ABSTRACT

Title of Dissertation: MODELING AND SOFTWARE SYNTHESIS FOR MULTIPROCESSOR IMPLEMENTATION OF WIRELESS COMMUNICATION SYSTEMS

Shuoxin Lin, Doctor of Philosophy, 2016

Directed by: Professor Shuvra Bhattacharyya
Department of Electrical and Computer Engineering

In recent years, the complexity of designing embedded signal processing systems for wireless communications has increased significantly based on the need to support increasing levels of operational flexibility and adaptivity, while also supporting increasing data rates and bandwidths. These trends pose important design and implementation challenges to meet the required demands on communication system performance, real-time operation, energy efficiency, and reconfigurability.

Dataflow models of computation provide a useful framework that can be built upon to address these challenges. Dataflow models provide high-level abstractions for specifying, analyzing and implementing a wide range of embedded signal processing applications. They allow designers to specify an application using high-level, platform-independent representations, and synthesize optimized embedded software that is targeted to specific types of hardware resources and design constraints.

The growing complexity of wireless communication systems, as motivated above, along with the complexity of system-on-chip platforms for embedded signal

processing result in new problems that must be addressed in developing effective dataflow-based design methodologies. First, significant improvements to dataflow-based models and methods are needed to effectively utilize heterogeneous computing platforms and multiple forms of parallelism under stringent constraints on real-time performance and energy consumption. Second, effective modeling and analysis methods for handling dynamic parameters within dataflow graph components are needed for reliable and efficient management of system-level adaptivity and reconfiguration.

In this thesis, we address these problems by developing an integrated framework that exploits pipeline, data and task-level parallelism in dataflow models under memory constraints, and proposing novel dataflow modeling concepts and performance optimization techniques for design and implementation of dynamically parameterized communication systems. The main contributions of the thesis are summarized as follows:

(1) Software synthesis framework for heterogeneous signal processing platforms. We have developed an integrated dataflow-based design framework called *DIF-GPU*, which provides a toolset for specification, optimization and software synthesis of embedded software targeted to heterogeneous CPU-GPU platforms. DIF-GPU incorporates novel models and methods in the dataflow interchange format (DIF) that are geared toward design optimization of signal processing systems on heterogeneous architectures composed of multicore CPUs and GPUs. DIF-GPU helps to free developers from low-level, platform-specific fine-tuning, and allows them to focus on higher-level aspects of communication system design.

(2) Vectorization in DIF-GPU. In the context of dataflow models for embedded signal processing, vectorization is an important transformation for exploiting data parallelism. We have developed new techniques for integrated dataflow graph vectorization and scheduling on heterogeneous platforms. These techniques are developed in the DIF-GPU framework to provide optimized vectorization and scheduling capabilities for hybrid CPU-GPU platforms under memory constraints. For the targeted class of platforms, these techniques are shown to provide significantly better processing throughput compared to previous methods for a given memory constraint. We demonstrate our integrated vectorization and scheduling techniques by applying them to an Orthogonal Frequency Division Multiplexing (OFDM) receiver system.

(3) Modeling parameterized, dynamic dataflow behavior. We introduce a novel modeling method, called *parameterized sets of modes* (*PSMs*), that enables efficient representation and analysis of adaptive and dynamically reconfigurable signal processing functionality. PSMs can be viewed as high-level abstractions that model parameterized functionality involving groups of related regimes of operation ("modes") for dynamic dataflow models. We develop formal foundations for PSM-based modeling, and demonstrate the utility of this form of modeling by using it to develop efficient methods for scheduling dynamically parameterized dataflow graphs on different types of relevant platforms.

Modeling and Software Synthesis for Multiprocessor Implementation of Wireless Communication Systems

by

Shuoxin Lin

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
2016

Dedication

To my wife, son and parents

Acknowledgments

I sincerely thank Prof. Shuvra Bhattacharyya for his invaluable support, guidance and encouragement throughout the years of my PhD study. His continuous support from all aspects of my work and life has allowed me to complete this study, and overcome difficulties of all kinds. His generous and insightful guidance is indispensable for my successful completion of my study. I am truly grateful to have him as my advisor.

I also want to thank my committee members, Professor Levine, Professor Franklin, Professor Srivastava and Professor Tao for providing insightful and valuable feedbacks.

I am grateful to Dr. William Plishker, Dr. Chung-Ching Shen, Dr. Lai-huei Wang, Dr. George Zaki, Ms. Yanzhou Liu, and other colleagues and collaborators for their generous help. I would also like to thank Marshall Plan Foundation and Fachhochschule Salzburg for providing a great opportunity for collaborative research.

Finally I give my special thanks to my wonderful parents, my wife and my son for their love and understanding.

The research underlying this thesis was supported in part by the Laboratory for Telecommunication Sciences, U.S. National Science Foundation, and Tekes — The Finnish Funding Agency For Innovation.

Table of Contents

List of Figures

Chapter 1

Introduction

The evolution of technologies to support fifth generation (5G) wireless communication systems has resulted in rapidly increasing requirements on throughput and flexibility in design and implementation of embedded signal processing systems for wireless communications. On one hand, to support key components in 5G compatible user devices, such as massive multiple input, multiple output (MIMO) systems [1], interference alignment [2] and device-to-device (D2D) communication [3], the baseband processing speed of 5G user devices needs to be dramatically improved. Meanwhile, capabilities such as context-awareness and multi-radio-access technologies require 5G user devices to be highly flexible with cognitive capabilities for spectrum sensing and multi-layer reconfiguration (e.g., see [4, 5, 6]).

Dataflow models of computation provide a useful framework that can be built upon to address these challenges. Dataflow models provide high-level abstractions for specifying, analyzing and implementing a wide range of embedded system applications (see [7]). A dataflow graph is a directed graph $G = (V, E)$ with a set of vertices (*actors*) V and a set of edges E. An actor $v \in V$ represents a computational task of arbitrary complexity. An edge $e = (u, v) \in E$ represents a first-in, first-out (FIFO) buffer that stores data values as they are produced by u and consumed by v.

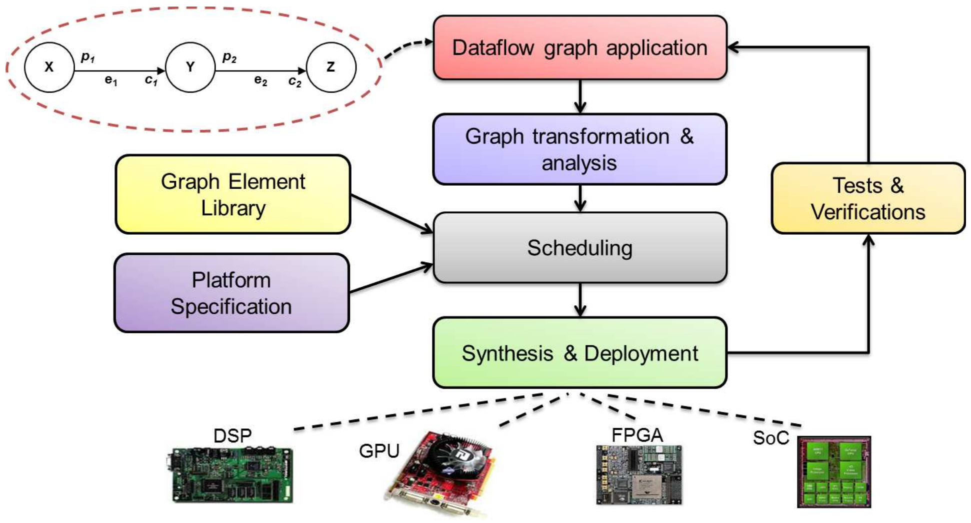


Figure 1.1: Overview of dataflow-based design framework.

Dataflow models offer a promising foundation for systematic design methodologies for those developing wireless communication systems, in part because they provide natural, scalable and retargetable representations of signal processing applications [7]. Figure 1.1 demonstrates the general workflow of dataflow-based design.

Key advantages provided by effective incorporation of dataflow-based design methodologies include:

- *Rapid Prototyping* — Given libraries of dataflow graph components (actors and FIFOs), designers can quickly generate system-level prototypes for concept validation and analysis under multiple design constraints [8].
- *Retargetability* — Since dataflow graphs are high-level abstractions of the underlying applications, designers can migrate a common dataflow model of an application across different types of computing platforms, while changes are mostly localized to the implementations of individual actors and FIFO types.
- *Explicit Parallelism* — In dataflow graphs, pipeline-, data- and task-level par-

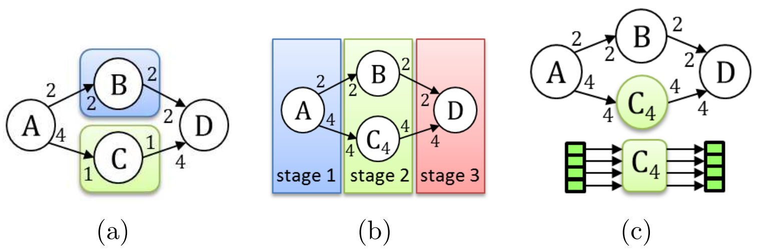


Figure 1.2: Parallelism expressed in a synchronous dataflow graph. (a) Task parallelism. (b) Pipeline parallelism. (c) Data parallelism by actor vectorization.

allelism are explicitly exposed, as illustrated in Figure 1.2. Pipeline parallelism can be achieved by overlapping the execution of actor firings that belong to different graph iterations (see Figure 1.2(a)); task parallelism can be exploited by assigning different actors to different processors or processor cores (see Figure 1.2(b)); and data parallelism can be exploited by *actor vectorization*, where different firings of the same actor execute simultaneously to process different tokens (see Figure 1.2(c)).

These advantages of applying dataflow-based design methodologies can significantly improve the designer's productivity and effectiveness in validating and optimizing complex signal processing implementations that must satisfy stringent, multi-dimensional constraints.

Software synthesis from dataflow graphs [9] is the process of generating efficient embedded software implementations from applications specified as dataflow graphs. As described above, new requirements on flexibility and throughput emerging from 5G technologies pose novel constraints on software synthesis from dataflow graphs. First, significant improvements to dataflow-based models and methods are needed to effectively utilize heterogeneous computing platforms and multiple forms of parallelism under stringent constraints on real-time performance and energy consumption. Second, effective modeling and analysis methods for handling dynamic parameters within dataflow graph components are needed for reliable and efficient management of system-level adaptivity and reconfiguration.

In this thesis, we address these problems by developing an integrated framework that exploits pipeline, data and task-level parallelism in dataflow models under memory constraints, and proposing novel dataflow modeling concepts and performance optimization techniques for design and implementation of dynamically parameterized communication systems. We outline the contributions of this thesis in the remainder of this chapter.

