} a_j. \tag{4.7}$$

The usage of the *maximum* function here allows all pipeline stages to have sufficient numbers of parallel threads to carry out all actor firings concurrently, provided that there are sufficient hardware resources available. For pipeline stages with smaller degrees of parallelism, the additional threads are simply redundant placeholders that remain *inactive* (idle). As a cost for exploiting parallelism in this greedy manner, these idle threads can cause execution overhead. A potential way to reduce such overhead is to employ a suitable fine-grained, actor clustering strategy for design hierarchies. We discuss this kind of clustering process further in Section 5.1.

4.1.3 Deriving Dataflow Rates for Intra-Channel Edges

In this section, we discuss the derivation of production and consumption rates for edges that are contained in channels. For a given channel X, let $E_I(X) \in \mathit{edges}(X)$ denote the subset of edges that have `ii` as the source actor; similarly, let $E_O(X) \in \mathit{edges}(X)$ denote the subset of edges that have `io` as the sink actor. In the pipelined AMCG, the consumption rate of each edge $e_{in,i}$ in each dimension k is derived as the total amount of dataflow (within a given mapping cluster iteration) from the interface input actor to the channel X. That is,

$$c_{e_{in,i},k} = \sum_{e \in E_I(X_i)} c_{e,k}. \tag{4.8}$$

Similarly, the production rate of each output edge $e_{out,i}$ in each dimension k is derived as the summation of the production rates of all edges in $E_O(X)$:

$$p_{e_{out,i},k} = \sum_{e \in E_O(X_i)} p_{e,k}. \tag{4.9}$$

To ensure the existence of non-trivial solutions for the balance equations of the pipelined AMCG graph, the production rate of each $e_{in,i}$ and the consumption rate of each $e_{out,i}$ are accordingly set to

$$p_{e_{in,i},k} = \frac{c_{e_{in,i},k}}{fvect_{\delta(X),k}}, \tag{4.10}$$

and

$$c_{e_{out,i},k} = \frac{p_{e_{out,i},k}}{fvect_{\delta(X),k}}. \tag{4.11}$$

Recall from Section 4.1.2 that $\delta(X)$ represents the pipelined AMCG actor that represents the channel X.

From Equation 4.10 and Equation 4.11, observe that production and consumption rates in the pipelined AMCG can be non-integer-valued, which is different from conventional MDSDF graphs, and has some relationships to the concept of fractional rate dataflow graphs [39]. However in our application of pipelined AMCGs to dataflow graph scheduling, we derive positive integer firing vectors through application of certain transformations. This is discussed further in Section 4.2.

4.1.4 An Example of Pipelined AMCG Construction

An example of pipelined AMCG derivation is illustrated in Figure 4.3. This illustration demonstrates that the pipelined AMCG takes the form of a simple pattern that can be described using efficient techniques, such as topological patterns [12] for compact graph expression, as well as for integration with standard dataflow-based design representation techniques (e.g., see [40, 41]). Additionally, our proposed partitioning methods provide a systematic framework for analyzing separate components of the partitions separately (e.g., individual pipelined AMCGs) to lower overall analysis complexity through a divide and conquer approach. Furthermore, this framework allows designers a method for identifying relevant parts of a schedule to extract specific details (e.g., idle threads) as suggested by feedback from experiments. Such features of our proposed mapping cluster partitioning framework are illustrated concretely in Section 5.1.

We also emphasize here that the concept of "pipelining" represented in the pipelined AMCG and related aspects of mapping clusters is an abstract form of pipelining and does not relate directly to any specific form of hardware pipeline organization or scheduling technique. In particular, pipelined AMCGs represent the decomposition into sequences of "linearly-dependent" dataflow subgraphs of functionality within MDSDF-based application representations. In this sense, pipelined

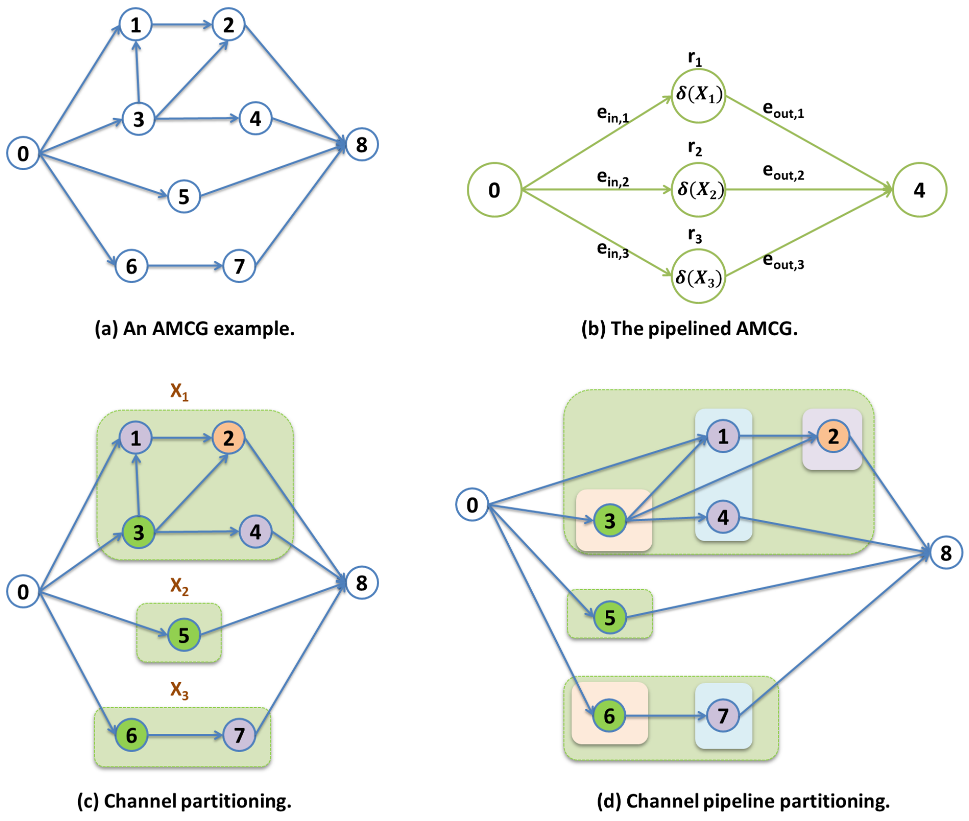


Figure 4.3: An example of pipelined AMCG derivation.

AMCGs can be viewed as a form of macro-pipelining for signal processing dataflow graphs, as discussed, for example, in [42].

4.1.5 Graph Decomposition Example

In Figure 4.4 through Figure 4.8, we illustrate through an example the relationships among application dataflow graphs, PMDHM clustered graphs, AMCGs, channel partitions, pipeline partitions, and pipelined AMCGs. This decomposition shows different abstractions of a list fixed-complexity sphere decoder application, which is an important application in wireless communications [29]). The different abstractions shown in Figure 4.4 through Figure 4.8 are based on the different kinds of representations applied and defined in this section. These illustrations also introduce (by example) a new representation, which we refer to as the *PMDHM clustered graph*, where each mapping cluster corresponds to a single vertex or equivalently, each kernel corresponds to a single vertex. The PMDHM clustered graph can be viewed as the top level hierarchical representation used in our proposed PMDHM-based design flow. This illustration of a PMDHM clustered graph (Figure 4.5(b)) shows a decomposition based on application of a single kernel. In Section 5.1 of this thesis we provide more details on this example as well as experimental results based on its implementation. We also examine in Section 5.1 an alternative decomposition for this application that involves multiple kernels.

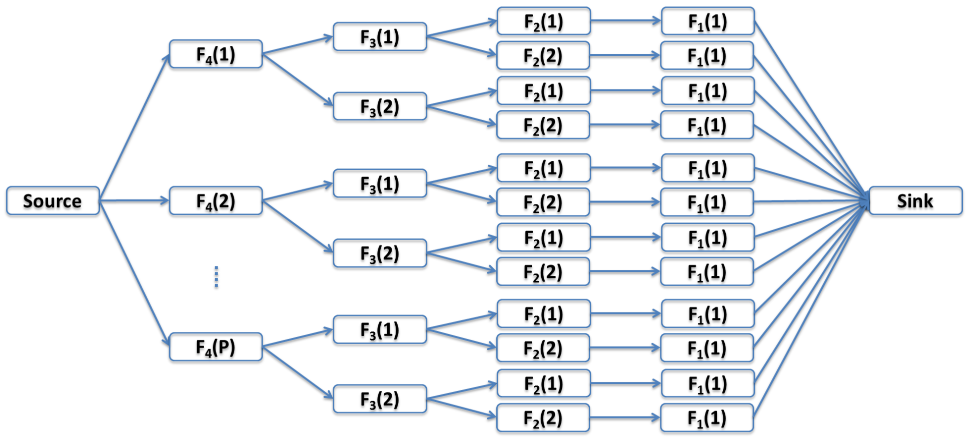


Figure 4.4: The application graph of (1,2,2,P) LFSD subsystem.

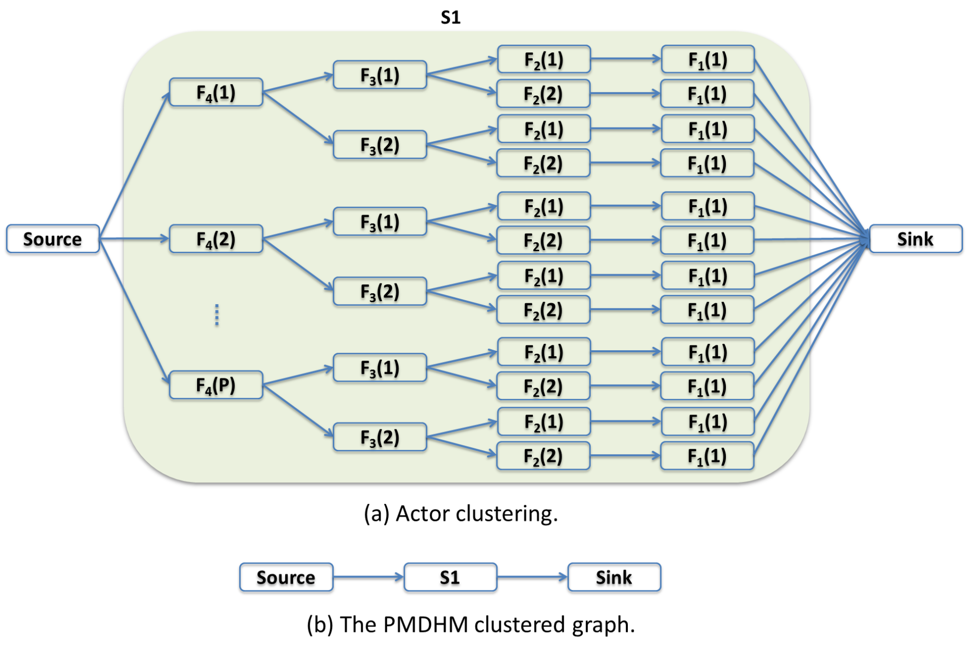


Figure 4.5: The actor clustering and the PMDHM clustered graph.

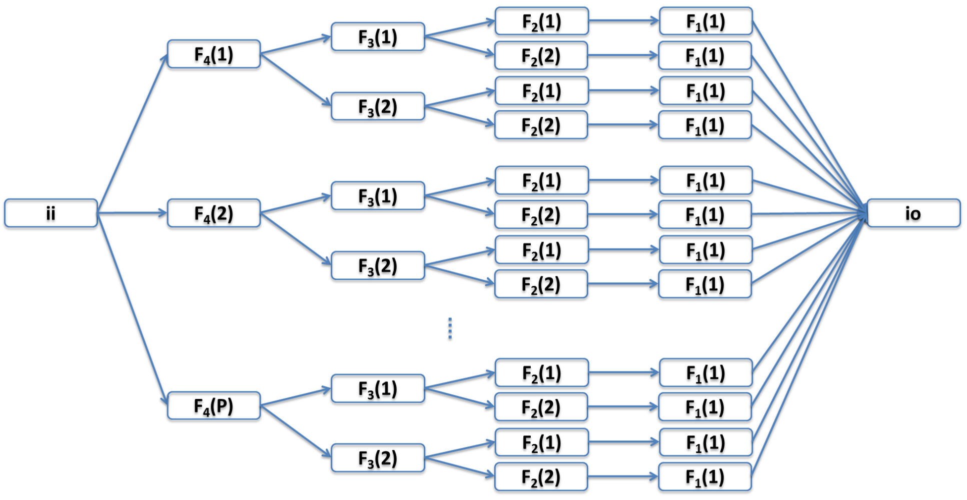
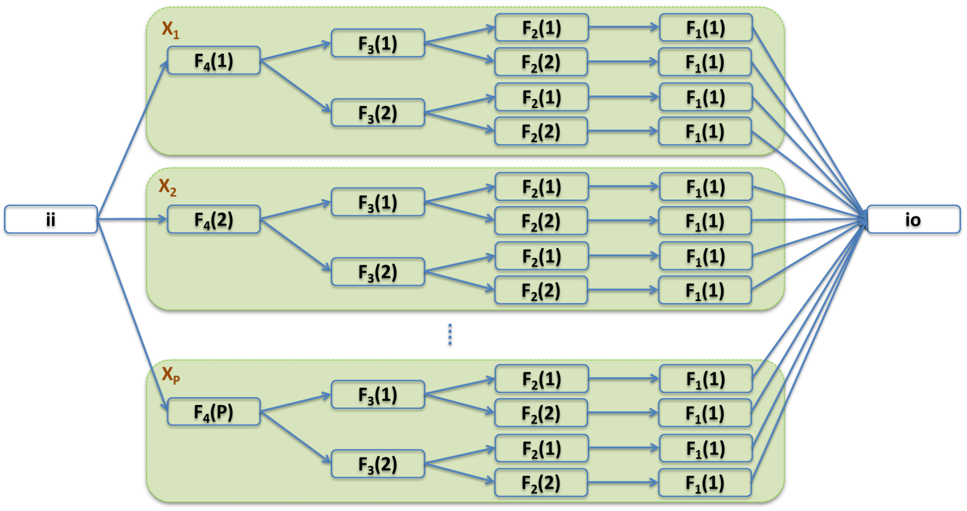
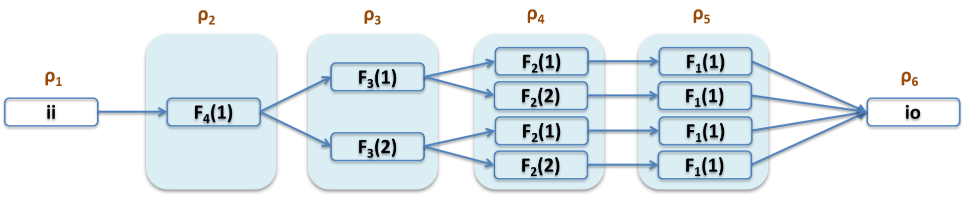


Figure 4.6: The AMCG associated with supernode S1.