1.1 Contributions

In this thesis, we develop an integrated framework that synthesizes high-throughput embedded software from applications specified using dataflow models. The synthesized software is targeted to *hybrid CPU-GPU platforms* (*HCGPs*), which is a class of heterogeneous computing platforms that is of increasing relevance in embedded signal processing systems. We have also developed new modeling techniques for representation of reconfigurable applications using dynamic dataflow graphs, and efficient scheduling of these graphs. Here, by *scheduling*, we mean the assignment of actors to processing resources, and the ordering of actors that share the same resource. Scheduling is a critical aspect of software synthesis that has strong influence on relevant implementation metrics.

Although our work is driven by emerging challenges in design and implementation of wireless communication systems, it is envisioned that the tools and techniques developed in this work can be readily adapted across a wide range of other signal processing application areas in which system-level reconfiguration capabilities and heterogeneous multi-processor architectures are relevant. Investigating such adaptations is a useful direction for future work.

The contributions of this thesis are presented in three main parts. The first part presents an integrated software synthesis framework for mapping dataflow graphs onto HCGPs. The second part develops new models and methods for memory-constrained, HCGP-targeted vectorization and scheduling of dataflow graphs. The third part introduces new methods to specify and analyze parameterized dynamic dataflow functionality for reconfigurable signal processing systems.

1.1.1 Design Framework for Heterogeneous Platforms

Software development for heterogeneous embedded platforms without high-level, model-based support can be challenging and inefficient, since programming models for heterogeneous platforms, such as CUDA [10] and OpenCL [11], require the developer to manually manage low level operations including data transfer, task scheduling, and processor synchronization. In addition, the performance gain on CPU-GPU platforms is dependent on various complex factors, including the nature of the computational tasks that are executed, details of the targeted hardware platforms, and the amount of parallel data available (e.g., see [12]). Thus, case-by-case derivation and implementation at the system level can be highly error-prone and time-consuming. Without careful consideration of the hardware details, the speedup gain from GPU can be reduced or depleted (e.g., see [13, 14]).

A major challenge in developing such high-level, model-based design tools for CPU-GPU platforms is the optimized exploitation and integration of heterogeneous forms of parallelism, including pipeline, data, and task parallelism. Such optimization is critical to effective utilization of CPU-GPU platforms and is complicated due to inter-related design issues that include task scheduling, interprocessor communication, and memory management.

In this thesis, we have developed an integrated dataflow-based design framework called *DIF-GPU*, which addresses this challenge, and provides new methods for synthesis and optimization of embedded software targeted to HCGPs. DIF-GPU builds on the dataflow interchange format (DIF) package, which is a software environment for developing and experimenting with dataflow-based design methods and synthesis techniques for embedded signal processing systems [15]. Our framework frees developers from low-level, platform-specific fine-tuning, and allows designers to focus on higher level aspects of system design, such as iteration on the set of supported application features or quality-of-service trade-offs.

The main contributions of this framework are summarized as follows. (1) We have developed novel techniques for integrating dataflow vectorization into the scheduling process as an effective way to exploit data-parallelism when implementing SDF graphs on hybrid CPU-GPU platforms. (2) We have developed compile-time tools and a run-time system for scheduling vectorized dataflow actors on CPU-

GPU platforms in a manner that provides significant throughput improvement over conventional scheduling techniques. (3) We have integrated vectorization, scheduling and code generation capabilities in the DIF-GPU framework to provide a high level of automation in generating efficient embedded software from SDF representations.

The details of the DIF-GPU framework are presented in Chapter 3.

1.1.2 Memory-Constrained Vectorization and Scheduling

System-level performance optimization requires efficient utilization of both CPU cores and GPUs on HCGPs. Manual performance tuning on a case-by-case suffers from inefficiency and can lead to sub-optimal solutions. When system constraints or the target platforms are changed, designer often need to repeat the same process, which further reduces development productivity. Therefore, methods based on high-level models that systematically explores parallelization opportunities in the presence of system constraints on various HCGPs are highly desirable.

GPUs can achieve high throughput gain over CPUs when parallel data are abundant; when parallel data is insufficient, however, GPU performance can be inferior to CPU cores. For SDF graphs, such amount of parallel data may not explicitly present; in this case, vectorization is needed for effective exploitation of data parallelism and GPU utilization. However, this situation has been largely neglected by previous research efforts, which assume the SDF models has expressed adequate data parallelism. Because actor firing time scales differently with vectorization on CPU and GPU, throughput optimization problem on HCGPs faces more challenges as it has to take both vectorization and scheduling into consideration. We refer the problem that considers vectorization, scheduling and pipelining together for data, task and pipeline-parallelism exploitation on SDF graphs as the SDF *vectorization-scheduling throughput optimization (VSTO)* problem.

In this thesis, we present novel vectorization and scheduling techniques that aims at solving VSTO under memory constraint. Our contributions are summarized as follows. (1) We formally present the VSTO problem on HCGPs. (2) We propose effective vectorization and scheduling strategy for for VSTO problem. (3) We extend the DIF-GPU integrated framework to analyze, optimize VSTO problem to synthesize throughput efficient implementation on hybrid CPU-GPU platforms. (4) We show that our methods can achieve greater throughput improvement than previous methods under same memory constraint. We demonstrate our approach by applying our methods to *Orthogonal Frequency Division Multiplexing (OFDM)* Receiver (OFDM-RX), a practical wireless communication application.

The details of vectorization and scheduling methods are presented in Chapter 4.

1.1.3 Modeling Parameterized Dynamic Dataflow

For complex embedded applications, implementation at various levels based on multidimensional criteria, is important to meet the design requirements for system functionality. Dataflow modeling and analysis techniques for static dataflow models, such as *Synchronous Dataflow* (*SDF*) [16] and cyclo-static dataflow (*CSDF) [17],* *are well established. However, the static models lack the ability to express dynamism in the application such as parameterization, data-dependent state transition, and reconfiguration.*

The Core Functional Dataflow *(*CFDF) is proposed to express dynamics in complex signal processing applications. CFDF applies the concept of actor “modes”, where different modes can have differing dataflow behavior, and mode transitions can be data-dependent. CFDF is tailored to natural design of actors with dynamic functionality, and facilitates prototyping of dataflow applications, as well as identification of more specialized dataflow behaviors [18].

When using CFDF, a designer specifies the behavior of the different modes of each CFDF actor, and the transitions among these modes. The trend of increasing flexibility in DSP systems has resulted in complex design spaces, involving large sets of user-constraints, system parameters, hardware specifications, and their interaction. Assigning an one-to-one correspondence between an actor’s mode and a configuration will create a large number of modes. As the number of modes grows and the mode transitions become more complex due to increased dynamic behavior in the application, CFDF formulations can become unwieldy in terms of actor specification, analysis and implementation.

In this thesis, our contributions in modeling dynamic dataflow behavior are summarized as follows. (1) We introduce a novel modeling method, called *parameterized sets of modes* (*PSM*), which is a high-level abstraction that efficiently represents parameterized functionality within groups of related modes for CFDF actors. (2) We have developed the formal foundations of PSM-based modeling, and demonstrate its utility through combining with analysis of static scheduling regions [19] and processor selection on CPU-GPU heterogeneous platform. (3) We develop two case studies as examples of PSM applications, involving the mapping of reconfigurable wireless communication functionality into efficient implementations.

The concept of PSM and its application to efficient scheduling are presented in detail in Chapter 5.

1.2 Outline of Thesis

The remainder of this dissertation is organized as follows. Chapter 2 provides background on various topics that are relevant for this research, including dataflow modeling, hybrid CPU-GPU computing platforms, and specific dataflow tools and techniques that our proposed new methods and design framework build upon. In Chapter 3, we present DIF-GPU, which is an integrated, dataflow-based tool for vectorization, scheduling, and software synthesis targeted to hybrid CPU-GPU platforms. The emphasis in this chapter is on the novel capabilities for software synthesis that are incorporated in DIF-GPU. In Chapter 4, we present in detail the models and algorithms for vectorization and scheduling that are used in DIF-GPU. These methods are geared toward optimization of signal processing throughput under memory constraints and constraints on processing resources. In Chapter 5, we present the formalism of parameterized sets of modes (PSMs) and discuss their application to efficient scheduling of dataflow graphs. We conclude in Chapter 6 with a summary of the developments in the thesis, and a discussion of directions for future work.

Chapter 2

Background

In this chapter, we provide background on core concepts that are applied and built upon in the work presented in this thesis.

2.1 Synchronous Dataflow

As briefly described in Chapter 1, a dataflow graph is a directed graph $G = (V, E)$ composed of a set of vertices (*actors*) V and a set of edges E. An actor $v \in V$ represents a computational task of arbitrary complexity. An edge $e = (u, v) \in E$ represents a first-in, first-out (FIFO) data buffer that stores data values (*tokens*) as they are communicated from the output of actor u to the input of v. Tokens represent the basic unit of data that is processed by actors. Dataflow actors are executed in terms of discrete units of execution, called *firings*, of the associated actors.

Synchronous dataflow (SDF) is a specialized form of dataflow in which the numbers of tokens produced by an actor onto each output edge and consumed from each input edge are constant across all firings of the actor [16]. SDF is used widely in the design and implementation of signal processing systems (e.g., see [7]). An important feature of properly-constructed SDF graphs is that they can be executed indefinitely (e.g., on unbounded streams of input data) with bounded memory requirements, which is important for signal processing systems [16]. Such bounded memory execution can be achieved using a scheduling construct called a *valid periodic schedule* or simply *valid schedule*. SDF graphs for which valid schedules exist are called *consistent SDF graphs*. Each edge e in an SDF graph is associated with a constant *production rate* and *consumption rate*, where these "rates" are in terms of tokens per actor firing. These rates are denoted, respectively, as $prd(e)$ and $cns(e)$.

For each actor v in a consistent SDF graph, there is a unique *repetition count* $q(v)$, which gives the minimum number of firings of v in a valid schedule. The vector q of these repetition counts, indexed by the actors in the associated SDF graph, is called the *repetitions vector* of the graph. An SDF actor in DIF-GPU may have *state* that can be changed across firings, and affect the computation of the actor without changing the production and consumption rates. For purposes of analysis in which stateless actors are assumed (e.g., for certain kinds of throughput analysis algorithms), an actor with state can be converted to a stateless actor by modeling the state externally to the actor through a self-loop edge. Here, by a self-loop edge, we mean an edge whose source and sink vertices are the same. We refer to an actor without state as a *stateless actor*, and an SDF graph containing only stateless actors as a *stateless SDF graph*.

2.2 Core Functional Dataflow

As system complexity increases, coarse-grained, dynamic dataflow models have gained increasing significance for their flexibility and their power in exposing high level application structure that is relevant for deriving optimized implementations.

Core functional dataflow(CFDF) is a deterministic sub-class of enable-invoke dataflow (EIDF) [8] in which dynamic functionality in an actor is specified as a set of actor *modes*. More formally, each CFDF actor A is characterized by a nonempty set $M_A = \{m_1, m_2, \ldots, m_n\}$ of modes in which it can execute, and for any given mode $m \in M_A$, the actor A consumes a fixed number of tokens per firing on each input port, and produces a fixed number of tokens per firing on each output port. These production and consumption rates may vary across different modes, but must be constant for any given mode. Each CFDF actor A is also characterized by its *enabling function* ε_A, which determines whether or not, based on a given set of token populations on its input FIFOs, A is enabled. If A has at least one input port, then this enabling function can be viewed as a mapping shown in Equation 2.1:

$$\varepsilon_A : (T_A \times M_A) \rightarrow B, \tag{2.1}$$

where $T_A = N^{|in(A)|}$ denotes the set of all possible buffer populations for input ports of A (assuming some underlying ordering of these ports) [8]. If A has no input ports, then its enabling function reduces simply to the Boolean constant *true*. The CFDF formulation of enabling functions can easily be generalized to take into account finite-capacity output buffers (i.e., by requiring sufficient free space on output buffers before allowing an actor to be fireable).

When A fires, A executes in its current mode, then selects one *next mode* from its set of modes. In each mode, A possesses SDF dataflow behavior, meaning that the production/consumption rates on all actor output/input ports are known, constant values. The next mode determines the mode in which the next actor invocation executes (unless the actor mode is reset or otherwise overridden by the controlling scheduler). The next mode determined during an actor invocation can be fixed (known at compile time) or data dependent.

Although algorithms to compute periodic schedules for static dataflow models have been established, the problem of finding periodic schedules for buffer-bounded execution for CFDF and other dynamic dataflow models is still unsolved. For brevity and clarity, we suppress details of bounded buffer CFDF execution in this thesis, and we simply assume that FIFOs have unbounded token capacity, unless otherwise stated.

2.3 Heterogeneous Computing Platforms

Heterogeneous computing platforms (HCPs) consist of multiple processor types, such as mixed combinations of CPUs and GPUs. In this thesis, we target the class of hybrid CPU-GPU architectures, which is of growing popularity in embedded signal processing systems. Each platform in this class consists of a multicore CPU that is integrated with one or more GPUs. The CPU controls overall execution flow, and is thus referred to as the "host" of the enclosing heterogeneous multiprocessor platform. The GPU receives instructions and data from the CPU, and is referred to as the (acceleration) device.

Each GPU has its own memory (device memory), which is separated from main memory and other device memory. In the class of HCPs that we target in this thesis, a shared bus is used for data transfer between the CPU and the GPUs, and for any data transfer between different GPUs. The CPU can read and write directly on the main memory; however, when data required for a GPU task in an HCP is outside the device memory, the GPU copies the data into device memory via the shared bus. Data transfers between the host and each device are referred to as *host-to-device* or *device-to-host* data transfers depending on the direction. These types of data transfers can result in large overhead that can significantly reduce the performance gain of HCPs [14].

Chapter 3

DIF-GPU Design Framework

Heterogeneous computing platforms with multicore CPUs and GPUs are of increasing interest to designers of embedded signal processing systems since they offer the potential for significant performance boost while maintaining the flexibility of software-based design flows. Developing optimized implementations for CPU-GPU platforms is challenging due to complex, inter-related design issues, including task scheduling, interprocessor communication, memory management, and modeling and exploitation of different forms of parallelism. In this part of thesis, we present an automated, dataflow based, design framework called DIF-GPU for application mapping and software synthesis on heterogeneous CPU-GPU platforms. DIF-GPU is based on novel extensions to the dataflow interchange format (DIF) package, which is a software environment for developing and experimenting with dataflow-based design methods and synthesis techniques for embedded signal processing systems. DIF-GPU exploits multiple forms of parallelism by deeply incorporating efficient vectorization and scheduling techniques for synchronous dataflow specifications, and incorporating techniques for streamlining interprocessor communication. DIF-GPU also provides software synthesis capabilities to help accelerate the process of moving from high-level application models to optimized implementations.

Material described in this chapter has been published in [20].

3.1 Introduction

Driven by continuously growing demand for functionality and performance, many types of embedded signal processing systems now utilize heterogeneous multiprocessor platforms. Among a variety of available classes of heterogeneous platforms, multicore CPU-GPU platforms, which integrate multicore CPUs and GPU devices, have been shown to provide significant performance gains on a wide range of embedded applications. Examples of widely-used CPU-GPU product families are the *NVIDIA Tegra* and *ARM Mali*.

As motivated in Chapter 1 and elaborated on in [7], model-based design methodologies for embedded signal processing help to free developers from low-level, platform-specific fine-tuning, and enable more design effort to be directed to higher-level aspects of system design. However, important challenges must be addressed in the development of model-based design tools that are of practical utility for state-of-the-art CPU-GPU platforms. These challenges include effective methods for analysis and design optimization that incorporate integrated consideration of:

- exploitation of pipeline, data, and task-level parallelism;
- management of interprocessor data transfer and synchronization;
- buffer allocation; and
- task scheduling.

The DIF-GPU design framework that we have developed aims to address these challenges, and provides new methods for synthesis and optimization of embedded software targeted to heterogeneous CPU-GPU platforms. DIF-GPU is based specifically on the synchronous dataflow (SDF) model of computation [16], which is widely used for model-based design of embedded software (e.g., see [7]). DIF-GPU exploits pipeline, task and data parallelism in SDF-based application specifications by systematically integrating actor (dataflow software component) level vectorization, dataflow graph scheduling, dataflow buffer management, interprocessor communication, and software synthesis.

Figure 3.1(a) demonstrates a simple example of an SDF graph. This graph represents an upsampling subsystem that contains a signal source *src*, a signal sink *snk* (e.g., an interface to a memory buffer or file where the upsampled output stream is to be stored), and an upsampler *usp*.

In addition to applying the DIF package, as mentioned above, DIF-GPU also applies the lightweight dataflow environment (LIDE) [21], which provides a programming methodology and associated application programming interfaces (APIs) for implementing dataflow graph actors and edges in a wide variety of platform-oriented languages, such as C, C++, CUDA, and Verilog.

3.2 Related Work

A variety of model-based design frameworks has been explored previously for heterogeneous multiprocessor platforms. For example, StreamIt [22] provides a dataflow-based programming model and design infrastructure for stream process-

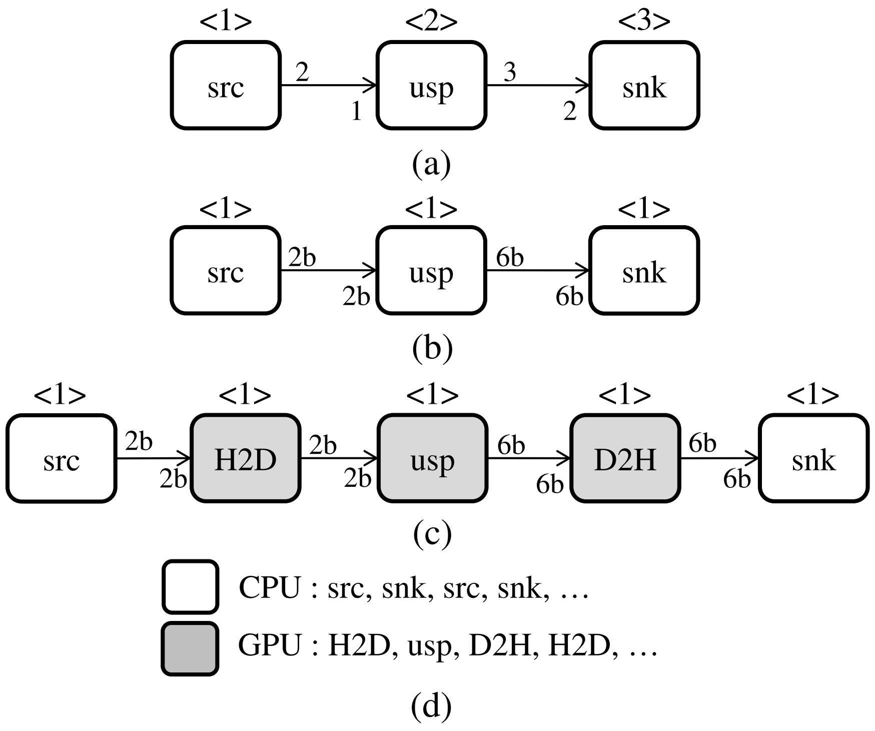


Figure 3.1: An example dataflow application graph. (a) Original SDF graph. (b) Vectorized SDF graph $\nu_b(G)$. (c) $\nu_b(G)$ after insertion of data transfer actors.

ing applications. StreamIt provides dataflow abstractions for computing blocks and facilitates mapping strategies for various computing platforms. The works in [23, 24, 25] extend the StreamIt back-end to support heterogeneous CPU-GPU platforms and develop throughput optimization techniques for dataflow programs running on GPUs. The focal point in these works is to improve the throughput of the GPU kernels generated from dataflow graphs by optimizing GPU memory accesses, register allocation and processor utilization. In this thesis, however, our focus is optimization across heterogeneous platforms that utilize both CPUs and GPUs for computational tasks. Another distinguishing feature of our work is that we allow any number of kernels to be generated and scheduled from a single dataflow graph, while these related works each generate a single kernel from each graph.

A workflow that combines vectorization, a Mixed Integer Programming (MIP) scheduler, and integration with the GNU Radio environment is proposed in [26]. The DIF-GPU framework provides unique capabilities compared to [26] by providing a highly automated, standalone toolset (independent of GNU Radio); its support for multicore CPUs; its incorporation of efficient scheduling heuristics that are adapted for CPU-GPU implementation; and its extensibility for easily integrating other scheduling heuristics.

Various other research efforts have also targeted heterogeneous platforms from dataflow graphs and related models. For example, tools for generating CUDA code from task graph specifications are presented in [27, 28]. In contrast to DIF-GPU, these prior works do not exploit data parallelism resulting from vectorizing dataflow actors. In [29], a dataflow-based tool is built on top of OpenCL to execute applications specified as SDF graphs on heterogeneous systems. This tool provides high-level abstractions for parallel actor invocations and FIFO communication, along with an OpenCL code synthesizer. In this tool, however, exploitation of data parallelism is restricted to stateless SDF graphs. Furthermore, automated scheduling capabilities are not provided for mapping the input SDF graphs. DIF-GPU goes beyond the aforementioned methods and tools through its deep integration of graph- and actor-level vectorization into the scheduling process, and its integration of SDF graph scheduling and software synthesis facilities.

Programming models supporting run-time task scheduling and parallelization on hybrid CPU-GPU platforms have been proposed, such as FastFlow [30] and OmpSS [31]. DIF-GPU is different compared to these approaches in its foundation in formal dataflow semantics, and its integrated emphasis on compile-time dataflow transformations, scheduling, and software synthesis.