(a) The channel partition.



(b) A pipeline partition for a single channel.

Figure 4.7: The channel and pipeline partitions for the LFSD mapping cluster.

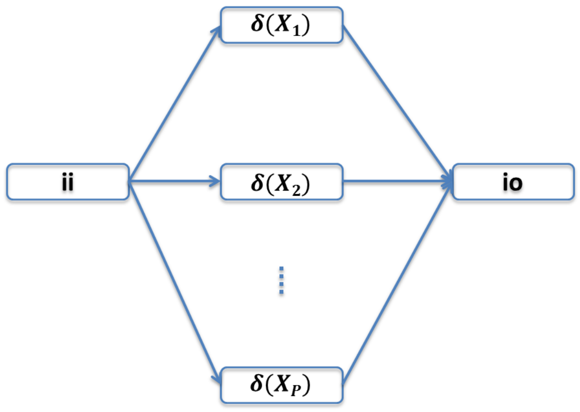


Figure 4.8: The pipelined AMCG associated with supernode S1.

4.2 Mapping Process and Optimization

4.2.1 Overview

In Section 4.1, we have introduced processes to construct a top-level graph (the PMDHM clustered graph) and the associated pipelined AMCGs, which explicitly expose application parallelism at multiple levels from MDSDF specifications. In this section, this top-level graph representation and the associated pipelined AMCGs are utilized to derive optimized parameters for mapping applications onto multi-level parallel platforms. We present in detail the mapping process for a hierarchical architecture of platform depth equal to two, and specifically for CUDA-based GPU implementation. However, our mapping process has been developed with an objective of facilitating adaptation to larger platform depths, and to supporting implementation on other instances of multi-level parallel platforms (beyond CUDA-based GPU implementation). Development of such adaptations is a useful direction for future investigation.

In a two-level parallel programming model, such as the CUDA or OpenCL programming model, we refer to top-level execution units as *kernels*. Kernels require user-specified information on *parallel dimensions* — i.e., a M-tuple vector for the degree of top-level parallelism (*grid size*), and another M-tuple vector for parallelism at the second (lower) level (*block size*) where M is supported up to three in CUDA. In such a two-level parallel programming model, we denote the parallel dimensions for kernel K as $\lambda_K = (\lambda_{K,1}, \lambda_{K,2})$, where $\lambda_{K,1}$ and $\lambda_{K,2}$ are the M-dimensional grid and block sizes, respectively.

Presently, the PMDHM framework requires that M, the dimensionality of the production and consumption rates in the MDSDF application graph, is less than or equal to the dimensionality p of the targeted parallel programming model. Thus, for example, in our targeting of CUDA, the application graph must satisfy $M \leq 3$. Extending the PMDHM framework with additional transformations that allow for $M > p$ is a useful direction for further investigation. However, note that $M \leq 3$ covers a broad class of important signal processing applications, including applications in wireless communications (where typically $M = 1$), image processing (where typically $M = 2$), and video processing (where typically $M = 3$).

In the remainder of this chapter, we focus on CUDA as the lower level (actor-level) programming model for development of our mapping process onto the targeted class of two-level architectures. We maintain this focus on CUDA for concreteness and because our experiments, described in Section 5.1 and Section 5.2, have been developed using CUDA-enabled NVIDIA GPUs as the target platforms. However, we envision that the mapping approaches described here can be readily adapted to other kinds of multi-level parallel architectures, such as those supported by OpenCL. Development and demonstration of of such adaptations is a useful direction for future work.

In the process of mapping an application dataflow graph into a CUDA-enabled GPU implementation, it is important to carefully derive the dimensions for the different levels of parallelism — inefficient configuration of such mapping parameters can degrade performance [23]. In the remainder of this section, we develop a systematic approach, based on application of pipelined AMCGs, for deriving mapping parameters in CUDA-based GPU implementation. We refer to this mapping approach as the *PMDHM supernode transformation.* The PMDHM supernode transformation includes methods for increasing the diversity of the design space of parallel dimensions through a novel transformation technique for adapting dataflow within AMCGs.

4.2.2 PMDHM Supernode Transformation

Given an MDSDF graph G, we denote the firing vector (defined in Section 4.1.1) for an actor α in the graph as as fvect_{α}. Suppose that A is a supernode in the top-level graph (the PMDHM clustered graph) G with firing vector $\mathit{fvect}_A = \theta_A$. Suppose also that the associated pipelined AMCG (denoted C_A) has n non-interface nodes, $A_1, A_2, \ldots, A_n$, with firing vector $\mathit{fvect}_{A_i} = \theta_{A_i}$ for each A_i.

Intuitively, using the PMDHM supernode transformation, we can parameterize fvect_A and each fvect_{A_i} by transforming the dataflow of A and C_A, while keeping the same total amount of dataflow for A in a periodic schedule iteration. To apply the PMDHM supernode transformation, suppose that $\theta_{A,k}$ (the component of θ_A in the kth dimension) can be factorized such that

$$(d_{A,k} \mid \theta_{A,k}) = q_{A,k}, \tag{4.12}$$

where the notation $(a \mid b) = c$ is used to indicate that a is a positive divisor of a positive integer b with the associated quotient $c = b/a$ (the remainder of this division operation is zero since a is a divisor).

Now suppose that $d_{A,1}, d_{A,2}, \ldots, d_{A,M}$ is a sequence of M values such that each $d_{A,i}$ is a positive divisor of $\theta_{A,i}$. Recall that M in this context represents the number of dimensions in the associated multidimensional dataflow (production or consumption) rates. With a minor abuse of notation, we define the vector $d_A = [d_{A,1}, d_{A,2}, \ldots, d_{A,M}]$, and we define the set of possible d_As (based on all possible combinations of positive divisors) as D. We then introduce a parameter *qtvect*, called the *quotient vector* parameter of the given supernode A, such that *qtvect* has D as its domain (set of admissible parameter value settings). The quotient vector can be viewed as a parameter for transforming dataflow in a supernode A such that for each dimension k, A consumes from each input port and produces onto each output port $d_{A,k}$ times as many tokens as in the original settings (i.e., before application of the transformation). We refer to this transformation in terms of the *qtvect* parameter as the *PMDHM supernode transformation.* Upon application of the PMDHM supernode transformation, the supernode A is replaced by (transformed to) a new supernode A' with a firing vector defined by

$$fvect_{A',k} = q_{A,k} = \theta_{A,k}/d_{A,k}, k \in 1, 2, \ldots, M, \tag{4.13}$$

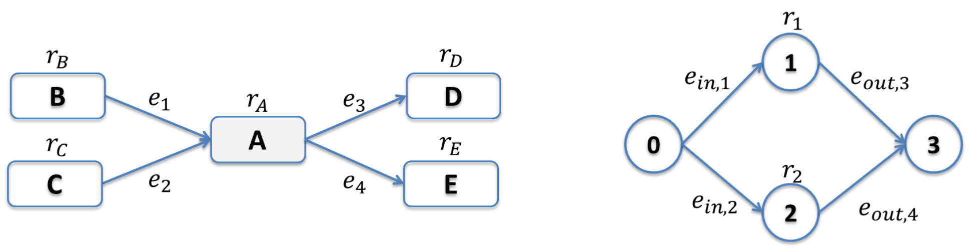
where d_A is the applied configuration (setting) of the *qtvect* parameter.

Such change of dataflow at different levels corresponds to applying a blocking factor [9] of $d_{A,k}$ to the periodic scheduling of C_A for each kth dimension. In the pipelined AMCG of A', the interface actors (`ii` and `io`) are still fired once per iteration period of the resulting schedule (constructed based on the applied blocking factors). From the application of the parameter *qtvect*, both the production rates of `ii` at all of its output ports and the consumption rates of `io` at all of its input ports are accordingly multiplied by $d_{A,k}$ in each dimension k. In addition, each non-interface actor α in A' has the same consumption and production rates at its input and output ports as the rates for the corresponding actor in A. However, the firing vector for each actor is changed based on the applied setting d_A of *qtvect*. The new firing vector is computed as:

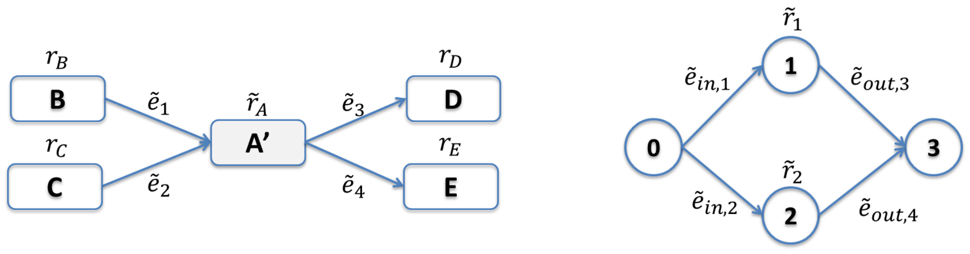
$$fvect_{\alpha,k} = d_{A,k} \times \theta_{\beta,k} \tag{4.14}$$

for each dimension k, where β is the actor corresponding to α in A (i.e., in the original supernode).

Figure 4.9 shows an example of the PMDHM supernode transformation. This example applies the PMDHM supernode transformation to the supernode A, which contains two non-interface vertices in the associated pipelined AMCG. Note that



(a) An example of a PMDHM clustered graph (left) and the pipelined AMCG of A (right).



(b) The graphs after the PMDHM supernode transformation.

Figure 4.9: An example of the PMDHM supernode transformation.

the firing vector and dataflow rates can be different before (Figure 4.9(a)) and after (Figure 4.9(b)) the PMDHM supernode transformation, as described above.

The configuration of the quotient vector parameter *qtvect* affects the amount of parallelism exploited in both the top and bottom levels of the targeted two-level parallel architecture. Thus *qtvect* should be set carefully to optimize performance.

If a supernode A is targeted to a kernel K_A on an two-level parallel platform, the parallel dimensions of K_A are derived from the PMDHM supernode transformation. Intuitively, the level-1 parallel dimension (i.e., the kernel size) $\lambda_{K_A,1}$ of kernel K_A is configured as the firing vector (parallel degree) of its associated supernode A in the PMDHM clustered graph.

$$\lambda_{K_A,1} = q_A, \tag{4.15}$$

where $q_A = [q_{A,1}, q_{A,2}, \ldots q_{A,M}]$, and recall from Equation 4.13 that for each i, $q_{A,k} = \theta_{A,k}/d_{A,k}$.

The block size of K_A is derived from the pipelined AMCG of A. Since all non-interface nodes in the pipelined AMCG can be executed in parallel, the bottom-level parallel dimension (i.e., the block size) $\lambda_{K_A,2}$ is set as the total parallel degree (i.e., the summation of parallel degrees over the associated non-interface nodes):

$$\lambda_{K_A,2,k} = \sum_{i=1}^{n} d_{A,k}\theta_{A_i,k} \tag{4.16}$$

for each dimension k. Such a summation-based setting of parallel dimensions may possibly worsen the overhead due to idle threads, as described in Section 4.1.2, especially for large M (e.g., when we are developing multidimensional signal processing applications involving signals of high dimensionality). A possible direction for addressing this issue is to apply dataflow transformations in different dimensions of the multidimensional production and consumption rates to avoid redundant allocation of threads. Exploration of such transformations is an interesting direction for future work.

In summary, the two-level mapping dimensions for a kernel K_A mapped from a supernode A are parameterized as shown in Equation 4.15 and Equation 4.16. Strategic configuration of the quotient vector parameter *qtvect* can aid in the optimization of specific metrics, such as throughput and latency. This is because different settings of *qtvect* in general give rise to different sets of parallel dimensions, which in turn affect metrics such as kernel execution times and buffer memory re-

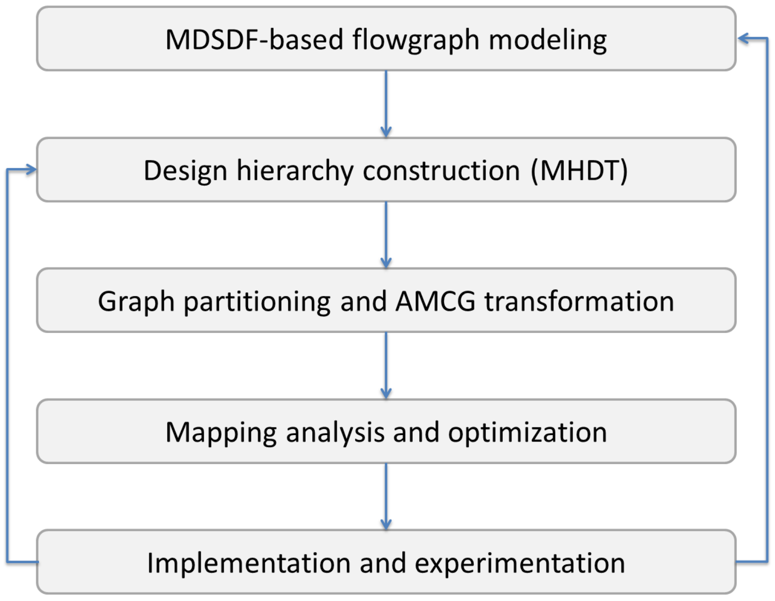

MDSDF-based flowgraph modeling
Design hierarchy construction (MHDT)
Graph partitioning and AMCG transformation
Mapping analysis and optimization
Implementation and experimentation

Figure 4.10: Illustration of our proposed PMDHM-based design methodology for design and implementation of signal processing systems.

quirements.

If kernel runtimes can be estimated through methods such as actor profiling or runtime prediction models (e.g., see [43, 44]), then the results of such estimations can be used to help configure *qtvect*. A specific approach to such estimation-driven configuration of *qtvect* is discussed in Section 5.1.

Figure 4.10 summarizes the developments of this section with an illustration of our proposed PMDHM-based design methodology for design and implementation of signal processing systems. In Section 5.1 and Section 5.2, we demonstrate applications of this new design methodology to relevant practical applications.

4.3 Related Work

A variety of dataflow based tools has evolved in recent years for design and implementation of signal processing systems (e.g., see [1, 45, 46]). In this section, we summarize a number of recent efforts beyond MDSDF that have focused especially on multidimensional dataflow modeling.