3.3 DIF-GPU Framework

DIF-GPU integrates important aspects of synthesizing high performance software targeted to hybrid CPU-GPU platforms from dataflow models. These aspects include vectorization, scheduling, and code synthesis of cooperating C and CUDA subsystems. The DIF-GPU framework is depicted in Figure 3.2. The framework provides systematic approaches for exploiting data, task and pipeline parallelism in the given dataflow model, and managing interprocessor data transfers efficiently. In the following sections, we describe the different steps that comprise the integrated workflow of DIF-GPU.

3.3.1 Dataflow Graph Specification

The workflow in DIF-GPU begins with a dataflow model representing the given DSP application, which is specified in the DIF language. The DIF language, supported as a component of the DIF package introduced in Section 3.1, is a design language for specifying signal processing systems in terms of dataflow models of computation. The DIF language is focused on representing abstract modeling structure in terms of actors, edges, hierarchical subgraphs, and their interconnections and associated model-specific properties, such as production and consumption rates for SDF components, and production and consumption sequences for cyclo-

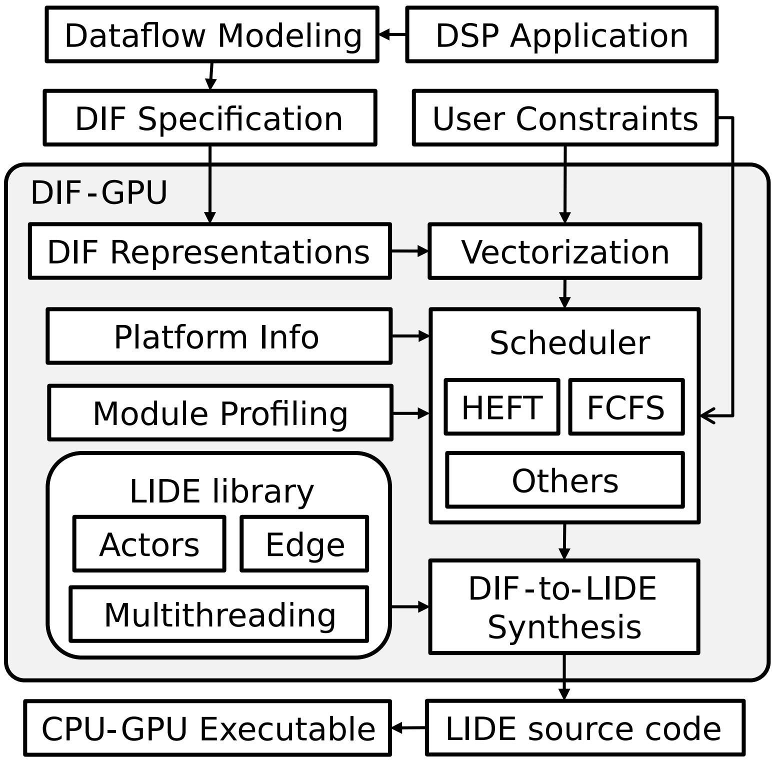


Figure 3.2: The DIF-GPU framework.

static dataflow (CSDF) [17] components. A variety of different dataflow modeling styles, including SDF, CSDF, Boolean dataflow [32], and core functional dataflow [8], are supported in the DIF language. For further background on the DIF language, we refer the reader to [33, 15].

Signal processing applications specified in the DIF language are referred to as *DIF specifications*. DIF-GPU supports only DIF specifications that are based on SDF. Extension of DIF-GPU to work with other forms of dataflow is a useful direction for future work. In DIF-GPU, DIF specifications are first parsed and converted into internal representations in the DIF package. The vectorization and scheduling features of DIF-GPU, as well as the code synthesis capabilities, operate on these internal representations. Figure 3.3 shows the DIF specification of the

```
sdf usp_graph {
    topology {
        nodes = src, usp, snk;
        edges = e1 (src,usp), e2 (usp,snk);
    }
    production {e1 = 2; e2 = 3; }
    consumption {e1 = 1; e2 = 2; }
    attribute edge_type {
        e1 = "float"; e2 = "float";
    }
    actor src {
        name = "src_1f";
        port_0 : OUTPUT = e1;
    }
    actor usp {
        name = "usp3";
        GPU_enabled = 1;
        port_0 : INPUT = e1;
        port_1 : OUTPUT = e2;
    }
    actor snk {
        name = "snk_1f";
        port_0 : INPUT = e2;
    }
}
```

Figure 3.3: A DIF specification of the example graph shown in Figure 3.1(a).

acyclic dataflow graph depicted in Figure 3.1(a), where *usp* has GPU-accelerated implementation, while *src* and *snk* does not.

Currently in DIF-GPU, we assume the input SDF graph is acyclic. A wide variety of practical signal processing systems can be represented as acyclic SDF graphs (e.g., see [7]). For SDF graphs that contain cycles, the delays within cycles impose bounds on the amount of vectorization that can be applied [34]. Investigating systematic approaches for vectorization with cycle- and delay-induced bounds in DIF-GPU is a useful direction for future work.

3.3.2 Implementation of Actors and Edges

To implement the internal functionality of actors and edges in DIF-GPU, we employ the lightweight dataflow environment (LIDE), which was introduced in Section 3.1. In this section, we provide a brief overview of selected aspects of LIDE with emphasis on aspects that are especially relevant to DIF-GPU. For more details on LIDE and its underlying dataflow semantics, which are based on core functional dataflow (of which SDF is a special case), we refer the reader to [21, 35, 8].

Developing an actor in LIDE requires implementation of four methods for the actor — these are called the `new`, `enable`, `invoke` and `terminate` methods. The `new` method performs memory allocation and initialization for the actor. The `enable` method returns a Boolean value indicating the *enable condition* for the actor — i.e., whether there is sufficient data on its input edges, and sufficient empty space on its output edges to support a firing of the actor. Use of the `enable` method can be bypassed if the enable condition can be validated at compile time — e.g., when implementing a static schedule for an SDF graph. The `invoke` method consumes input tokens from the actor input edges, performs the computation associated with a firing of the actor, and produces the resulting output tokens on the actor output edges. LIDE does not place restrictions on the complexity of the invoke method. In DIF-GPU, we parameterize the invoke method by the vectorization degree N that is applied to the actor. Thus, N firings of the original (unvectorized) actor can be executed in parallel on the target GPU through an invocation of the `invoke` method (assuming that there are sufficient resources available on the GPU to support this many parallel firings). The `terminate` method frees memory that has been dynamically allocated for the actor.

The APIs for actor implementation in LIDE are abstract (language-independent), and can be realized in arbitrary actor implementation languages. When the APIs are realized in a particular implementation language XYZ, we refer to the resulting specialized version of LIDE as LIDE-XYZ. In our experiments with DIF-GPU in this chapter, we use LIDE-C for actor implementation targeted to a CPU and LIDE-CUDA for actor implementation targeted to an NVIDIA GPU. This combined use of LIDE-C and LIDE-CUDA provides the actor implementation approach for the CPU-GPU platform that we target in our experiments.

Figure 3.4 shows the LIDE-C/CUDA dataflow structure and interface method declarations of actor *usp* in the example graph.

3.3.3 Vectorization

In DIF-GPU, we apply both *actor-level* and *graph-level* vectorization for effective exploitation of data parallelism. In this chapter, we describe graph-level vectorization, which is conceptually simpler than actor-level vectorization, as a first demonstration of vectorization techniques within the DIF-GPU framework. Our contributions to actor-level vectorization are presented in Chapter 4.

The amount of graph-level vectorization applied is in general a positive integer, which is referred to as the *graph-level vectorization degree* (*GVD*). Use of a GVD in scheduling that is greater than 1 implies scheduling an unfolded version of the

```c
struct usp_context{
#include "actor_context_common.h"
  enable_function_type enable;
  invoke_function_type invoke;
  /* Input and output FIFOs */
  fifo_pointer fifo_input;
  fifo_pointer fifo_output;
  int prod_rate;
  int cons_rate;
  /* State variables */
  /* ... */
};

/* new method */
usp_context* usp_new(fifo_pointer fifo_input,
    fifo_pointer fifo_output,
    int prod_rate, int cons_rate,
    int processor_type);
/* enable method */
boolean usp_enable(usp_context *context);
/* invoke method */
void usp_invoke(usp_context *context);
/* terminate method */
void usp_terminate(usp_context *context);
```

Figure 3.4: Dataflow Structure and interface method declarations of LIDE-CUDA *usp* actor in Figure 3.1(a).

input dataflow graph [36]. This is a specialized form of unfolded scheduling where successive executions of individual actors are constrained to execute in blocks, as determined by the GVD. The vectorization degree of a given actor v in the input dataflow graph is given as $q(v) \times b$, where b is the GVD.

Let b be a GVD that is applied to an SDF graph $G = (V, E)$ in DIF-GPU. Then we derive another SDF graph $\nu_b(G)$, called the the b-vectorized graph of G. The b-vectorized graph may also be referred to simply as the *vectorized graph*. In $\nu_b(G)$, the actors and edges are in one-to-one correspondence with the actors and edges in G, respectively. Each actor v in $\nu_b(G)$ represents a vectorized version of the corresponding actor in G with vectorization degree $b \times q(v)$, where q is the repetitions vector of G. Accordingly, the dataflow rate (production or consumption rate) associated with each actor port in $\nu_b(G)$ is b times the dataflow rate of the corresponding actor port in G. Figure 3.1(b) shows the transformed version of the graph in Figure 3.1(a) after graph-level vectorization is applied.

Note that the repetition count of any actor in a b-vectorized graph is unity, independently of the value of b. In other words, if r represents the repetitions vector of the b-vectorized graph of G, for some $b \geq 1$, then $r(v) = 1$ for every actor v in $\nu_b(G)$. Because the repetition counts are uniformly equal to unity, $\nu_b(G)$ can be scheduled by drawing from the large class of existing *task graph scheduling algorithms* (e.g., see [37]). Here, by a task graph, we mean an acyclic SDF graph in which all actors are fired at the same average rate. DIF-GPU exploits this connection to task graph scheduling for derivation of efficient vectorized schedules. Scheduling techniques in DIF-GPU are discussed further in Section 3.3.4.

In DIF-GPU, we require that the LIDE-CUDA actor implementations employed are vectorized. That is, each actor A should incorporate a positive-integer-valued vectorization parameter $vect(A)$, which specifies the number of successive firings of A that are "treated as a single unit" for scheduling purposes, and effectively transforms A into the vectorized version of A in $\nu_b(G)$.

3.3.4 Scheduling

Dataflow scheduling for heterogeneous platforms is a complex problem. The problem is complicated by differences in actor execution times among different types of processors, and the overhead of interprocessor communication. Although finding optimal schedules in this context is NP-hard, a variety of heuristics has been developed for related task graph scheduling problems [37].

In DIF-GPU, the scheduler takes the vectorized SDF graph $\nu_b(G)$ produced in the vectorization step, and generates a schedule for the given CPU-GPU target platform. $\nu_b(G)$'s schedule encompasses the transient phase and one or more iterations of the periodic phase. The periodic schedule can then be encapsulated within an infinite- or finite-iteration loop to coordinate iterative execution of the application.

As described in Section 3.3.3, $\nu_b(G)$ is in the form of a task graph, and thus, various available task graph scheduling techniques can be applied. Currently, DIF-GPU incorporates the following two scheduling methods.

First Come First Serve (FCFS). The FCFS method is a greedy algorithm that manages a list of actors (the "ready list") that have sufficient data to be executed at the current stage in the scheduling process. As the schedule evolves, the ready list is updated. When a processor becomes available, the scheduler assigns an actor with the shortest execution time in the ready list to that processor. FCFS scheduling has been studied previously in the context of CPU-GPU implementation by Teodoro et al. [38].

Heterogeneous Earliest Finish Time (HEFT) [39]. HEFT is a list scheduling heuristic that takes into account the different running times of the task graph actors as well as data transfer times for the targeted heterogeneous platform. The HEFT scheduler manages a list of actors that are ready to be executed and evaluates, at each scheduling step, all combinations of enabled actors and processors. To schedule an actor, it selects the actor-processor pair with the earliest finish time, and schedules the actor onto the corresponding processor with an insertion-based approach.

In DIF-GPU, each actor has a profile containing its estimated execution time on each type of processor as a function of the vectorization degree. The profile is derived experimentally by running the actor for a selected subset of vectorization degrees and measuring the resulting execution times. During scheduling, the execution time of an actor with a given vectorization degree is estimated using linear interpolation on the actor's profile.

DIF-GPU uses a self-timed scheduling approach for implementing schedules [40]. The timing information used to construct the schedule is discarded before the code generation phase, and only the sequence of vectorized actor firings on each processor is used. At run time, each processor is configured to execute its assigned subset of

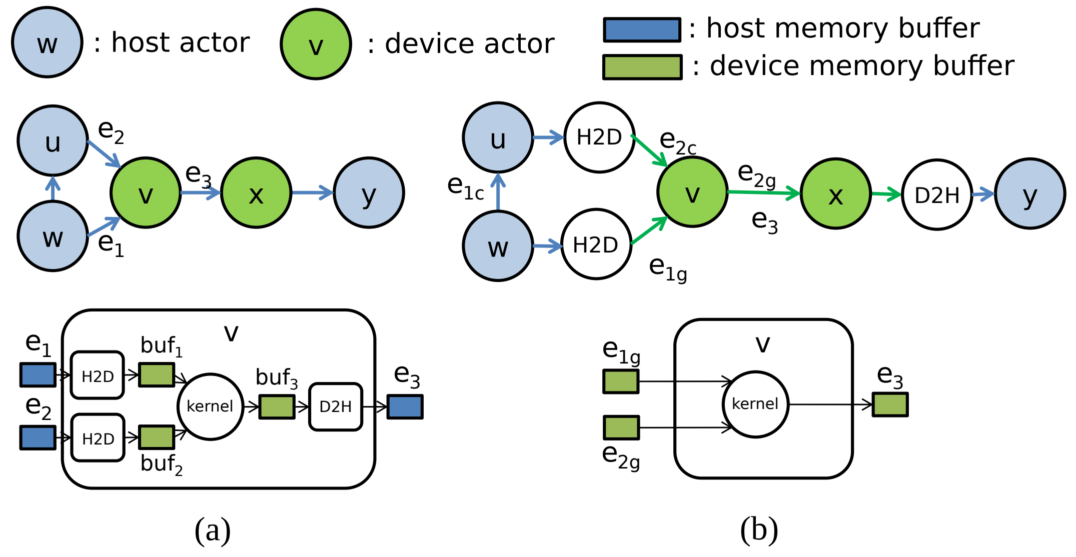


Figure 3.5: Illustration of methods for handling CPU-GPU data transfers in dataflow schedules. (a) Host-centered FIFO allocation. (b) Mapping-dependent FIFO allocation.

actors in the order specified by the schedule, and this order repeats in successive graph iterations.

3.3.5 Managing Interprocessor Data Transfers

As described in Chapter 2, CPUs and GPUs in the targeted class of platforms have separate memory spaces. Dataflow actors mapped to a processor p_1 are not able to access data in the space of another processor p_2 unless the data is first transferred to the space of p_1. To support separate memory spaces in a dataflow design framework, one simple solution is to require data transfers between the memory spaces to be handled within actor implementation, while maintaining all FIFO buffers (associated with the graph edges) in the host memory. We refer to this approach, depicted in Figure 3.5(a), as *Host-Centered FIFO Allocation* (*HCFA*).

HCFA is simple to implement within the DIF-GPU framework, but leads to large amounts of overhead due to excessive CPU-GPU data transfer. In Figure 3.5(a), for example, executing actor v on the GPU using this method requires v to transfer output data from a GPU-allocated buffer buf_3 to a host-allocated FIFO e_3; similarly, executing actor x requires transfer of its input data from e_3 to the internal GPU memory buffer. These two data transfer requirements can significantly reduce performance.

To provide more efficient interprocessor communication, we apply a method that we call *Mapping-Dependent FIFO Allocation* (*MDFA*). In MDFA, FIFOs corresponding to dataflow graph edges are allocated in host or device memory depending on which processor the source actor of an edge is assigned to — that is, the FIFO for edge e is assigned to the memory space of the processor that executes the actor connected to the source of e (rather than unconditionally being assigned to host memory). To handle interprocessor communication, we introduce two special types of actors, *h2d* and *d2h*, which stand for *host-to-device transfer* and *device-to-host transfer*, to explicitly move data between host and device processors.

In DIF-GPU, *h2d* and *d2h* actors are inserted after scheduling and prior to code generation in the DIF-based intermediate representation for software synthesis. Insertion of these actors in illustrated in Figure 3.5(b). In this way, interprocessor data transfer only occurs at locations where such data transfer is necessary, as determined by the schedule.

More specifically, for each application dataflow graph edge $e = (u, v)$ in which u is mapped to the CPU and v is mapped to a GPU, we instantiate an *h2d* actor μ, connect u to μ with an edge e_c, and connect μ to v with an edge e_g. Edge e_c

is implemented using a FIFO buffer that is allocated in host memory, while e_g is implemented using a FIFO in GPU memory. The original edge e is then removed from the internal representation graph. A similar transformation is performed for each edge $e = (u, v)$ in which u is mapped to a GPU and v is mapped to the CPU.

In addition to improving the efficiency of interprocessor communication, use of *h2d* and *d2h* actors in DIF-GPU simplifies the process of actor implementation, as it frees designers from the need to manage details of memory allocation and data transfer associated with interprocessor communication. Furthermore, it is sufficient in the framework to employ only LIDE-CUDA actors in which computational kernels produce and consume data in the same memory space.

3.3.6 Runtime

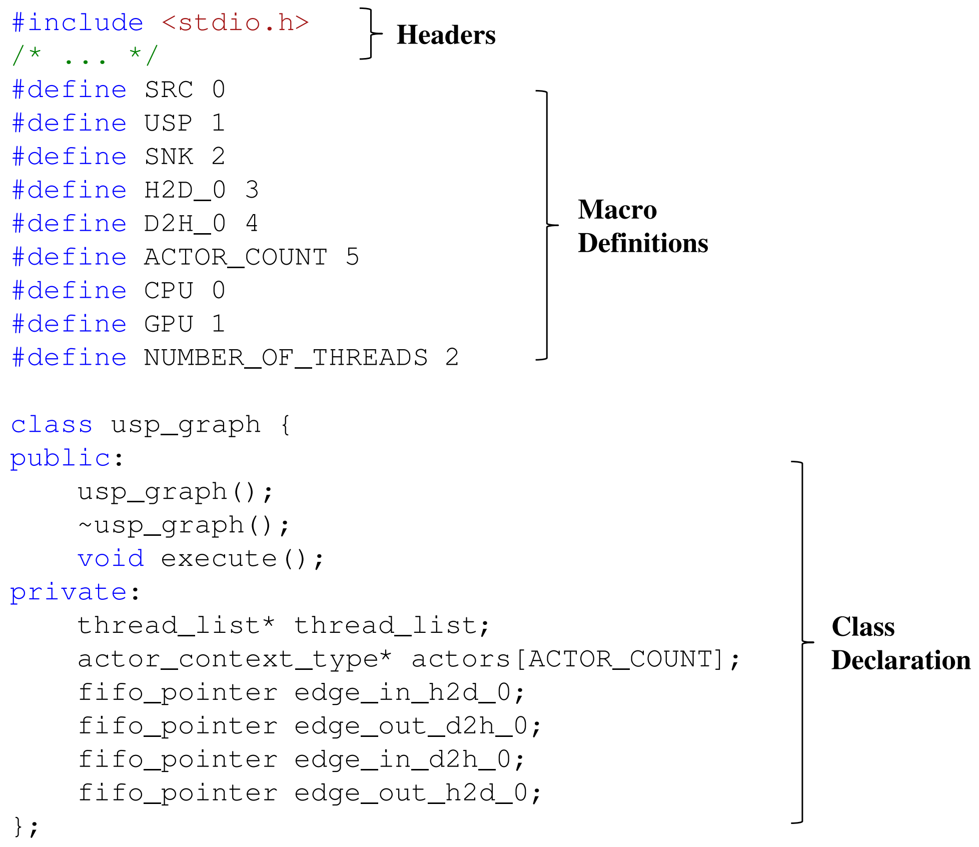
Parallel execution of actor firings on heterogeneous multiprocessors is implemented in DIF-GPU by using POSIX threads. Each processor is assigned a single POSIX thread, and all actors mapped to that processor are fired by the corresponding thread. During execution, each thread monitors the enable conditions of the actors assigned to it (see Section 3.3.2), and fires actors once they are enabled. When an actor finishes execution, the enclosing thread sends a message to other threads notifying them to check the enable conditions for their actors. When a thread has no enabled actors, it is blocked until it is notified by another thread. Blocking of threads that do not have any enabled actors allows DIF-GPU to avoids wasteful “busy-waiting” by threads.

3.3.7 Code Generation

DIF-GPU generates well-structured, human readable source code for compilation with back-end tools associated with the targeted HCP. Given a dataflow application graph G that is provided as input to DIF-GPU, we refer to the resulting synthesized software implementation as the *synthesized package* of G. The synthesized package contains a C++ header file (`.h` file), a C++ implementation file (`.cpp` file), and code that implements the schedule for each processor.

The C++ header and implementation files define a class that encapsulates the computation for G. Graph-level input streams, output streams, and parameters can be applied through constructor arguments. In this manner, DIF-GPU generates an object-oriented module rather than generating a `main` function as the entry point for the derived implementation. Through their modular structure, the implementations generated by DIF-GPU can be integrated flexibly into different design frameworks. This flexibility of integration is useful, for example, for generating DSP components in larger designs where it is not desired to employ dataflow techniques for all parts of the designs.