Keinert et al. propose an extension of MDSDF, called windowed synchronous dataflow (WSDF) [47]. WSDF allows modeling of sliding window algorithms for a multidimensional applications. Array–OL [48] is a language devoted to applications that involve multidimensional signal processing. Two levels of description are used for modeling parallelism in Array–OL — one is the global model for defining task parallelism, while the other is the local model for expressing data parallelism. Blocked dataflow (BLDF) [49] provides meta-modeling semantics that can be used to represent block-based and multidimensional processing in terms of different specialized dataflow models.

McAllister et al. [50] augment the MDSDF model with parameterized array expressions. Their modeling approach, called Multidimensional Arrayed Synchronous Dataflow (MASD), provides graph range parameters to control token dimensions at input and output ports. These parameters enable systematic trade-off exploration between actor network size and token size.

Additionally, a number of GPU-related design frameworks contain features that are relevant to efficient targeting from MDSDF graphs. Hou et al. present a programming language called BSGP for GPU-based implementation [51]. In BSGP, programmers write C-like sequential programs and provide special directives to configure parallel processing on the targeted GPU. The BSGP compiler translates such programs into kernels for GPU execution. GStream, a streaming-oriented design framework for GPUs, is proposed by Zhang and Mueller [52]. GStream presents a streaming abstraction dedicated to expressing data parallelism for massively parallel architectures. To offer unified programming methods for domain-specific accelerators, such as GPUs, the authors in [53] take a pragma-based approach to expressing tasks as computational kernels.

In Chapter 4 through Chapter 5, we present a novel design method, building on the MDSDF model of computation, for hierarchical exploitation of parallelism in signal processing applications. This design method, called parameterized multidimensional design hierarchy mapping (PMDHM), exposes parallelism from multidimensional dataflow graphs at different design levels in a platform-independent way, and facilitates the exploitation of parallelism using suitable platform-specific mapping optimizations at the back-end of the enclosing design flow. Graph clustering and MDSDF dataflow analysis are developed and applied in novel ways to provide a systematic framework for mapping applications to processing platforms that employ parallelism at multiple levels.

5 PMDHM Case Studies

5.1 Case Study: LFSD Subsystem for MIMO Detection

In this section, we develop a case study that demonstrates our proposed PMDHM design methodology through the GPU-based implementation of the list fixed-complexity sphere decoder (LFSD) application, which we introduced in Section 3.5. The LFSD is a computationally-intensive subsystem. The independence of operations in the LFSD among subcarriers helps to make it suitable for parallel implementation. Through this concrete and practical application example, we demonstrate how PMDHM can be applied to efficiently and systematically explore implementation trade-offs across alternative design configurations. For an overview of the functionality and parameters associated with the LFSD, we refer the reader to Section 3.5.

5.1.1 Application Graph

In the parallel design of a $\lambda = (n_1, n_2, n_3, n_4)$ LFSD subsystem, we employ two-level hierarchical parallelism for GPU implementation. At the top level, LFSD operations across different subcarriers are processed in parallel, and at the the bottom level, parallelism is exploited within the LFSD operations for individual subcarriers. At the bottom level, multiple parallel threads compute a list of candidates for each subcarrier.

In the serial design discussed in Section 3.5, the actor E_i, which represents the fixed-complexity sphere decoder (FSD) computation for layer i, consumes one token from its input port and produces n_i tokens on its output port on each invocation. The computations for the different n_i values are not independent, and thus, for parallel implementation, designers need to carefully consider issues such memory sharing and thread synchronization to ensure correctness of the results.

To avoid design complications and run-time overhead associated with such memory sharing and thread synchronization, a set $S_i = \{F_i(j) \mid 1 \leq j \leq n_i\}$ of n_i actors for each layer i is used to replace each E_i in our parallel design. At layer i, each $F_i(k)$ consumes one token and generates n_{i-1} (with $n_0 = 1$) copies of a token that encapsulates the kth optimal result for the process in the next layer. For an S_i that contains multiple actors, the actors can be executed in parallel since their underlying computations are independent. Although this new (transformed) design, where the S_is are used in place of the E_is, introduces some redundant computations, the transformed design can provide GPUs with more parallel threads, and can provide this parallelism without data sharing or synchronization needed between the introduced parallel threads.

Figure 4.4 illustrates an example of a dataflow graph for a $(1, 2, 2, P)$ LFSD subsystem for P-QAM, 4×4 MIMO detection using our parallel design, as described above. At Layer 4 (the leftmost part of Figure 4.4), full search is applied — i.e., there are P actors, $F_4(1), F_4(2), \ldots, F_4(P)$, where each $F_4(j)$ computes the jth candidate and produces two tokens (encapsulating the same jth candidate). These produced tokens are sent for further processing by Layer 3.

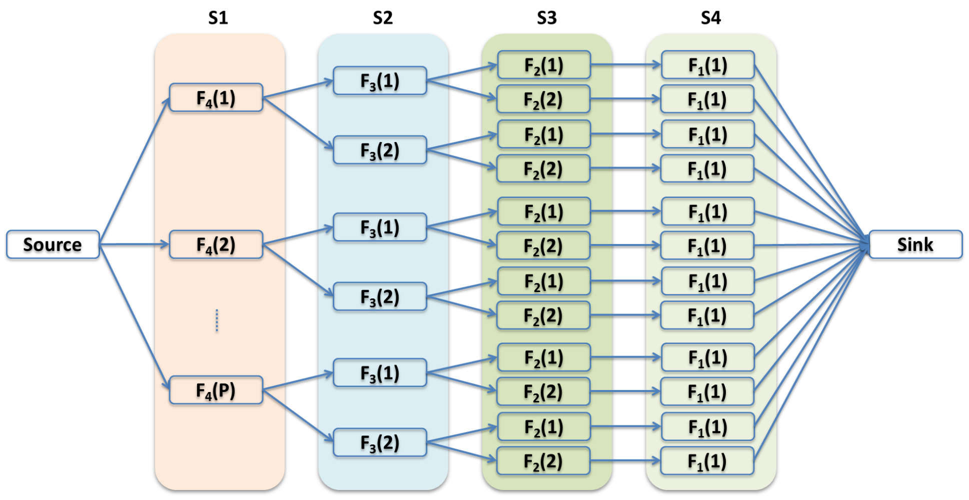
The actors $F_3(1)$ and $F_3(2)$ then compute, respectively, the optimal and second-best points according to the input tokens from the previous layer. Similarly, at Layer 2, the actors $F_2(1)$ and $F_2(2)$ consume the two results computed in Layer 3, and produce the optimal and second-best outputs of the FSD computation based on the input tokens received from Layer 3. Finally, the actor $F_1(1)$ at Layer 1 processes the input tokens received from Layer 2, and produces a single candidate around the ML solution (see Section 3.5.1) as the overall subsystem output.

5.1.2 DH Exploration

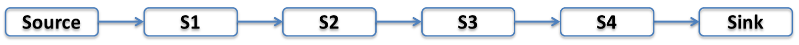
We apply our proposed PMDHM framework to the dataflow graph of the LFSD subsystem described in Section 5.1.1 to derive optimized mapping parameters for the targeted GPU platform. First, we perform actor clustering to generate mapping clusters. Here, two clustering approaches are considered:

- *Clustering Approach A* combines all processing elements into a single supernode, which is mapped to only one GPU kernel.
- *Clustering Approach B* produces four supernodes, where the ith supernode contains all processing elements at Layer i. A GPU kernel is mapped from each of these four supernodes, resulting in four generated kernels.

Compared to Clustering Approach A, Clustering Approach B may introduce additional kernel overhead (e.g., repeated computation of thread data indices) but the mapping parameters of the GPU kernels can be more specialized to their associated layers (e.g., more fine-grained control over mapping parameters). Figure 4.5



(a) Actor clustering.



(b) The PMDHM clustered graph.

Figure 5.1: Clustering Approach B.

and Figure 5.1 depict, respectively, Clustering Approaches A and B and their corresponding PMDHM clustered graphs for the application graph in Figure 4.4.

The next step in PMDHM is to construct the pipelined AMCG for each mapping cluster to explore second-level parallelism. We take the mapping cluster using Clustering Approach A as an example. Figure 4.7(a) shows the channel partitioning results for this mapping cluster. There are P channels $(X_1, X_2, \ldots, X_P)$ that can be processed in parallel. The pipeline partitioning process is further carried out for each channel as illustrated in Figure 4.7(b), where six pipeline stages (including two stages associated with `ii` and `io`) are drawn to indicate their serial (chain-structured) dependencies. In the four central (non-interface) pipeline stages $(\rho_2, \rho_3, \rho_4, \rho_5)$, there are groups of one, two, four, and four parallel threads that execute the stages, respectively. Accordingly, four threads (the maximum of 1, 2, 4, and 4) will be assigned to each channel, which results in 3 and 2 idle threads, respectively, in ρ_2 and ρ_3. The pipelined AMCG of the mapping cluster under Clustering Approach A is shown in Figure 4.8.

The PMDHM supernode transformation presented in Section 4.2 is then applied to the resulting pipelined AMCG to generate optimized mapping dimensions for each GPU kernel.

5.1.3 Experiments

In our experiments, an NVIDIA GTX680 GPU with 2GB memory and an Intel Core I7 3.4GHz CPU with 8GB memory are used.

We first compare the performance between implementations with and without application of our proposed PMDHM framework. We assume that there are N_{sc} subcarriers available at a time for parallel processing in each kernel.

The LFSD algorithm is applied on each subcarrier independently. In a (n_1, n_2, n_3, n_4) LFSD subsystem, for each subcarrier, there are $N_L = n_1 n_2 n_3 n_4$ possible parallel threads that can be executed (e.g., see Figure 4.4).

We first examine Clustering Approach A without use of PMDHM. For this case, based on the structure of the application, the kernel dimension is set to be (N_{sc}, N_L) — i.e., each grid contains N_{sc} blocks, and each block contains N_L threads to process LFSD on one subcarrier.

For Clustering Approach B without use of PMDHM, the ith layer can have

N_i parallel threads for its associated kernel, where N_i is the number of actors that can be processed in parallel at layer i — i.e., $N_4 = n_4$, $N_3 = n_4 n_3$, $N_2 = n_4 n_3 n_2$, and $N_1 = n_4 n_3 n_2 n_1$ (see Figure 5.1(a)). Therefore, the kernel dimension of each ith layer is configured as (N_{sc}, N_i).

These mapping parameter settings for Clustering Approaches A and B (without PMDHM) are determined by hand, through examination of relevant application properties.

By contrast, through our proposed PMDHM framework, the mapping parameters can be systematically derived to provide optimized performance, and allow application developers to focus more design effort on the kernel implementations and other important aspects of the design process.

Key experiment settings include the following. We conduct the experiments on $M \times M$ P-QAM modulation with $M = 4$ and $P = 16, 64$. As in Section 3.5, list sizes of P, $2P$, and $4P$ are considered in the subsystems associated with $(1, 1, 1, P)$, $(1, 1, 2, P)$, and $(1, 2, 2, P)$, respectively. The numbers of subcarriers available for one kernel launch ranges from 1 to 2048 to explore the impact of mapping parameters on kernel performance.

Figure 5.2 shows a performance comparison of the 16-QAM scheme using Clustering Approach A with and without application of the PMDHM framework. As we can see from Figure 5.2(a), the average runtime per subcarrier (y-axis) improves as the number of subcarriers per kernel (x-axis) grows. Such a trend is as expected since the acceleration provided by a GPU benefits from large amounts of data to be processed simultaneously. When $N_{sc} \leq 128$, there is little performance difference between the baseline (marked as “reg”, which is short for “regular”) and optimized (marked as “opt”) parameter settings since performance limitations arise due to the limited amount of parallel data available. Here, by baseline and optimized settings, we mean the settings derived respectively without and with application of the proposed PMDHM framework.

After the point of $N_{sc} = 128$, the implementation with the baseline mapping parameters starts to saturate very quickly, which results from inappropriate kernel dimensions: each kernel block contains 16, 32, and 64 threads, respectively, for $N_L = P$, $N_L = 2P$, and $N_L = 4P$, which are not optimal settings for this scenario, as we can see from the performance derived using the optimized parameter settings. In this region, the implementation with the optimized mapping parameters clearly outperforms the corresponding regular (baseline) one.

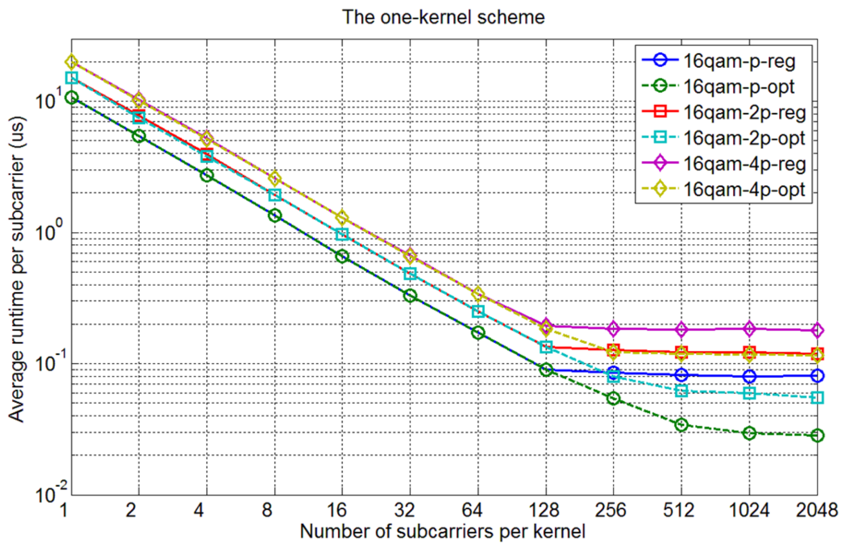
After the point of $N_{sc} = 128$, the performance of the optimized settings continue to improve and finally saturates at 256, 512, and 1024 for $N_L = P$, $N_L = 2P$, and $N_L = 4P$, respectively. From Figure 5.2(a), we observe that the baseline performance has less improvement as N_{sc} grows when $N_{sc} > 128$ for all settings of list sizes, which suggests that the maximum allowable block size is potentially causing a performance bottleneck.

In contrast, the performance of the optimized solution saturates at the same number of total threads (i.e., 16,384 total threads) for different list sizes. This is because the PMDHM framework is agile at finding efficient mapping parameters for different application specifications, and hence can remove potential performance bottlenecks due to improper kernel dimensions that affect the baseline case. From this comparison, we also see that given the same number of total threads, improper settings of parallel dimensions at the two levels can cause performance degradation. In particular, we can observe such performance degradation from the baseline settings.

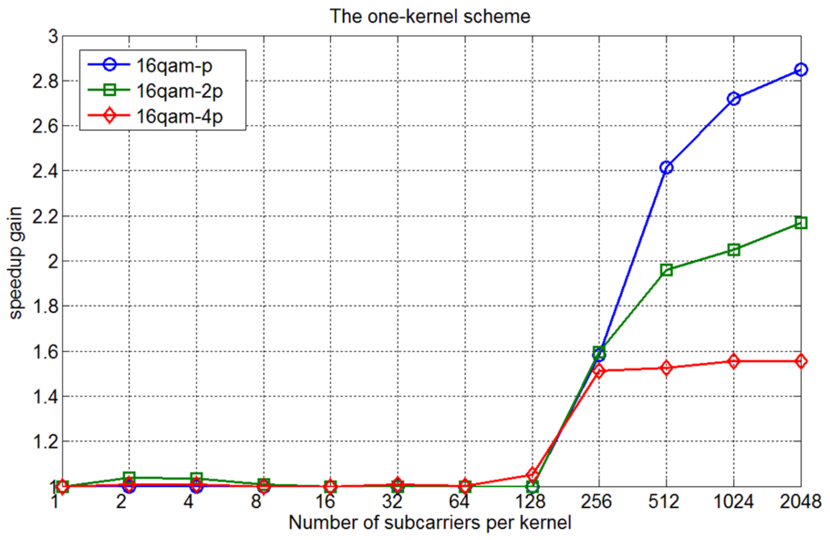
The resulting acceleration using the PMDHM framework is illustrated in Figure 5.2(b). From this figure, we see that the speedup gain achieved for this application using the PMDHM framework can be up to 2.8X, 2.2X, and 1.6X for the cases $N_L = P$, $N_L = 2P$, and $N_L = 4P$, respectively.

The situation for 64-QAM modulation is different, as shown in Figure 5.3. Here, performance enhancement by applying PMDHM can only be seen for the case $N_L = P$. The baseline settings of mapping parameters using (N_{sc}, N_L) for $N_L = 2P$ and $N_L = 4P$ are suitable for these scenarios, and the PMDHM framework reaches almost the same quality of results as using the regular settings. However, an exception can be seen for $N_L = P$, where up to a 27% performance improvement can be attained using PMDHM, as illustrated in Figure 5.3(b). From the above results, our proposed framework is shown to provide an efficient configuration of kernel dimensions regardless of application specifications, while the quality of the baseline approach is fragile — i.e., it is seen to be highly sensitive to different settings of application parameters.

As shown in Figure 5.4 for 16-QAM modulation and Figure 5.5 for 64-QAM modulation, experimental results for Clustering Approach B also illustrate significant gains achieved using the PMDHM framework. In these experiments, the PMDHM framework enhances performance by up to 2.5X, 2.2X, and 1.7X for 16-

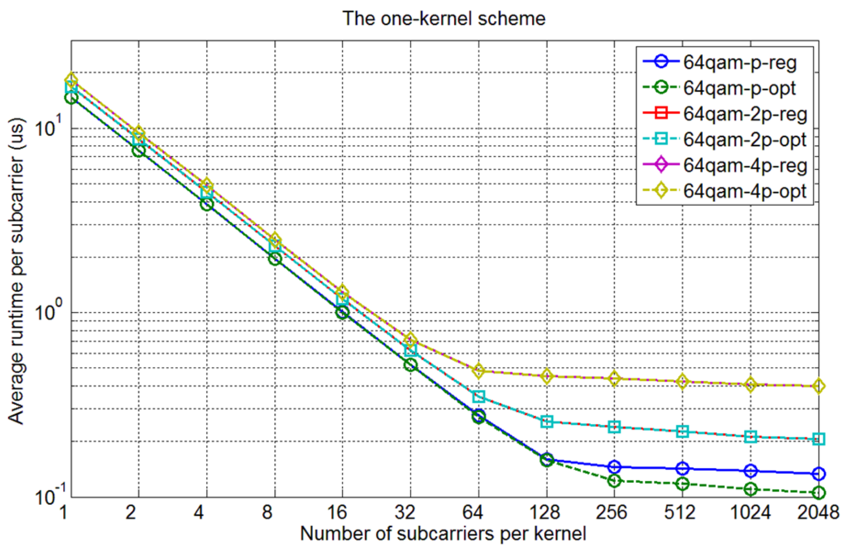


(a) The average runtime comparison.

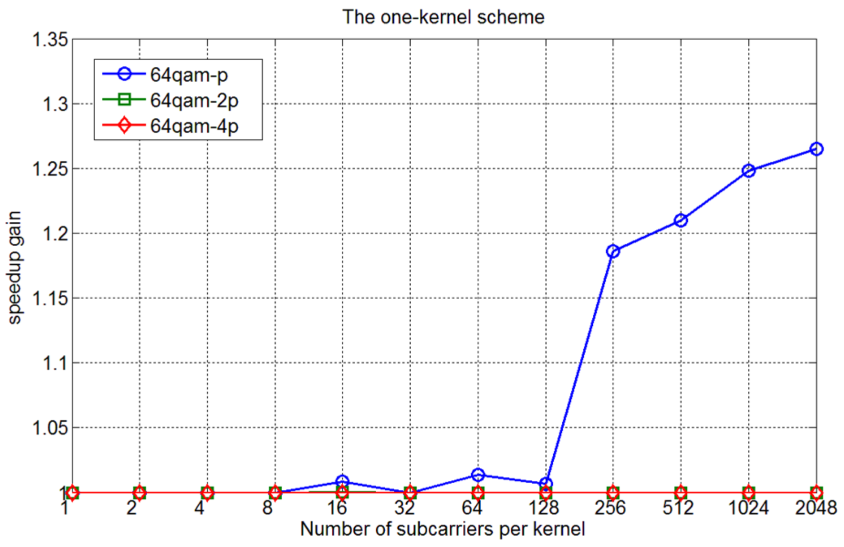


(b) The speedup gain using PMDHM.

Figure 5.2: Performance comparison of Clustering Approach A for 16-QAM with and without PMDHM.



(a) The average runtime comparison.



(b) The speedup gain using PMDHM.

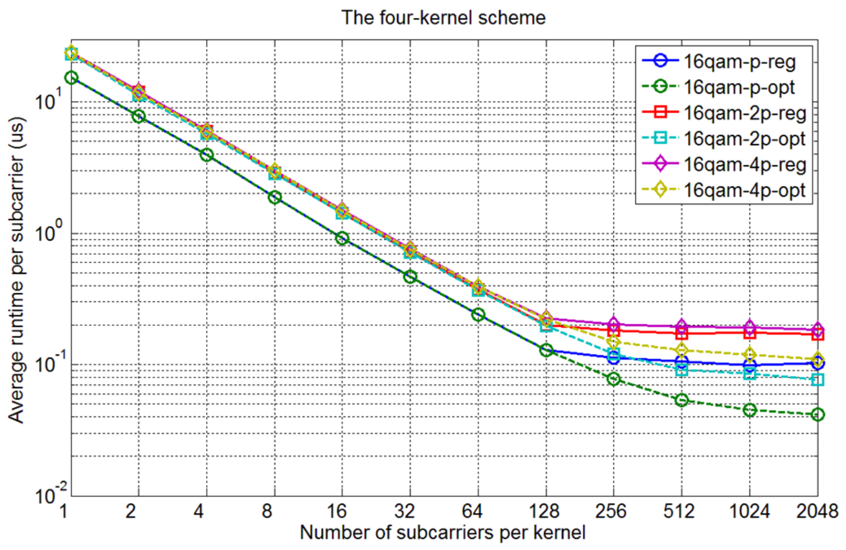
Figure 5.3: Performance comparison of Clustering Approach A for 64-QAM with and without PMDHM.

QAM with $N_L = P$, $N_L = 2P$, and $N_L = 4P$, respectively. For the case of 64-QAM with $N_L = P$, runtime improvement of up to 25% can be attained for large numbers of subcarriers per kernel.

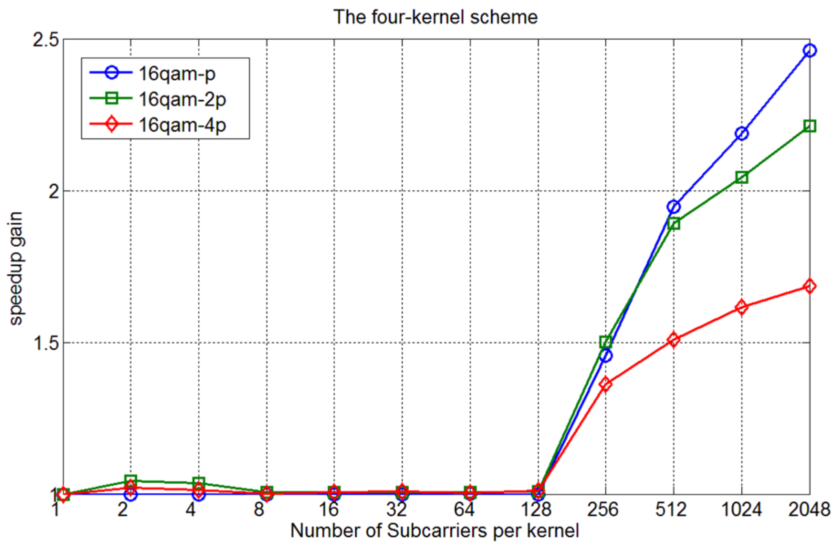
Figure 5.6 depicts a performance comparison of Clustering Approaches A and B with application of the PMDHM framework. Compared to Clustering Approach A, Clustering Approach B groups each layer into a separate kernel and thus, optimization can be carried out in a specialized way on each kernel to potentially obtain better mapping parameters and reduce the impact of idle threads. However, there is also a potential cost to Clustering Approach B in terms of introducing kernel invocation overhead.

From the experimental results, we observe that Clustering Approach A outperforms Clustering Approach B, but the difference decreases progressively as the list size increases. For small list sizes, we expect that Clustering Approach A has better performance primarily because it involves significantly less kernel invocation overhead. For larger list sizes, overhead due to inactive threads becomes significant in Clustering Approach A. For example, in the $N_L = 2P$ case, P threads must wait for the other P threads to finish in Layer 4. This effect becomes increasingly strong for larger list sizes. For $N_L = 4P$, there are $3P$ and $2P$ idle threads, respectively, in Layer 4 and Layer 3, which influence the reduced improvement of Clustering Approach A compared to Clustering Approach B. This reduction in improvement is shown in Figure 5.6(c).

Next, we present results of experiments that measure the acceleration achieved by using a GPU platform compared to a CPU. To measure CPU-based performance,

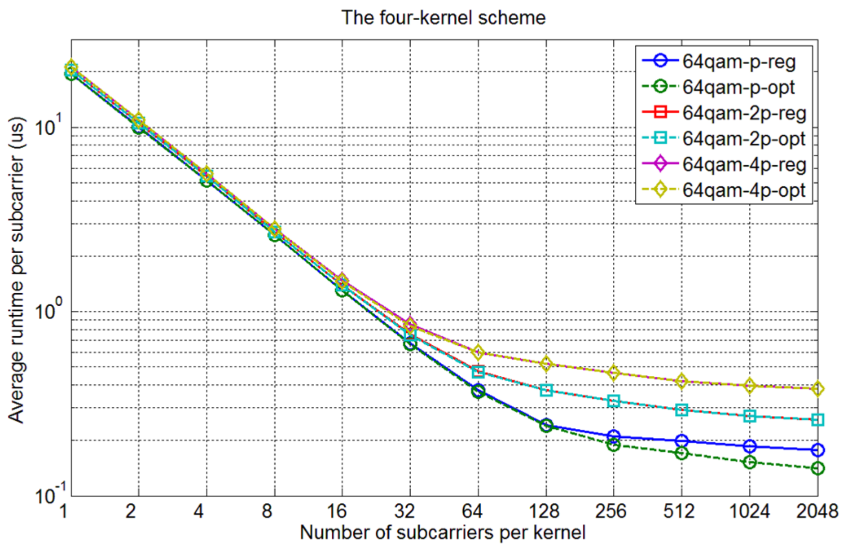


(a) The average runtime comparison.

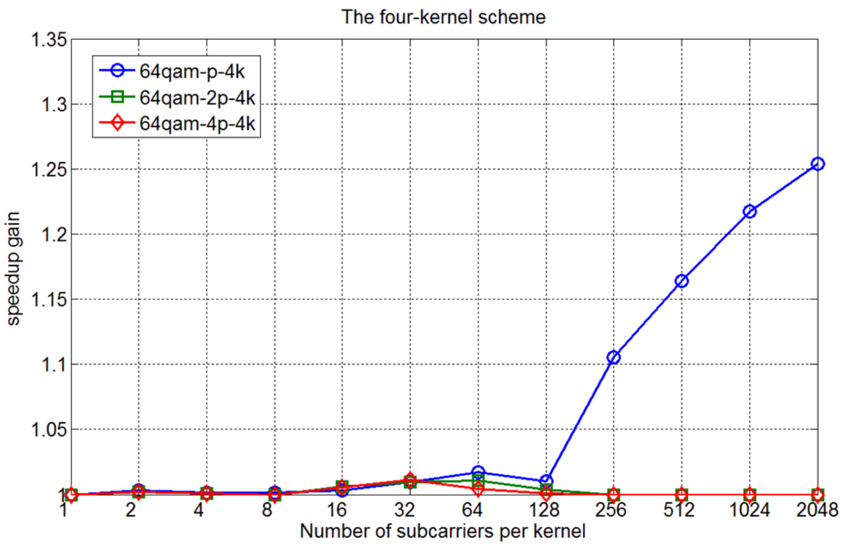


(b) The speedup gain using PMDHM.

Figure 5.4: Performance comparison of Clustering Approach B for 16-QAM with and without PMDHM.

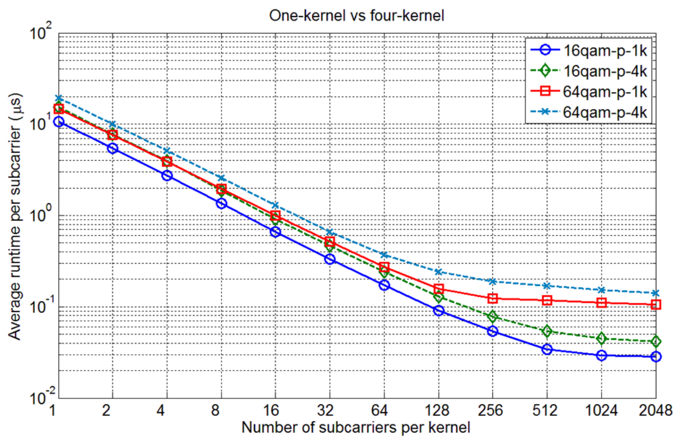


(a) The average runtime comparison.