Figure 3.6 and Figure 3.7 show the synthesized code for the header and implementation files, respectively, when the GVD is set as $b = 2$, and the actor-to-processor assignment shown in Figure 3.1(d) is used. The synthesized C++ class `usp_graph` contains a constructor, a destructor, and an `execute` method. The member variables of the synthesized class include (1) a thread list that contains the threads to execute the dataflow graph on the target platform, as described in



```
#include <stdio.h>
/* ... */
#define SRC 0
#define USP 1
#define SNK 2
#define H2D_0 3
#define D2H_0 4
#define ACTOR_COUNT 5
#define CPU 0
#define GPU 1
#define NUMBER_OF_THREADS 2

class usp_graph {
public:
    usp_graph();
    ~usp_graph();
    void execute();
private:
    thread_list* thread_list;
    actor_context_type* actors[ACTOR_COUNT];
    fifo_pointer edge_in_h2d_0;
    fifo_pointer edge_out_d2h_0;
    fifo_pointer edge_in_d2h_0;
    fifo_pointer edge_out_h2d_0;
};
```

Figure 3.6: Generated header file code for the example of Figure 3.1(a).

Section 3.3.4; (2) a list of pointers to the actor contexts; and (3) a list of pointers to the FIFOs that implement the dataflow graph edges. The *context* of an actor A in LIDE is a data structure that encapsulates relevant details, including pointers to the FIFOs that are associated with the edges incident to A; function pointers to the `enable` and `invoke` methods of A; and parameters and state variables associated with A [35].

3.4 Experiments

In this section, we demonstrate the DIF-GPU framework through experiments using an Intel Core i7-2600K Quad-core CPU with an NVIDIA GeForce GTX680

```
#include "usp_graph.h"
usp_graph::usp_graph(){
    /* Create edges */
    edge_in_h2d_0 = fifo_new(4, sizeof(float), CPU);
    edge_out_d2h_0 = fifo_new(12, sizeof(float), CPU);      FIFO
    edge_in_d2h_0 = fifo_new(12, sizeof(float), GPU);       creation
    edge_out_h2d_0 = fifo_new(4, sizeof(float), GPU);
/* Create actors */
    actors[D2H_0] = (actor_context_type*) memcpy_new(
            edge_in_d2h_0,edge_out_d2h_0,12,12,sizeof(float), GPU);
    actors[SNK] = (actor_context_type*) snk_1f_new(
            edge_out_d2h_0,12, CPU);
    actors[H2D_0] = (actor_context_type*) memcpy_new(                   Actor
            edge_in_h2d_0,edge_out_h2d_0,4,4,sizeof(float), GPU);      creation
    actors[USP] = (actor_context_type*) usp3_new(
            edge_out_h2d_0,edge_in_d2h_0,4,12, GPU);
    actors[SRC] = (actor_context_type*) src_1f_new(
            edge_in_h2d_0,4, CPU);
/* Create schedules of each thread */
    const char* thread_schedules[NUMBER_OF_THREADS] =
            {"thread_0.txt","thread_1.txt"};                            Thread
    thread_list = thread_list_init(NUMBER_OF_THREADS,                  Initialization
        thread_schedules, actors, ACTOR_COUNT);
}
void usp_graph::execute(){                                              Graph
    thread_list_scheduler(thread_list);                                 Execution
}
usp_graph::~usp_graph(){
    /* Terminate threads */
    thread_list_terminate(thread_list);
    /* Free FIFOs */
    fifo_free(edge_in_h2d_0);
    fifo_free(edge_out_d2h_0);
    fifo_free(edge_in_d2h_0);                                           Resource
    fifo_free(edge_out_h2d_0);                                          Deallocation
    /* Destroy actors */
    memcpy_terminate((memcpy_context_type*)actors[D2H_0]);
    snk_1f_terminate((snk_1f_context_type*)actors[SNK]);
    memcpy_terminate((memcpy_context_type*)actors[H2D_0]);
    usp3_terminate((usp3_context_type*)actors[USP]);
    src_1f_terminate((src_1f_context_type*)actors[SRC]);
}
```

Figure 3.7: Generated implementation file code for the example of Figure 3.1(a).

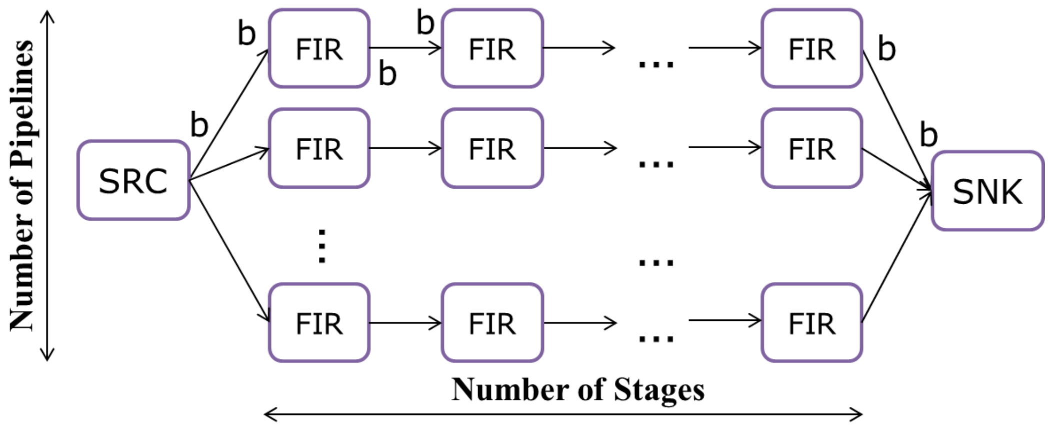


Figure 3.8: An illustration of the vectorized MP-Sched benchmark.

GPU running Ubuntu Linux 12.04. LIDE-C and LIDE-CUDA code for the actor implementations is compiled using GCC 4.6.3 and the NVIDIA CUDA compiler (NVCC) 7.0, respectively.

3.4.1 Benchmarks

In our experimental evaluation, we employ MP-Sched, which is a parameterized family of benchmarks [26] that is representative of an important class of digital processing subsystems for wireless communications, and was designed originally for use with the GNU Radio environment [41]. MP-Sched is illustrated in Figure 3.8. The MP-Sched benchmarks are defined by a parameterized signal processing flow graph that consists of a grid of $P \times S$ finite impulse response (FIR) actors, where P is the number of signal processing pipelines and S is the number of stages in each pipeline.

We apply graph-level and actor-level vectorization, as discussed in Section 3.3.3, to optimize GPU-based execution of MP-Sched. If b denotes the GVD, then each vectorized FIR filter actor consumes b tokens from its input edge and produces b tokens on its output edge. Additionally, the *SRC* actor produces b tokens on each output edge, and the *SNK* actor consumes b tokens from each input edge.

The operation employed in the FIR filter actors is given as follows:

$$y[n] = \sum_{k=0}^{L-1} W[k]x[n-k], \tag{3.1}$$

where x is the input, y is the output, W is an array of filter coefficients, and L is the number of coefficients. In our experiments, we use $L = 8$. Equation 3.1 involves data-parallel operations that can be efficiently exploited on a GPU (e.g., see [42]). The vectorized LIDE-CUDA FIR actor used in this benchmark is implemented with GPU-acceleration so that data-parallelism can be exploited using actor-level vectorization, as discussed in Section 3.3.3.

The topology of this benchmark also exhibits task-level parallelism, as actors from different pipelines can be executed concurrently on different processors. To evaluate the performance gain achieved by using the DIF-GPU framework, we define the throughput of an implementation of the MP-Sched benchmark as:

$$Th = b/T, \tag{3.2}$$

where b is the GVD used in the implementation, and T is the time to complete one iteration of the schedule of the vectorized graph. The units of this throughput metric are SDF graph iterations (relative to the original, unvectorized graph) per unit time. Table 3.1 shows the benchmark, platform, and scheduler configurations

Table 3.1: Benchmark, platform and scheduler configurations. Here "CC" stands for "CPU Core".

Item	Value
Grid Size	2x5, 4x4, 6x3
Platform	1 CC + 1 GPU, 3 CCs + 1 GPU
Scheduler	HEFT, FCFS

used in our experimentation with the DIF-GPU Framework.

3.4.2 Evaluation of Actor Performance and Data Transfer Cost

Figure 3.9 shows the speedups measured for the LIDE-CUDA-based FIR actor implementation running on a GPU for a wide range of vectorization degrees (from 2^7 to 2^{19}). The speedups reported here do not include the cost of CPU-GPU data transfer, as data transfer is handled separately in this framework (i.e., through the software synthesis process rather than through the actor implementation process). From these results, we see that the speedup increases gradually when the vectorization degree b ranges from 2^7 to 2^{10}, and then rapidly from 2^{11} and 2^{16}, as larger numbers of SIMD processors are utilized to process increasing amounts of data. The speedup saturates as b approaches 2^{17}. When $b \leq 2^8$, the actor runs more slowly compared to the CPU when mapped onto a GPU because the resources in the GPU are significantly under-utilized.

To evaluate the data transfer overhead of applying the HCFA and MDFA techniques for FIFO allocation in DIF-GPU, we map the SRC and SNK actors on the CPU and all of the FIR actors on the GPU, and we apply a vectorization degree of $b = 4096$. Table 3.2 compares the resulting throughput and percentage

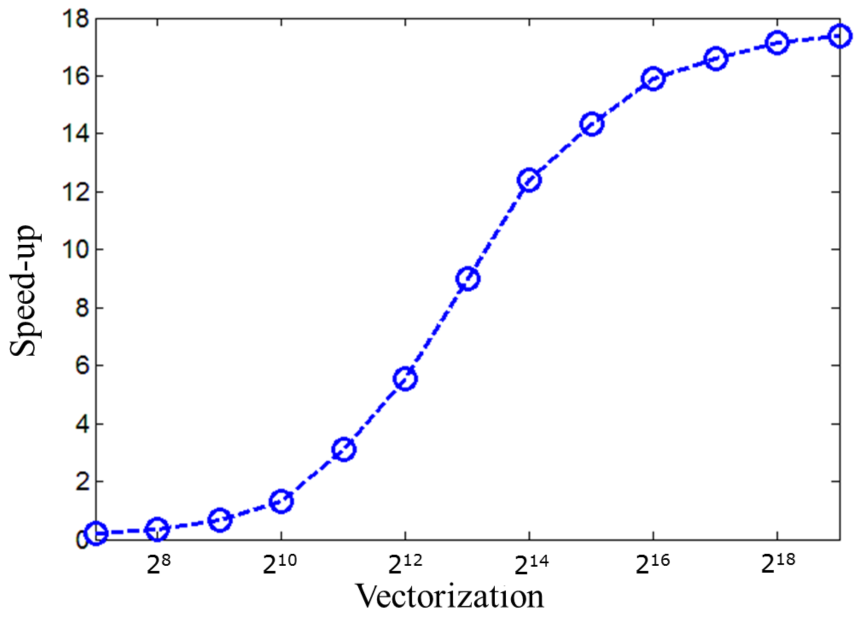


Figure 3.9: Speedup of the FIR filter actor.