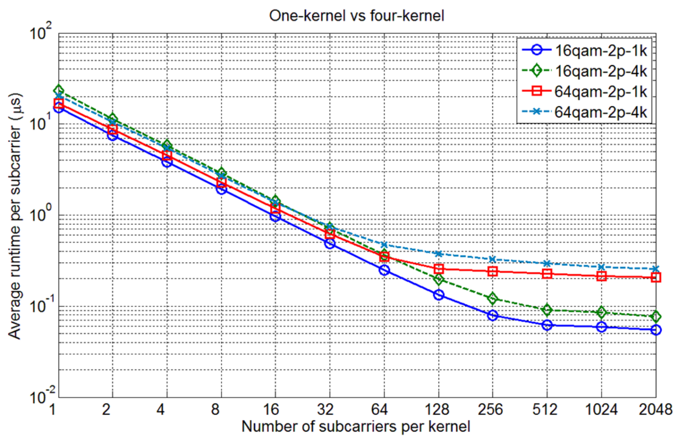


(b) The speedup gain using PMDHM.

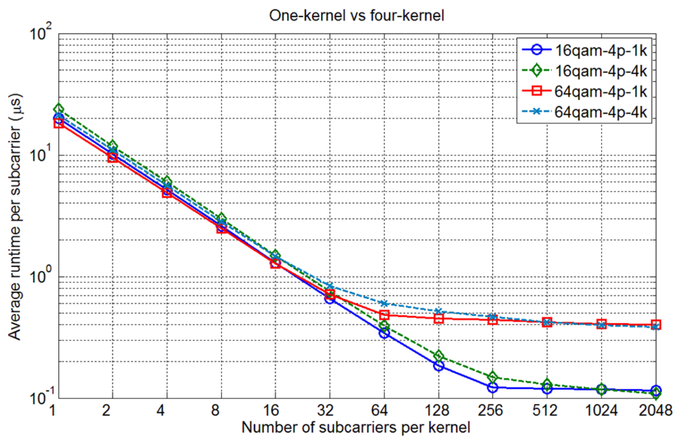
Figure 5.5: Performance comparison of Clustering Approach B for 64-QAM with and without PMDHM.



(a) $N_L = P$.



(b) $N_L = 2P$.



(c) $N_L = 4P$.

Figure 5.6: Performance comparison of Clustering Approaches A and B with the PMDHM framework.

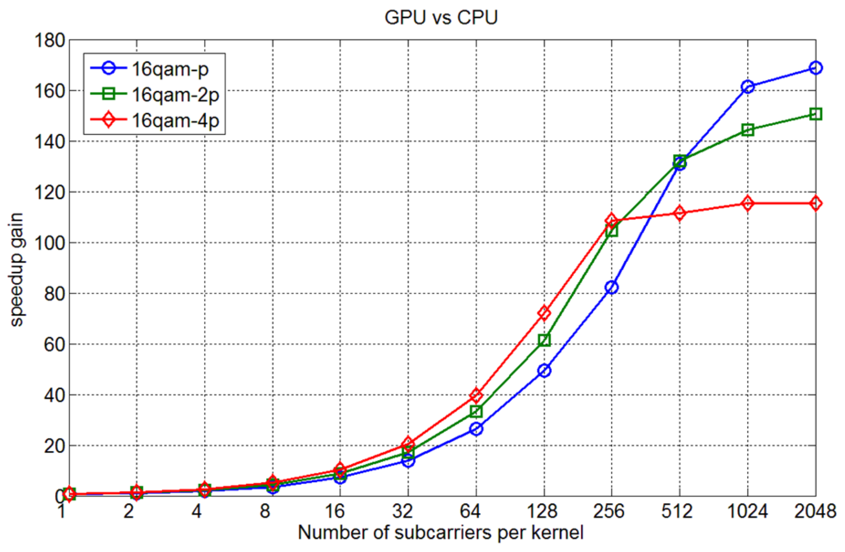
we consider both a single-core implementation using sequential code and a multi-core implementation based on the OpenMP API. For the GPU implementation, Clustering Approach A with PMDHM is used in the comparison. The acceleration levels achieved by GPU- and CPU-based implementations are depicted, respectively, in Figure 5.7 and Figure 5.8.

From the results, we see that the speedup increases as the number of subcarriers grows, and the rate of this increase slows for larger numbers of subcarriers. We can also see from the results that lower amounts of speedup are achieved for larger list sizes.

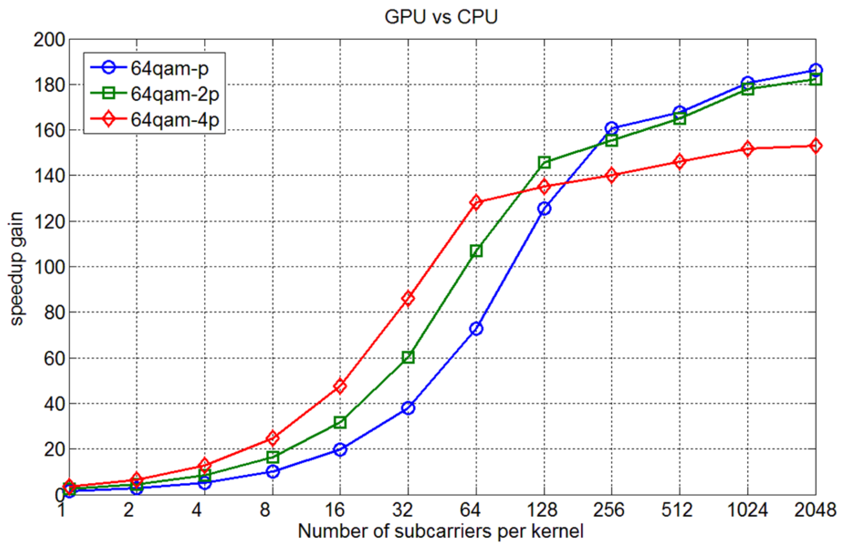
Speedup gains ranging from 120X to 180X can be obtained using GPUs compared to single-core CPUs, as we can see from Figure 5.7. Even in comparison to the multi-core CPU implementation, the GPU implementation can achieve significant speedup gains — ranging from 25X to 42X, as shown in Figure 5.8.

5.1.4 Summary

In this chapter, we have demonstrated important features and advantages of our proposed PMDHM framework through an application case study of a list fixed-complexity sphere decoder (LFSD) subsystem, which is an important subsystem in modern wireless communication applications. Through this case study, we have demonstrated concretely how the PMDHM framework can help system designers to experiment with and optimize mapping parameters through a structured process that is based on novel methods of dataflow graph modeling, analysis and transfor-

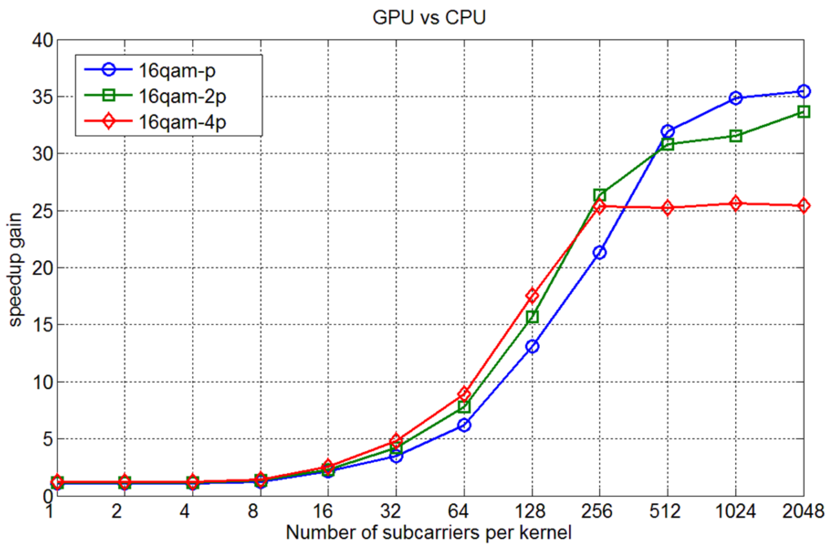


(a) The 16-QAM modulation.

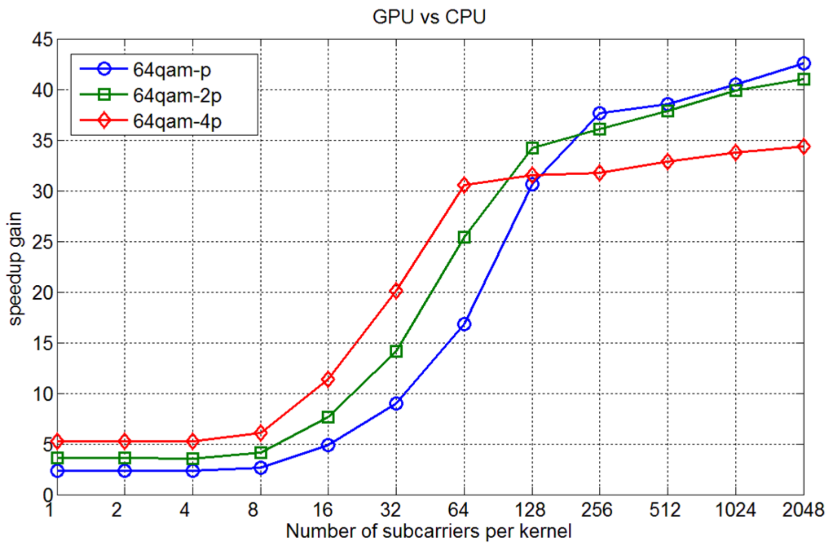


(b) The 64-QAM modulation.

Figure 5.7: Performance comparisons for GPU and single-core CPU implementations.



(a) The 16-QAM modulation.



(b) The 64-QAM modulation.

Figure 5.8: Performance comparisons for GPU and multi-core CPU implementations.

mation. An important benefit of the features demonstrated in this chapter is the potential to reduce design and optimization effort for engineering teams, which can in turn help to reduce costs, accelerate time-to-market or allow other design and validation tasks to be focused on more intensively.

5.2 Case Study: Integral Histogram

To further demonstrate our proposed PMDHM framework, we map an image processing application based on integral histogram computation [54] onto a GPU target platform. Our proposed PMDHM framework can be applied flexibly according to system properties and application specifications. Different from the case study of the LFSD, which we presented in Section 5.1, kernel dimensions for this application are configured by hand based on the derived design hierarchies to demonstrate an alternative approach to applying PMDHM. A preliminary version of material in this section is presented in [34].

The *integral histogram* (*IH*) first maps pixels into a set of non-overlapping ranges ("bins"), and then performs a 2-D scan. Two scan orders, cross-weave and wavefront, are explored in [55]. The cross-weave scan processes the image in the first dimension (horizontal scan) followed by a scan in the second dimension (vertical scan). Instead of applying two passes, the wavefront scan propagates an anti-diagonal wavefront calculation as it operates through a single scan.

In our experiments, we incorporate use of a *tiled* image processing approach, where the image is separated into blocks (tiles) of neighboring pixels. Tiled approaches can be useful for GPU implementation to enhance parallel execution across multiple threads [11]. In particular, we explore in this case study a *tiled integral histogram* (*TIH*) approach for efficient mapping into GPU implementations.

The overall input image size for IH computation is denoted as $(I_w \times I_h)$ pixels, and the number of histogram bins is denoted as N_b. In TIH computation, an image is tiled as an $(N_w \times N_h)$ rectangular arrangement of tiles, where each tile has a $(T_w \times T_h)$ rectangular arrangement of pixels. Here, $T_w = I_w/N_w$, and $T_h = I_h/N_h$. For each $(T_w \times T_h)$ tile, the IH is calculated independently. After computation of all $(N_w \times N_h)$ tile-level IHs, the results can be processed to derive the image-level IH result.

We experiment with both tiled and non-tiled versions for the cross-weave scan. We have observed that non-tiled configurations of our wavefront-based IH actor perform with unacceptable latency on the targeted GPU, and therefore, we employ only tiled configurations when using the wavefront scan.

5.2.1 Actor Design

For GPU-based implementation of IH computation, we design three types of two-dimensional signal processing actors. These actors are parameterized so that they can be statically or dynamically configured (e.g., using parameterized dataflow [7] integration with MDSDF) for the desired type of IH computation. This parameterization in conjunction with the PMDHM mapping approach helps designers to explore trade-offs involving different IH computation strategies in conjunction with efficient parallel realizations of these strategies.

Each of the three actors employed in our IH case study has a single input port and a single output port. These actors are described as follows.

First, the **Bin-Check** actor determines bin membership for pixels. The actor executes pixel checks of an image column for all bins with $CONS = (1, I_h)$ and $PROD = (1, I_h \times N_b)$. Here, and in the remainder of this section, we denote the two-dimensional (MDSDF) production and consumption rates of a given actor port as $PROD$ and $CONS$, respectively.

Second, the **Intra-Tile-IH** actor computes the IH, where the size of the input tile is specified by the actor parameters T_w (width) and T_h (height), and the scan order is specified by the *scan order* parameter of the actor. The supported settings for the scan order parameter are:

- *CWS*: Compute the IH using a cross-weave scan with tiling. The actor ports satisfy $CONS = PROD = (T_w, T_h)$
- *WFS*: Compute the IH using a wavefront scan with tiling. The ports again satisfy $CONS = PROD = (T_w, T_h)$
- *NT*: Compute the IH using a cross-weave scan without tiling — that is, calculate the IH for the input image directly with $CONS = PROD = (T_w, T_h)$.

The **Inter-Tile-IH** actor performs accumulation among tiles with a parameter, called the *accumulation order* parameter, to support different scan orders for performing the accumulation. In particular, horizontal, vertical, and wavefront scans

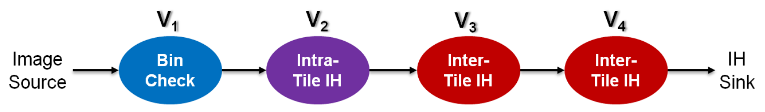


Figure 5.9: MDSDF graph for optionally-tiled IH computation.

Table 3: Application modes.

App mode	Method	V2 SOP	V3 SOP	V4 SOP
APP-CWS	cross-weave TIH	CWS	HS	VS
APP-WFS	wavefront TIH	WFS	WFS	IDLE
APP-NT	no tiling	NT	IDLE	IDLE

are used for accumulation order settings that are denoted HS, VS, and WFS, respectively. The actor ports of this actor (regardless of the accumulation order setting) satisfy $CONS = PROD = (I_w, I_h)$. In addition, the accumulation order parameter can be set to the value IDLE to bypass any accumulation. While in the IDLE configuration, the actor performs no computation, and simply passes its input to its output (through a simple pointer transfer to avoid memory transfer overhead).

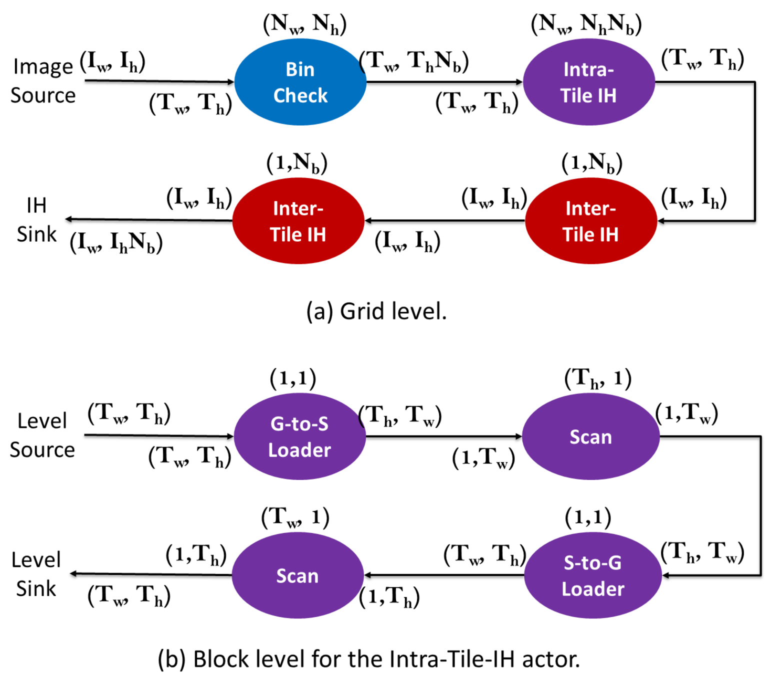
5.2.2 Application Graph

Given the actors developed in Section 5.2.1, one can implement the IH application with the MDSDF graph shown in Figure 5.9. The desired scan orders and tiling settings can be achieved by setting the actor parameter values appropriately. In the experiments, we show performance comparisons among three specific application modes, which are defined by the groups of parameter settings shown in Table 3. Here, *SOP* stands for “scan order parameter.”

5.2.3 Design Hierarchy Exploration

We customize the implementations for the different application modes by examining their MDSDF application graph representations separately, and deriving separate design hierarchies to guide the application mapping process. Taking the application mode labeled APP-CWS as an example, we show a design hierarchy in Figure 5.10 that can be used to derive an efficient implementation on the targeted GPU. In the grid level of target platform parallelism, which is illustrated in Figure 5.10(a), the 2-D indices shown above the actors represent the corresponding firing vectors that are derived from the design hierarchy (see Section 4.1). Each actor in the top level of the design hierarchy is mapped to a kernel function in the GPU, and the firing vector is used to configure the grid size.

Figure 5.10(b) depicts the second level (i.e., block level) for the Intra-Tile-IH actor. Figure 5.10(b) shows a hierarchical dataflow subgraph that specifies the internal functionality for the Intra-Tile-IH actor. To avoid non-coalesced memory access, the input data is loaded and transposed in the shared memory by the G-to-S Loader actor before the horizontal scan (G-to-S stands for "global-to-shared"). After the scan for each data row, the results are transferred from the shared memory back to the global memory by the S-to-G ("shared to global") Loader actor. Finally, a vertical scan is performed to obtain the IH for the input tile.



(a) Grid level.

(b) Block level for the Intra-Tile-IH actor.

Figure 5.10: Hierarchical dataflow graphs for cross-weave TIH.

5.2.4 Experiments

In our experiments, an NVIDIA GTX260 GPU and an Intel Xeon 3GHz CPU are used. We compare the three different application modes in Table 3. Table 4 depicts the grid and block sizes for GPU kernels. Performance is compared for four image sizes ($I_w \times I_h$): 32x32, 64x64, 256x256, and 512x512. Based on the number of GPU threads employed for each kernel, we choose a tile size of (32×16) in the APP-CWS mode for all image sizes. For the APP-WFS mode, tile sizes of (4×4), (8×8), (16×8), and (32×16) are chosen for successively larger image sizes. We evaluate the frame processing time, including the time required for memory transfer from the host to the device (GPU) and the processing time on the device. We do not include the time for memory transfer from the device back to the host because many

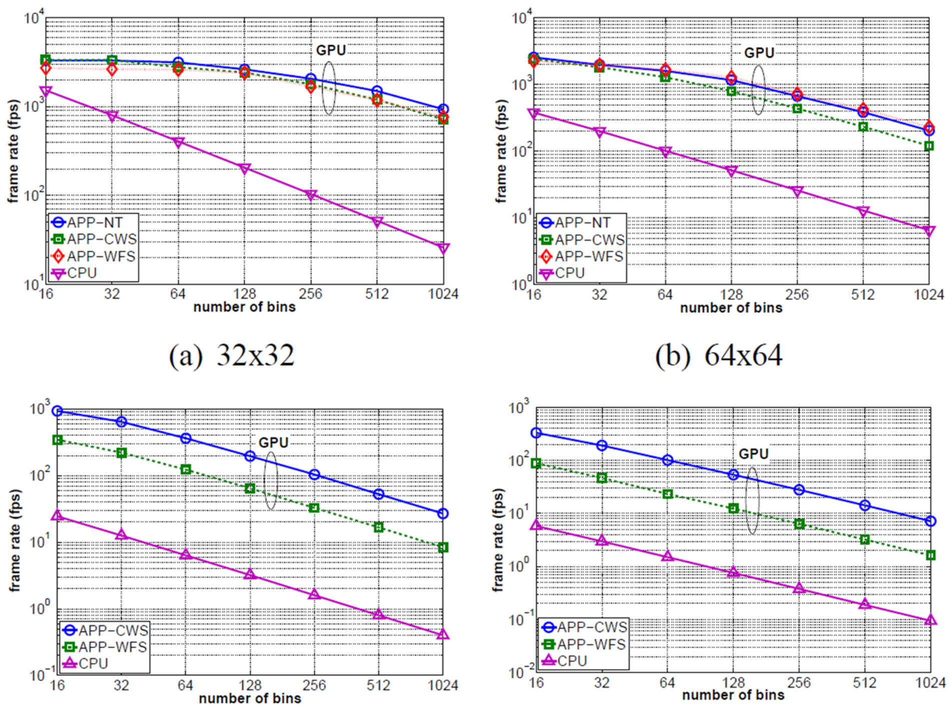
Table 4: Grid sizes (left) and block sizes (right) derived from design hierarchies in our experiments.

mode	V2 kernel	V3 kernel	V4 kernel
APP-CWS	$(N_w, N_h N_b)$ $(T_w, 1)$	$(1, N_b)$ (T_w, T_h)	$(1, N_b)$ (T_w, T_h)
APP-WFS	$(1, N_b)$ (N_w, N_h)	$(1, N_b)$ (T_w, T_h)	N/A
APP-NT	$(1, N_b)$ $(I_w, 1)$	N/A	N/A

applications that employ IH can be implemented on the GPU efficiently without need for data transfer back to the CPU.

Figure 5.11 shows the frame rates (i.e., $1/\tau$, where τ represents the average time in seconds required to process a single frame) for various bin sizes ranging from 16 to 1024. From the experimental results, we see that the GPU implementation of the IH consistently outperforms the CPU implementation, and that the speedup gains are approximately 35X for image sizes 32x32 and 64x64, 67X for image size 256x256, and 75X for image size 512x512.

Among the different GPU implementations for the 32x32 image size case, IH without tiling (APP-NT) provides the best performance since it avoids overhead from tiling. In the 64x64 case, however, APP-NT suffers from reduced inter-thread parallelism due to the large amount of shared memory required. The best performance is achieved in the APP-WFS mode as it provides more threads in the V2 kernel and less overhead due to tiling (V4 is bypassed). With image sizes of 256x256 and 512x512, we must use tiling due to the size limitations of the shared memory. Compared to APP-WFS, APP-CWS can offer better frame rates by providing more



(a) 32x32 (b) 64x64

(c) 256x256 (d) 512x512

Figure 5.11: Performance comparisons for different image sizes.

effective parallel execution on the target platform.

In summary, the best application mode for IH calculation depends on the image size, and thus MDSDF application modeling in conjunction with our PMDHM mapping approach are useful design methods to map IH computations systematically onto the targeted GPU platform. Such a systematic mapping approach leads to designs that can be mapped more efficiently, and that are more portable, and easier to maintain and extend.

6 Enhanced Scalable Schedule Trees

The *scalable schedule tree* (*SST*) model [13], built on the *generalized schedule tree* (*GST*) representation [18], is a formal method to represent and manipulate a class of parameterized dataflow graph schedules. The class of schedules targeted by SSTs is useful for implementing dataflow graph models that employ topological patterns. However, the SST method enforces certain forms of regularity in executing schedules, which restricts the flexibility with which the method can be applied to the mapping of dataflow graphs.

In this chapter, we introduce a traversal method using an *array iterator* design pattern to allow more flexible schedules so that a broad class of execution sequences can be accommodated. This allows for design and representation of a correspondingly broader range of useful schedules through the common framework of SSTs.

Our enhanced model for schedule representation is significantly more powerful than the original SST formulation, and as a target for scheduling techniques, this new model enables the development of correspondingly more flexible schedulers. We refer to this new form of schedule tree as the *enhanced SST model*.

Material in this chapter was published in preliminary form in [41].

6.1 SST Model

The SST model [13] has all of the features of a GST (e.g., see [18]) and additionally provides the following new features.

1. Parameterization. A node within an SST can be parameterized with a parameter set K. The semantics of how values associated with elements of K change is determined by the model of computation that is used for application specification (e.g., SDF with static graph parameters [56], parameterized dataflow [7], or scenario aware dataflow [57]), in conjunction with the scheduling strategy that is used to derive the schedule tree. This decoupling from parameter change semantics allows the SST model to be applied to different kinds of dataflow application models and design environments.

2. Guarded execution. An SST leaf node, which encapsulates a firing (execution) of an individual actor, has an optional *guarded* attribute, which indicates that firing of the corresponding actor should be preceded by a run-time fireability (*enabling*) check. Such an enabling check determines whether or not sufficient input data is available for the actor to fire. The guarded attribute of SSTs is motivated by the enable-invoke dataflow model of computation, where guarded executions play a fundamental role [58].

3. Dynamic iteration counts. Loop nodes can be dynamically parameterized in terms of SST parameters, which provides capabilities for data- or mode-dependent iteration in schedules. An SST loop node L can be viewed as a parameterizable form of the constant-iteration-count loop nodes in GSTs. An SST loop node L has an associated *iteration count evaluation function* $c_L : K \rightarrow \mathrm{Z}+$. An implementation of c_L takes as arguments zero or more of the parameters in K, and returns a non-negative integer (zero parameters are used if the iteration count is constant). Visitation of L

begins by calling c_L to determine the iteration count, and then executing the subtree of L successively a number of times equal to this count.

4. Arrayed children. In addition to leaf nodes and SST loop nodes, a third kind of internal node, called an *arrayed children node* (*ACN*), is introduced to represent schedule structures related to TPs.

An ACN z has an associated array children_z, which represents an ordered list of candidate children nodes during any execution of the SST subtree rooted at z. The array children_z has a positive integer size size_z, which gives the number of elements in the array.

Each element in children_z represents a schedule tree leaf node (i.e., an encapsulation of an actor in the enclosing dataflow graph), an SST loop node, or another SST — i.e., a "nested" SST.

In the enhanced SST model, ACN z also has two functions associated with it, which we denote as *trav_list_z* and *trav_count_z*. These functions determine how children_z is traversed during a given execution of the enclosing subtree. These functions take as arguments pre-specified subsets of the parameters of z, and return, respectively, an array and a non-negative integer. One or more of these functions can be constant-valued — dependence on parameter settings is not essential but rather a feature that is provided for enhanced flexibility.

6.2 SST Traversal Process

When an ACN z is visited during traversal (execution) of the enclosing schedule tree, the following sequence of steps, called the *SST traversal process*, is carried out.

(1) The parameter settings for z are updated by applying the evaluation function f_p for each parameter $p \in P_z$.

(2) The values of *trav_arr_z* and *trav_count_z* are evaluated in terms of the updated parameter settings. These values are stored in temporary variables, which we denote as `T` and `L`, respectively.

(3) To traverse the desired nodes in an ACN, we use the *array iterator* design pattern in conjunction with the traversal list array `T` and the traversal count `L` in the algorithm. The computation outlined by the pseudocode shown in Algorithm 3 is carried out, where `count` represents the iteration count evaluation function of the associated SST loop node, and `K` represents the set of parameters for the enclosing SST.

In [13], an ACN is designed to have three functions cinit_z, cstep_z, and climit_z, which allow for traversal of arrayed children nodes by stepping in regular patterns through the associated child node arrays. These regular patterns are in terms of parameterized initial indices, terminal indices, and step sizes for the arrays. In this work, by introducing the functions *trav_arr_z* and *trav_count_z* along with array iterators, we allow more flexible schedules, where a broad class of programmatic

Algorithm 3 Outline of the SST traversal process.

```
initialize a new iterator

while (the traversal is not done) {

   C = current node

   if C is a leaf node {

       execute the actor encapsulated by C

   } else if C is an SST loop node {

       Z = count(K)

       execute the loop node subtree Z times

   } else { // C is a nested SST

       recursively apply the SST traversal process to C

   }

   current node = next node;

}
```

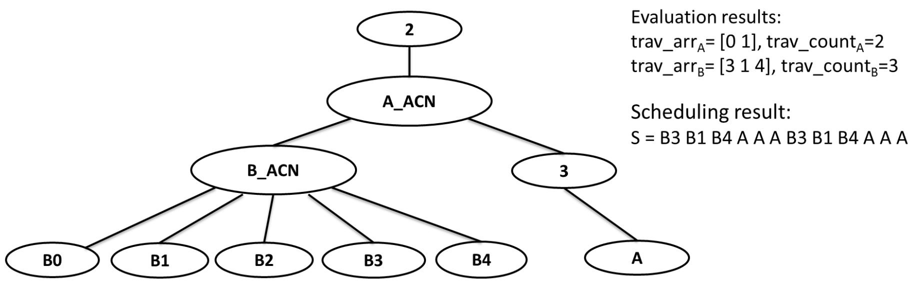

2
A_ACN
B_ACN
3
B0
B1
B2
B3
B4
A
Evaluation results:
trav_arr_A= [0 1], trav_count_A=2
trav_arr_B= [3 1 4], trav_count_B=3
Scheduling result:
S = B3 B1 B4 A A A B3 B1 B4 A A A

Figure 6.1: An example of an SST.

traversal sequences can be devised for an ACN. This allows for design and representation of a correspondingly broader range of schedules through the common framework of SSTs.