Table 3.2: Throughput and data transfer overhead for FIFO implementation based on HCFA and MDFA.

Topology		2 x 5	4 x 4	6 x 3
HCFA	$Th(10^6/s)$	4.80	2.84	2.52
	D2H	37.4%	37.6%	37.1%
	H2D	16.4%	16.2%	15.8%
MDFA	$Th(10^6/s)$	6.71	4.06	3.59
	D2H	17.2%	15.5%	20.8%
	H2D	6.7%	9.9%	8.2%

of time spent on H2D and D2H data transfers. These results show that the MDFA strategy for FIFO allocation improves application throughput by a large margin — 40.0%, 43.0% and 42.5% on 2x5, 4x4 and 6x3 MP-Sched configurations, respectively. Thus, incorporating MDFA within the DIF-GPU software synthesis framework helps significantly to streamline interprocessor communication and synchronization within the framework.

3.4.3 System-Level Evaluation

We evaluate the performance gain achieved by applying DIF-GPU with different benchmark (MP-Sched grid dimensions), scheduler, and platform configurations. We carry out this evaluation by comparing the achieved throughput to that measured from single-core baselines. These evaluations are performed using the different system configurations summarized in Table 3.1.

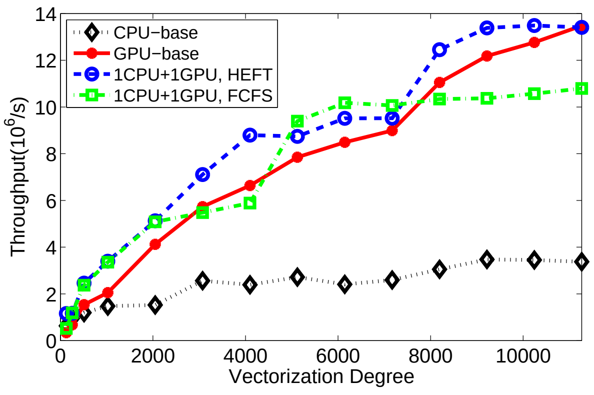
Figure 3.10 shows the throughput achieved for different system configurations. Here, the "single CPU baseline" (CPU baseline) maps all actors onto a single CPU core, while the "single GPU baseline" (GPU baseline) maps all FIR actors on the GPU and maps the SRC and SNK actors onto a single CPU core. The GPU baseline does not provide a throughput gain over the CPU baseline when $b \leq 256$ for the 2x5 and 4x4 MP-Sched benchmarks, and when $b \leq 512$ for the 6x3 benchmark. We expect that this is because the limited data block size prevents the GPU from achieving sufficient speedup from data-parallelism to overcome the data transfer overhead that is incurred. As b increases, the amount of available data parallelism increases accordingly, and the GPU baseline overtakes the performance of the CPU baseline. The maximum measured speedups of the GPU baseline compared to the CPU baseline are 4.0, 4.2, and 3.7, respectively, for the 2x5, 4x4, and 6x3 benchmark configurations.

The limited speedup of the GPU baseline over the CPU baseline is due to the range of vectorization degrees selected for our experiment, where $b \leq 11,264$. For $b > 11,264$, the GPU speedup continues to increase to significantly higher values.

However, in this range of b values — where the GPU has very large speedups — the targeted *hybrid* platform (multi-core CPU with single GPU) would not provide significant advantage over the GPU-baseline. Since the emphasis in this work is on techniques for hybrid CPU-GPU platforms, we focus on this range of smaller b values to demonstrate the joint utilization of CPU cores and the GPU device in our framework. Such smaller b values may be preferable, for example, due to system constraints on latency and buffer sizes.

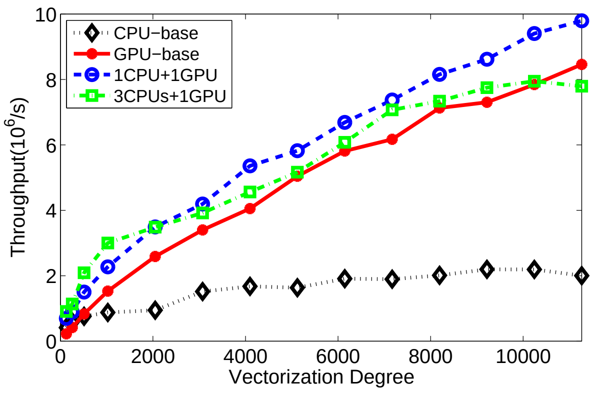
The throughput data shown in Figure 3.10 is averaged from measurements that are repeated 50 times for each configuration. Figure 3.11, 3.12 and 3.13 show box plots of throughput for different configurations under 2x5, 4x4 and 6x3 FIR filter grids, respectively. We observe that under some configurations, the throughput measurements show significant variances. These large variances have occurred under different target platforms, schedulers and FIR grid sizes. We expect that the throughput variations result from multiple factors across the entire operating system (OS) / hardware platform environment, including cache operations, multi-tasking and storage management.

To help demonstrate the utility of DIF-GPU in comparing CPU + GPU configurations over CPU-only or GPU-only solutions, we combine the CPU baseline and GPU baseline together into one "single processor" baseline by taking the maximum throughput of the CPU-only and GPU-only mappings. For small vectorization degrees, 3 CPU cores + 1 GPU generally provide the most throughput improvement over the single processor baselines, as the CPU cores are comparable to the GPU in terms of computational power when the data size is small. This improvement



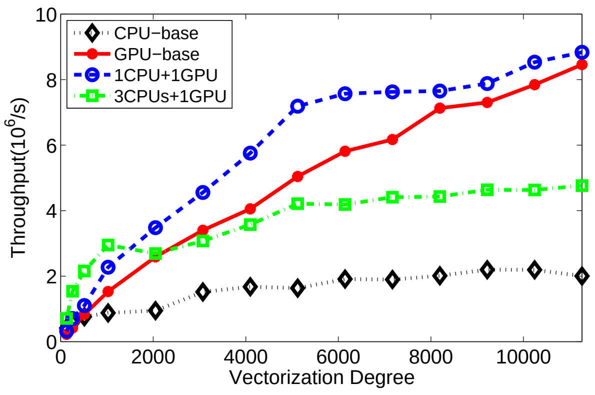
CPU−base
GPU−base
1CPU+1GPU, HEFT
1CPU+1GPU, FCFS
Throughput(10^6/s)
Vectorization Degree
0
2000
4000
6000
8000
10000
0
2
4
6
8
10
12
14

(a)



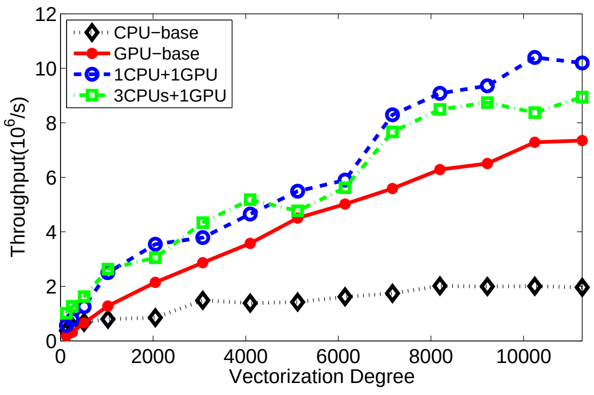
CPU−base
GPU−base
1CPU+1GPU
3CPUs+1GPU
Throughput(10^6/s)
Vectorization Degree
0
2000
4000
6000
8000
10000
0
2
4
6
8
10

(b)

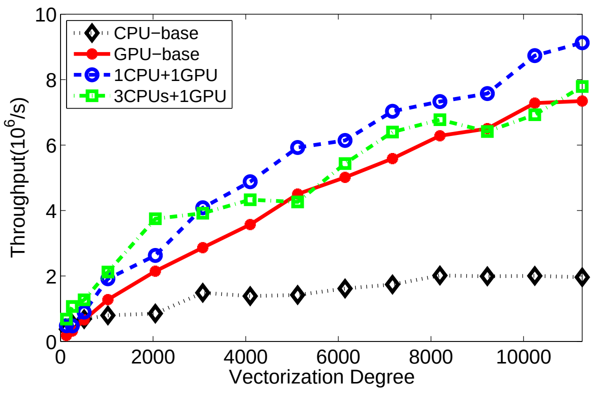


CPU−base
GPU−base
1CPU+1GPU
3CPUs+1GPU
Throughput(10^6/s)
Vectorization Degree
0
2000
4000
6000
8000
10000
0
2
4
6
8
10

(c)

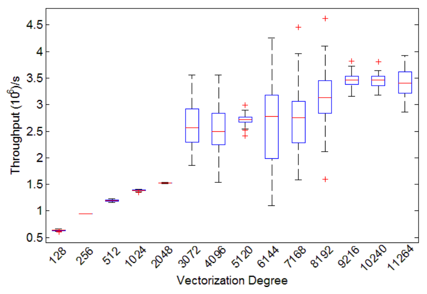


(d)

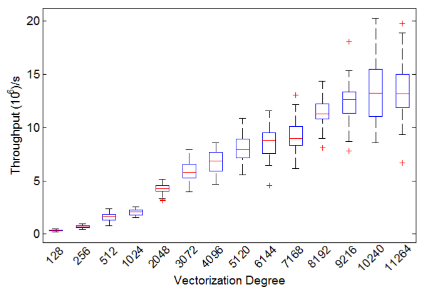


(e)

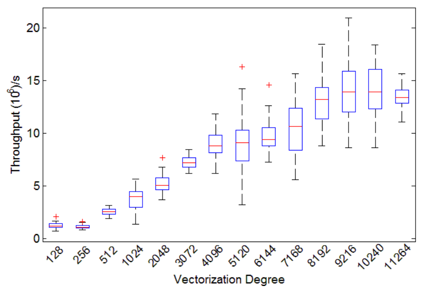
Figure 3.10: Throughput of $M \times N$ MP-Sched benchmarks for different platform configurations and schedulers. (a) 2x5 with FCFS and HEFT; (b) 4x4 with HEFT; (c) 4x4 with FCFS; (d) 6x3 with HEFT; (e) 6x3 with FCFS.