Figure 6.1 shows a synthetic example of a nested SST, where the scheduling result S shows the sequence of actor executions that results from traversing the given SST.

6.3 Case Study: Turbo Decoder

In this section, we present a case study. This case study provides a demonstration of turbo decoder implementation for wireless communication based on the SST concepts introduced in this document. The performance of implementations with different parameters is compared to demonstrate design trade-offs involved in applying the SST model, and illustrate the flexibility of the model.

6.3.1 The Dataflow Interchange Format

In this case study, we apply the *Dataflow Interchange Format* (*DIF*) framework, which provides a standard language, called The DIF Language (TDL), for applying a broad class of dataflow models of computations for signal processing applications [59]. Forms of dataflow semantics that can be expressed using TDL include graph topologies, hierarchical design structures, dataflow-related design properties, and actor-specific information. The associated software package in the DIF framework, called The DIF Package (TDP), provides intermediate representations for dataflow graphs that are specified by TDL, along with libraries of analysis techniques and transformations that operate on these representations. The analysis techniques can be used to enhance dataflow-based design flows based on TDL. Through generalized interchange capabilities provided by DIF, the analysis techniques can also be used to enhance design flows in other other dataflow environments that are interfaced to DIF (e.g., see [60, 61, 58]).

6.3.2 Turbo Codes

Turbo coding is an attractive channel coding scheme, which can provide near channel capacity performance [30]. In fact, turbo codes are used in many third and fourth generation mobile communication standards, such as CDMA2000, UMTS, WiMax, and LTE. In this section, we implement a turbo decoder and exploit features of the SST model to demonstrate useful trade-offs that can be realized with different SST-based scheduling schemes.

In this case study, we have assumed BPSK modulation (i.e., $+1$ or -1) and an AWGN channel with noise variance N_0. In the encoder, puncturing is applied for a targeted coding rate $R = 1/2$. After transmission through a wireless channel,

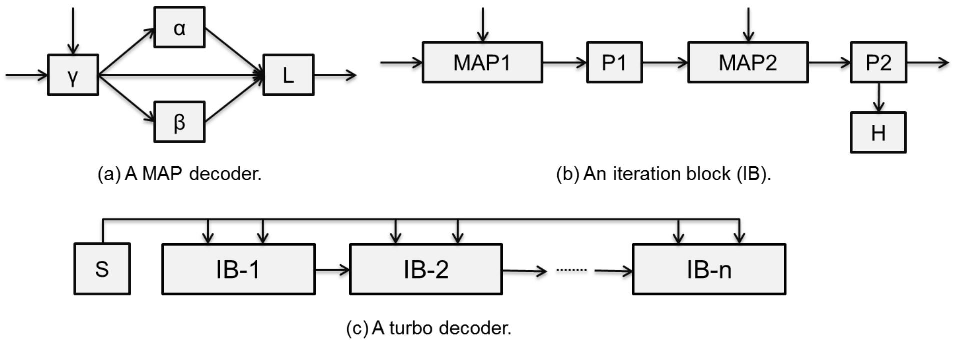


Figure 6.2: A dataflow graph representation of a turbo decoder.

the received BPSK symbols are decoded with a corresponding turbo decoder.

A *Maximum A-Posteriori* (MAP) [62] decoder is used for a component decoder with two inputs including *a-priori* information and the channel outputs. Four actors comprise a MAP decoder as illustrated in Figure 6.2(a). The γ actor computes the state transition probabilities for the input data. Then, the α and β actors evaluate, respectively, the forward and backward metrics based on the state transition probabilities. Finally, the L actor computes and exports the *Log Likelihood Ratios* (LLR) and the *extrinsic* component.

The turbo decoder operates iteratively. Each iteration block (IB) (Figure 6.2(b)) consists of two component decoders (*MAP1* and *MAP2*) that are linked by an interleaver (*P1*) and a de-interleaver (*P2*). The output of *P2* is sent to the H actor (hard decision) to generate the decoded bits provided that it is the final IB. Otherwise, the output is propagated to the next IB for another iteration. The overall graph of a turbo decoder is shown in Figure 6.2(c), where the S actor distributes the channel outputs to each IB and n is the maximal number of iterations in the implementation.

A TDL representation is specified in Figure 6.3(a) where four types of TPs, *broadcast*, *chain*, *broadmerge*, and *parallel* are employed. Among them, *broadmerge* (broadcast and then merge) is a user-defined TP as illustrated in Figure 6.3(b), which can reveal a potential structure for exploiting parallelism. In the TDL of $9n+1$ nodes, there are $2n+3$ TPs utilized to reduce the LOC cost to $2n+4$ lines of code (compared to $17n-1$ without the support of TPs). Take $n = 8$ as an instance. It needs 20 and 135 lines of code with and without TPs employed, respectively. This example shows again the significant benefit of code efficiency that can be obtained from using TPs.

6.3.3 Exploring Design Trade-offs using SSTs

Figure 6.4 shows the SST representation for the turbo decoder where we derive the SST construction from the hierarchical graphs in Figure 6.2. Underlying the root ACN (TD), there are n nested ACNs for the n IBs and one node for the source of the subsystem (derived from Figure 6.4(c)). Each nested ACN has five nodes including two nested ACNs for the two MAP decoders and three leaf nodes (see Figure 6.4(b)). Each MAP ACN contains four child nodes (see Figure 6.4(a)).

With the SST representation, developers can readily realize targeting schedules by imposing appropriate settings of parameters to explore design trade-offs. In the turbo decoder, for example, more iterations lead to more reliable decisions with the cost of more computational complexity. In wireless communication, the optimal number of iterations $d(\leq n)$ usually depends on many factors, such as channel quality, desired bit error rates (*BERs*), etc. Here, we leverage the flexibility of SST for scheduling with various numbers of iterations.

To demonstrate further, we exhibit the ACN settings using an example of $d = 3$ (i.e., three iterations for the turbo decoder). Under this scenario, only the the first three IBs will be executed and for each IB executed, the nodes *MAP1*, *P1*, *MAP2*, and *P2* are traversed. The last IB executed will visit the *H* node to generate the decoded bits while others will skip the *H* node; only the last IB has to perform the hard decision of the turbo decoder. In addition, all of the child nodes of the visited *MAP1* and *MAP2* nodes will be traversed to compute the LLR values.

To generate the corresponding schedules, the settings of ACNs can be configured as follows.

```
TD: trav_arr = [0 1 2 3], trav_count = 4.

IB-1: trav_arr = [0 1 2 3], trav_count = 4.

IB-2: trav_arr = [0 1 2 3], trav_count = 4.

IB-3: trav_arr = [0 1 2 3 4], trav_count = 5.
```

Here, the settings for the MAP ACNs are skipped, as all of their child nodes are always visited in the case study.

In the demonstration, we implement the turbo decoder and compare the performance in terms of system throughput and BER level with various iteration counts in SSTs. Our experiments on the turbo decoder case study are performed on a PC with an Intel 3GHz CPU and 4GB RAM. A parallel concatenated turbo code of memory 2 is used. The codeword size is 1024 bits and the maximal number of

Table 5: Experimental results for various numbers of iterations ($d \leq 8$). Note that the throughput values are normalized to the throughput achieved under the setting $d = 1$.

d	1	2	3	4	5	6	7	8
BER	2.6×10^{-2}	4.0×10^{-3}	7.2×10^{-4}	2.3×10^{-4}	1.3×10^{-4}	7.8×10^{-5}	6.8×10^{-5}	4.3×10^{-5}
Throughput	1.00	0.50	0.34	0.25	0.20	0.17	0.14	0.13

iterations $n = 8$. The signal-to-noise power ratio is set to 2dB.

The experimental results are listed in Table 5. As expected, the BERs decrease as d increases. The throughput levels decrease linearly, however, as the number of iterations since the IBs are the major components in this subsystem. According to the results, one can evaluate how many iterations should be applied to the subsystem to satisfy the system requirements by using less computational resources. Our experiments thus demonstrate concretely how SSTs can provide a formal path from scalable application analysis to the systematic exploration of implementation trade-offs in the design and implementation of signal processing systems.

6.4 Related Work

Scheduling is a critical aspect of implementing dataflow graphs (e.g., see [1]). Parameterized schedules have been studied before (e.g., see [7, 18]), and previously, production and consumption rates were key dataflow graph aspects that were used to generate parameterized schedules. In topological patterns, even if production and consumption rates are fixed, the schedule is still scalable in terms of the numbers of actors and edges. Such scalability, when formulated in term of topological patterns, leads to new opportunities and constraints for developing parameterized scheduling techniques.

Early work on parameterized scheduling for dataflow graphs was done in the context of parameterized dataflow representations. *Parameterized dataflow* is a meta-modeling technique that can be applied to any underlying "base" dataflow model, such as SDF [9], FRDF [63], and CSDF [15], for dynamically reconfiguring the behavior of dataflow actors, edges, subsystems, and graphs through parameter values [7]. Quasi-static scheduling techniques were developed for parameterized synchronous dataflow (PSDF), which is the integration of the parameterized dataflow meta-model with SDF as the base model [7]. However, in this work, parameterized scheduling for scalable topologies was not addressed — the underlying sets of actors and edges were assumed to be fixed.

The *reactive process networks* (RPN) model of computation supports the construction of analysis and synthesis tools for dynamic streaming multimedia applications that include both event-based and dataflow-based computations [64].

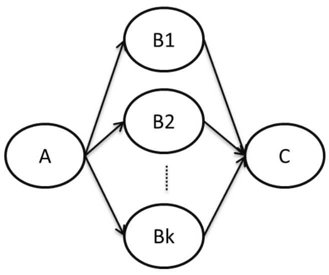
RPN provides an integration framework with run-time reconfiguration for event and stream processing that is flexible to handle run-time scheduling decisions and may also be used to represent non-deterministic stream processing behaviors. Using the *parameterized Kahn process network* (PKPN) model, designers can analyze the behavior of a parameterized system at runtime based on self-timed scheduling without introducing non-deterministic behaviors [65].

The operational semantics of the RPN and PKPN models can be viewed as extensions of the *Kahn process network* (KPN) modeling framework [66], where processes execute concurrently, applying blocking reads to assess availability of data on their inputs, and control is incorporated into processes in a distributed fashion without use of a global scheduler. While these models lead to flexible and efficient execution of KPN-related models, they, like the parameterized dataflow framework, do not address the scheduling of scalable topologies.

In this chapter, we have addressed key issues in parameterized scheduling for scalable topologies, and introduced a novel schedule model that provides for intuitive representation and efficient code generation for our targeted class of parameterized schedules.

```
topology {
  nodes = S, N[6n], A[2n], B[2n], H[n];
  edges =
    e1[2n] -> broadcast(S,N[0:3:6n]),
    e2[6n-1] -> chain(N[0:6n-1]),
    e3_1[4] -> broadmerge(N[0],A[0],B[0],N[1]),
    e3_2[4] -> broadmerge(N[3],A[1],B[1],N[4]),
                                ...
    e3_2n[4] -> broadmerge(N[6n-3],A[2n-1],
                           B[2n-1],N[6n-2]),
    e4[n] -> parallel(N[5:6:6n],H[0:n-1]);
}
```

(a) The TDL code for n IBs (see (c) for the mapping).



(b) A user-defined TP for the application. The usage is *broadmerge*(A, B1, B2, . . . , Bk,C).

Nodes	**Actors**
N[6i:6i+5]	γ_1, L_1, P1, γ_2, L_2, P2
A[2i:2i+1]	α_1, α_2
B[2i:2i+1]	β_1, β_2
H[i]	H

(c) The mapping from nodes in the topology to actors in the i-th IB. The subscripts of α, β, γ, and L indicate the associated parent MAP.

Figure 6.3: TDL code for using TPs to specify turbo a decoder.

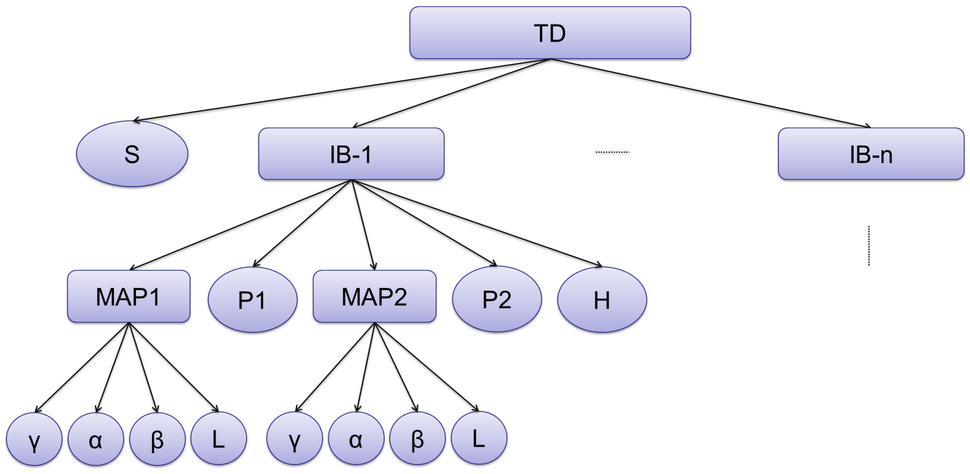


Figure 6.4: SST representation for a schedule of the targeted turbo decoder subsystem.

7 Conclusions and Future Work

In this chapter, we first summarize the contributions presented in the previous chapters of this thesis. Then, we list useful directions for future research.

7.1 Summary and Conclusions

In this thesis, we have addressed various aspects of dataflow techniques for modeling, mapping, and scheduling of dynamic signal processing applications for parallel platforms.

First, we have introduced a novel dataflow modeling approach, called core functional parameterized synchronous dataflow (CF-PSDF), that integrates core functional dataflow (CFDF) and parameterized synchronous dataflow (PSDF) techniques. CF-PSDF offers useful features including flexible dynamic parameter reconfiguration and enhanced support for quasi-static scheduling.

Using parameterized multi-mode and centralized-control methods, which we have developed in this these, the proposed CF-PSDF techniques can be applied to dynamic signal processing applications to facilitate use of efficient static and quasi-static scheduling techniques within a more general, dynamic dataflow modeling framework.

We have demonstrated the utility of CF-PSDF using a case study of soft MIMO detector implementation. Our experimental results show significant performance improvement through use of the streamlined scheduling techniques supported by CF-PSDF.

Next, we have developed a new design method, building on the MDSDF model of computation, for hierarchical exploitation of parallelism in multidimensional signal processing applications. This method, called parameterized multidimensional design hierarchy mapping (PMDHM), allows designers to explore alternative implementations in a manner that separates platform-specific parallel processing optimization from the behavioral specification, thereby enhancing portability and trade-off exploration. Our PMDHM approach includes intermediate models that provide a formal linkage between hierarchical layers of parallelism in the target platform and corresponding subsystems of the application that will be mapped onto these layers.

In the PMDHM approach, graph clustering, partitioning, and dataflow analysis and optimization are applied in novel ways to map applications to target platforms that employ parallelism at multiple levels. Applications involved in the fields of imaging processing and wireless communication are studied to demonstrate our proposed PMDHM framework. Experimental results, including detailed case studies involving list fixed-complexity sphere decoder and integral histogram implementation, show that fast GPU implementations with significant performance improvement can be derived systematically from the approach, as well as efficient trade-off analysis and optimization across different application modes.

In Chapter 6, we have presented an enhanced scalable schedule tree (SST) model for representing parameterized schedule structures based on topological patterns. This method not only offers formal modeling for deriving flexible schedules with topological patterns, but also provides a structured way to explore design trade-offs. Through a case study involving turbo decoding for communication systems, we have shown significant software coding efficiency by strategic application of topological patterns, and we have validated the scheduling flexibility provided by the proposed SST modeling techniques.

In summary, the dataflow-based techniques presented in this thesis provide systematic methods for design and implementation of signal processing applications that involve significant parameterization and dynamics in the underlying flowgraph structures. The methods that we have developed provide a variety of important features, including support for scalable and efficient design representations, scheduling techniques, and trade-off exploration on state-of-the-art processing platforms for signal processing systems.

7.2 Future Work

In this thesis, we have developed the PMDHM framework for optimized implementation of MDSDF graphs on embedded platforms that employ multiple levels of parallelism to enhance performance at different levels of granularity. At present, this framework is applied to map applications onto individual multicore platforms. Extension of the framework to handle platforms that contain multiple multicore devices is a useful direction for further investigation. Moreover, developing model-based, platform-specific optimization approaches, such as optimization for energy-efficient design strategies is an important direction of future research. We expect that an important direction in the development of such optimization processes is the construction of prediction models that can provide accurate estimation of relevant metrics (e.g., energy consumption, execution time, and latency) for the targeted platforms.

In this thesis, we have presented a formal design method for specifying topological patterns and deriving parameterized schedules from such patterns based on SST analysis. To efficiently handle dynamic dataflow (production and consumption rate) patterns in application behavior, it would be useful to extend the SST model to support important forms of scheduling dynamics.

For example, in the design of turbo decoders, engineers often use a fixed number of turbo iterations (e.g., eight iterations) to achieve good BER performance. However, early termination of iterations is possible under certain criteria to reduce the average computational complexity with minimal performance degradation [67]. Such optimization can enhance energy efficiency as well as real-time performance.

To integrate such dynamics into a model-based design methodology, one approach that can be investigated is combining the SST with the *dataflow schedule graph* (*DSG*) [68]. The DSG is a model for representing dataflow graph schedules that can support dynamic schedules, while the SST model provides a framework for derivation of efficient parameterized schedules from topological patterns. The integration of these two models, and their complementary features, is attractive as it can potentially provide systematic support for dynamic schedules using topological patterns, which will enhance both flexibility and scalability.

Model-based approaches can guide system designers in making important implementation decisions. We have demonstrated benefits gained from using model-based design in applications involving integral histogram computation and turbo decoding. The utility of model-based approaches can be further enhanced with code generation techniques that automatically translate high-level model-based specifications into hardware or embedded software implementations (e.g., see [1]). It is a useful direction of future investigation to build on our advances in the hierarchical mapping and scalable scheduling of dataflow graphs to develop code generation techniques that help to integrate and optimize these advances within automated design flows. This will involve refining our models to allow for efficient translation into embedded software code, and exploration of novel optimization techniques that help to streamline the generated code based on the associated modeling context.

Bibliography

[1] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, 2010.

[2] Y-K Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard, "Signal processing on platforms with multiple cores: Part 1 — overview and methodologies," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 24–25, November 2009, Guest Editors' Introduction.

[3] Y-K Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard, "Signal processing on platforms with multiple cores: Part 2 — design and applications," *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 20–21, March 2010, Guest Editors' Introduction.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[5] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.

[6] W. Plishker, N. Sane, and S. S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, April 2009, pp. 111–116.

[7] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001, DOI:10.1109/78.950795.

[8] S. Ha and E. A. Lee, "Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration," *IEEE Transactions on Computers*, vol. 40, no. 11, pp. 1225–1238, November 1991.

[9] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[10] P. K. Murthy and E. A. Lee, “Multidimensional synchronous dataflow,” *IEEE Transactions on Signal Processing*, vol. 50, no. 8, pp. 2064–2079, August 2002.

[11] *NVIDIA CUDA C Programming Guide*, April 2012, Version 4.2.

[12] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya, “Scalable representation of dataflow graph structures using topological patterns,” in *Proceedings of the IEEE Workshop on Signal Processing Systems*, San Francisco Bay Area, USA, October 2010, pp. 13–18.

[13] S. Wu, C. Shen, N. Sane, K. Davis, and S. Bhattacharyya, “Parameterized scheduling for signal processing systems using topological patterns,” in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Kyoto, Japan, March 2012, pp. 1561–1564.

[14] J. B. Dennis, “First version of a data flow procedure language,” Tech. Rep., Laboratory for Computer Science, Massachusetts Institute of Technology, May 1975.

[15] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-static dataflow,” *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[16] J. T. Buck and E. A. Lee, “Scheduling dynamic dataflow graphs using the token flow model,” in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.

[17] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, “Functional DIF for rapid prototyping,” in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[18] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere, “Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation,” *IEEE Transactions on Signal Processing*, vol. 55, no. 6, pp. 3126–3138, June 2007.

[19] W. Li, Z. Fan, X. Wei, and A. Kaufman, “Flow simulation with complex boundaries,” in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr, R. Fernando, and T. Sweeney, Eds. Addison-Wesley Professional, 2005.

[20] A. Griesser, S. De Roeck, A. Neubeck, and L. V. Gool and, “GPU-based foreground-background segmentation using an extended colinearity criterion,” in *Proceedings of the Workshop on Vision, Modeling and Visualization*, 2005.

[21] J. Franco, G. Bernabe, J. Fernandez, and M. E. Acacio, “A parallel implementation of the 2D wavelet transform using CUDA,” in *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 111–118.

[22] M. Wu, Y. Sun, G. Wang, and J. R. Cavallaro, "Implementation of a high throughput 3GPP turbo decoder on GPU," *Journal of Signal Processing Systems*, vol. 65, no. 2, 2011.

[23] *CUDA C Best Practice Guide*, 2013, `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`, visited on March 8, 2013.

[24] *ATI Stream Computing OpenCL Programming Guide*, June 2010.

[25] OpenACC, *The OpenACC Application Programming Interface, Version 1.0*, 2011.

[26] S. S. Bhattacharyya, W. Plishker, N. Sane, C. Shen, and H. Wu, "Modeling and optimization of dynamic signal processing in resource-aware sensor networks," in *Proceedings of the Workshop on Resources Aware Sensor and Surveillance Networks in conjunction with IEEE International Conference on Advanced Video and Signal-Based Surveillance*, Klagenfurt, Austria, August 2011, pp. 449–454.

[27] L. Wang, C. Shen, and S. S. Bhattacharyya, "Parameterized core functional dataflow graphs and their application to design and implementation of wireless communication systems," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Taipei, Taiwan, October 2013, pp. 1–6.

[28] E. Telatar, "Capacity of multi-antenna Gaussian channels," *European Transactions on Telecommunications*, vol. 10, no. 6, pp. 585–595, 1999.

[29] L. G. Barbero and J. S. Thompson, "Extending a fixed-complexity sphere decoder to obtain likelihood information for Turbo-MIMO systems," *IEEE Transactions on Vehicular Technology*, vol. 57, no. 5, pp. 2804–2814, 2008.

[30] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *IEEE International Conference on Communications*, 1993, pp. 1064–1070.

[31] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Computation of buffer capacities for throughput constrained and data dependent inter-task communication," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, March 2008.

[32] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity — the Ptolemy approach," *Proceedings of the IEEE*, January 2003.

[33] B. Kienhuis and E. F. Deprettere, "Modeling stream-based applications using the SBF model of computation," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, September 2001, pp. 385–394.

[34] L. Wang, C. Shen, G. Seetharaman, K. Palaniappan, and S. S. Bhattacharyya, "Multidimensional dataflow graph modeling and mapping for efficient GPU implementation," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Québec City, Canada, October 2012, pp. 300–305.

[35] L.-H. Wang, S. S. Bhattacharyya, A. Vosoughi, J. Cavallaro, M. Juntti, J. Boutellier, O. Silven, and M. Valkama, "Dataflow modeling and design for cognitive radio networks," in *Proceedings of the International Conference on Cognitive Radio Oriented Wireless Networks*, Washington, DC, July 2013, pp. 196–201.

[36] K. Thulasiraman and M. N. S. Swamy, *Graphs: Theory and Algorithms*, Wiley-Interscience.

[37] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[38] C. R. Johns and D. A. Brokenshire, "Introduction to the Cell Broadband Engine architecture," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 503–519, 2007.

[39] H. Oh and S. Ha, "Fractional rate dataflow model and efficient code synthesis for multimedia applications," *ACM SIGPLAN Notices*, vol. 37, July 2002.

[40] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya, "Topological patterns for scalable representation and analysis of dataflow graphs," *Journal of Signal Processing Systems*, vol. 65, no. 2, pp. 229–244, 2011.

[41] L. Wang, C.-C. Shen, S. Wu, and S. S. Bhattacharyya, "Parameterized scheduling of topological patterns in signal processing dataflow graphs," *Journal of Signal Processing Systems*, vol. 71, no. 3, pp. 275–286, June 2013, DOI:10.1007/s11265-012-0719-x.

[42] S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman, "Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems," *IEEE Transactions on Signal Processing*, vol. 43, no. 6, pp. 1468–1484, June 1995.

[43] S. Hong and H. Kim, "An integrated GPU power and performance model," in *International Symposium on Computer Architecture*, 2010, pp. 280–289.

[44] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2009, pp. 1–12.

[45] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, October 2008.

[46] S. Kwon, H. Jung, and S. Ha, "H.264 decoder algorithm specification and simulation in simulink and PeaCE," in *Proceedings of the International SoC Design Conference*, October 2004, pp. 9–12.

[47] J. Keinert, C. Haubelt, and J. Teich, "Modeling and analysis of windowed synchronous algorithms," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, May 2006.

[48] P. Boulet, "Array-OL revisited, multidimensional intensive signal processing specification," Tech. Rep., INRIA, February 2007.

[49] D. Ko and S. S. Bhattacharyya, "Modeling of block-based DSP systems," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 40, no. 3, pp. 289–299, July 2005.

[50] J. McAllister, R. Woods, R. Walke, and D. Reilly, "Synthesis and high level optimisation of multidimensional dataflow actor networks on FPGA," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2004.

[51] Q. Hou, K. Zhou, and B. Guo, "BSGP: bulk-synchronous GPU programming," in *Proceedings of ACM SIGGRAPH*, 2008.

[52] Y. Zhang and F. Mueller, "GStream: A general-purpose data streaming framework on GPU clusters," in *Proceedings of the International Conference on Parallel Processing*, 2011, pp. 245–254.

[53] E. Ayguade, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Orti, "An extension of the starss programming model for platforms with multiple GPUs," in *Euro-Par 2009 Parallel Processing*, vol. 5704 of *Lecture Notes in Computer Science*, pp. 851–862. Springer-Verlag, aug 2009.

[54] F. Porikli, "Integral histogram: a fast way to extract histograms in cartesian spaces," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2005, pp. 829–836.

[55] P. Bellens, K. Palaniappan, R. M. Badia, G. Seetharaman, and J. Labarta, "Parallel implementation of the integral histogram," in *Proceedings of the International Conference on Advanced Concepts for Intelligent Vision Systems*, 2011.

[56] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. S. Bhattacharyya, "Gabriel: A design environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 11, pp. 1751–1762, November 1989.

[57] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *Proceedings of the International Conference on Formal Methods and Models for Codesign*, July 2006.

[58] C. Shen, H. Wu, N. Sane, W. Plishker, and S. S. Bhattacharyya, "A design tool for efficient mapping of multimedia applications onto heterogeneous platforms," in *Proceedings of the IEEE International Conference on Multimedia and Expo*, Barcelona, Spain, July 2011, 6 pages in online proceedings.

[59] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[60] H. Kee, S. S. Bhattacharyya, and J. Kornerup, "Efficient static buffering to guarantee throughput-optimal FPGA implementation of synchronous dataflow graphs," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, July 2010, pp. 136–143.

[61] R. Gu, J. W. Janneck, S. S. Bhattacharyya, M. Raulet, M. Wipliez, and W. Plishker, "Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 11, pp. 1646–1657, November 2009.

[62] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Transactions on Information Theory*, vol. 20, no. 2, pp. 284–287, 1974.

[63] H. Oh and S. Ha, "Fractional rate dataflow model for efficient code synthesis," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 37, pp. 41–51, May 2004.

[64] M. Geilen and T. Basten, "Reactive process networks," in *Proceedings of the International Workshop on Embedded Software*, September 2004, pp. 137–146.

[65] H. Nikolov, T. Stefanov, and E. Deprettere, "Modeling and FPGA implementation of applications using parameterized process networks with non-static parameters," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 2005.

[66] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, 1974.

[67] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Transactions on Information Theory*, vol. 42, no. 2, pp. 429–445, 1996.

[68] H. Wu, C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya, "A model-based schedule representation for heterogeneous mapping of dataflow graphs," in *Proceedings of the International Heterogeneity in Computing Workshop*, Anchorage, Alaska, May 2011, pp. 66–77.