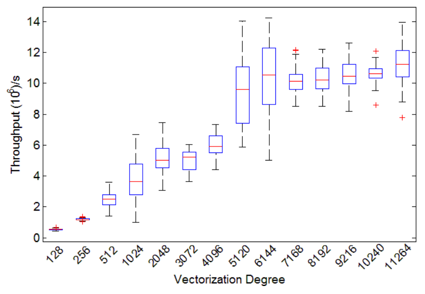
(a)



(b)



(c)

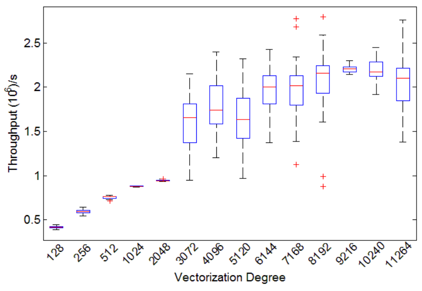


(d)

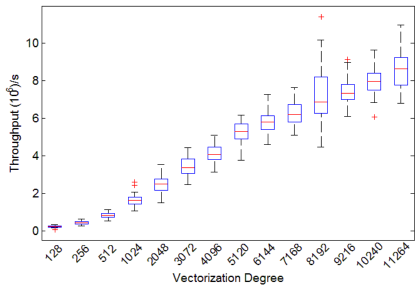
Figure 3.11: Throughput box plots of 2×5 MP-Sched benchmarks with different platform configurations and schedulers: (a) CPU-base; (b) GPU-base; (c) 1 CPU + 1 GPU, HEFT scheduler; (d) 1 CPU + 1 GPU, FCFS scheduler.

by employing more CPU cores can be seen for the (4x4, HEFT), (4x4, FCFS), and (6x3, EFT) configurations when $b \leq 1024$, and for (6x3, FCFS) when $b \leq 2048$. The maximum speedups measured over the single processor baseline are 2.5x and 2.0x for the 4x4 and 6x3 MP-Sched benchmarks, respectively.

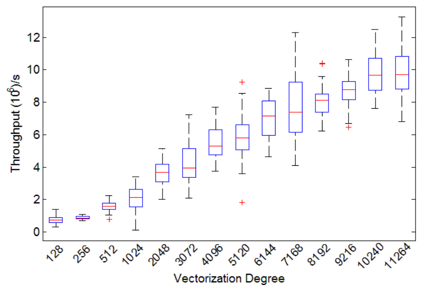
For large vectorization degrees ($b > 2048$), 1 CPU + 1 GPU generally achieves maximum throughput rather than the 3 CPU cores + 1 GPU configuration, as shown in Figure 3.10(b)–(e). The maximum measured throughput improvements of 1 CPU + 1 GPU over the GPU baseline are 33%, 43% and 65%, respectively, for the 2x5, 4x4 and 6x3 MP-Sched benchmarks. In some cases, 3 CPU cores + 1 GPU cannot provide consistent improvement over the GPU baseline when the FCFS scheduler is used, as shown in Figure 3.10(c) and 3.10(e).



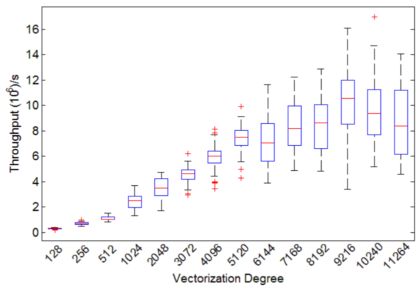
(a)



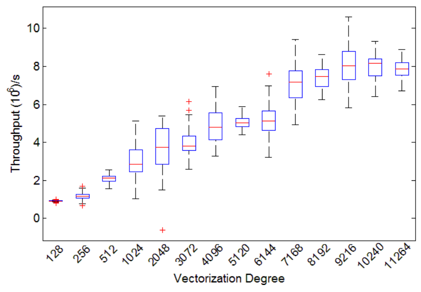
(b)



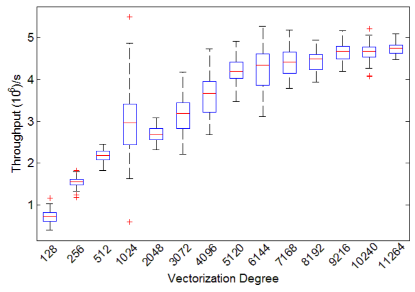
(c)



(d)

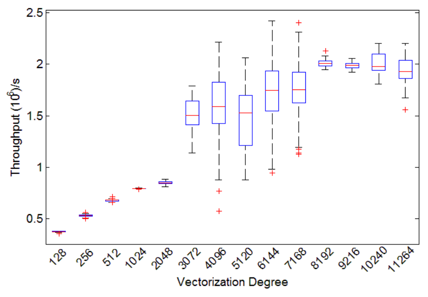


(e)

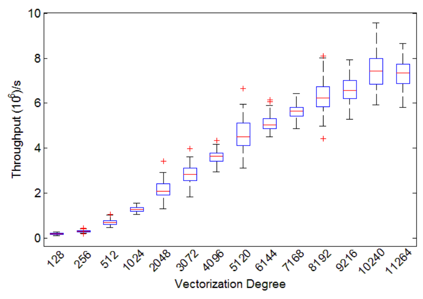


(f)

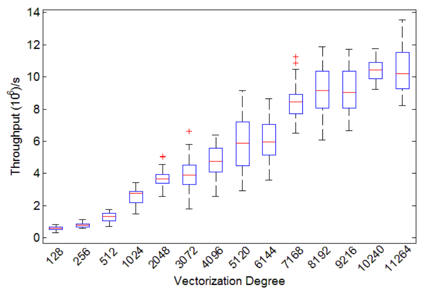
Figure 3.12: Throughput box plots of 4×4 MP-Sched benchmarks with different platform configurations and schedulers: (a) CPU-base; (b) GPU-base; (c) 1 CPU core + 1 GPU, HEFT scheduler; (d) 1 CPU core + 1 GPU, FCFS scheduler; (e) 3 CPU cores + 1 GPU, HEFT scheduler; (e) 3 CPU cores + 1 GPU, FCFS scheduler.



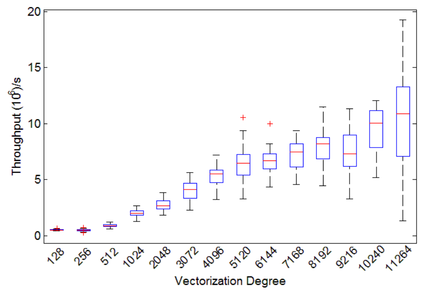
(a)



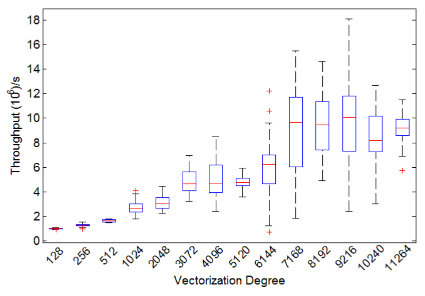
(b)



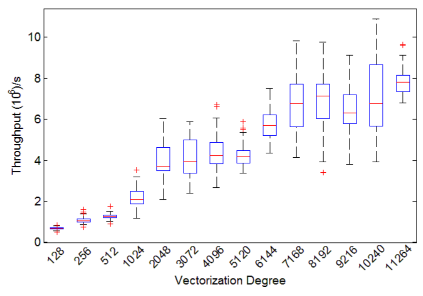
(c)



(d)



(e)



(f)

Figure 3.13: Throughput box plots of 6×3 MP-Sched benchmarks with different platform configurations and schedulers: (a) CPU-base; (b) GPU-base; (c) 1 CPU core + 1 GPU, HEFT scheduler; (d) 1 CPU core + 1 GPU, FCFS scheduler; (e) 3 CPU cores + 1 GPU, HEFT scheduler; (e) 3 CPU cores + 1 GPU, FCFS scheduler.

In summary, implementation of the MP-Sched benchmarks involves complex trade-offs among resource utilization (the number and types of processors employed), the vectorization degree, and the achieved throughput. In this section, we have demonstrated the utility of DIF-GPU in exploring such trade-offs through its automated and flexibly-configurable software synthesis capabilities.

3.4.4 Scheduling

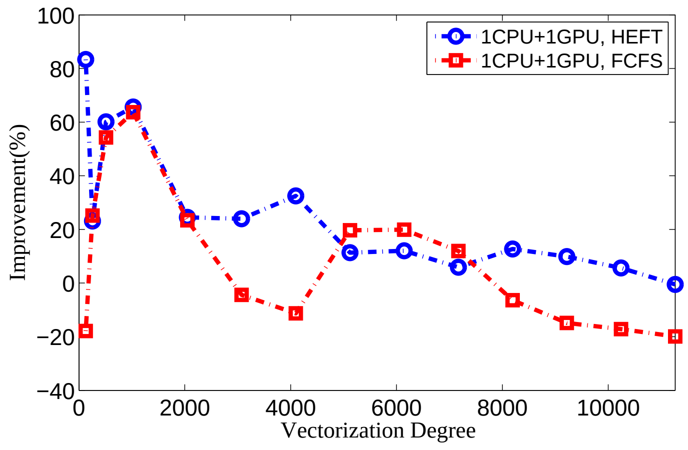
To experiment with the effect of the scheduling strategy in DIF-GPU, we compare the throughput improvement of the HEFT and FCFS schedulers over the GPU baseline, as shown in Figure 3.14. For the 1 CPU + 1 GPU target, HEFT achieves greater improvement when the vectorization degree is smaller than some lower threshold b_l, or greater than an upper threshold b_u, depending on the benchmark grid dimensions. For $b_l \leq b \leq b_u$, FCFS achieves greater improvement. For the 3 CPU cores + 1 GPU target, HEFT consistently performs better than FCFS.

We would like to emphasize here that DIF-GPU is not limited to use of HEFT or FCFS as the framework can readily be extended with other scheduling techniques. The objective of these experiments is to demonstrate the utility of DIF-GPU in exploring complex design spaces involving different combinations of platforms, scheduling strategies, vectorization degrees, and levels of application complexity.

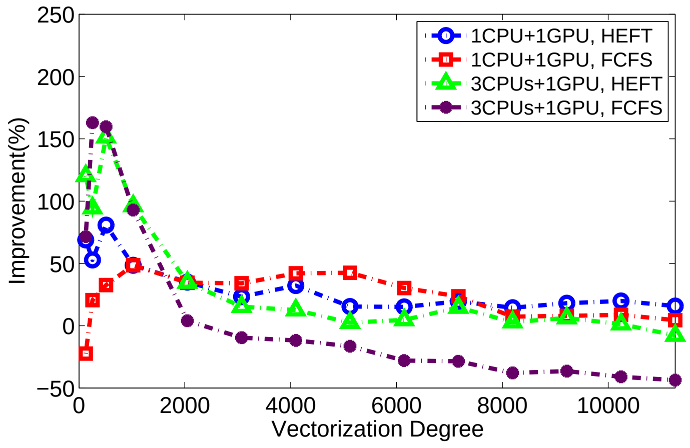
The overall results indicate that in the investigated design space, neither scheduler is uniformly better in terms of throughput improvement; the preferred scheduler depends on the vectorization, application (benchmark), and platform (VAP) configurations. However, HEFT is more consistent in providing throughput gains across different regions of the design space. Among all of the VAP combinations evaluated, HEFT provides throughput improvement on all combinations except for: (a) 1 CPU + 1 GPU with 2x5 MP-Sched, and $b \in \{5{,}120, 6{,}148, 7{,}192\}$; and (b) 3 CPU cores + 1 GPU with 4x4 MP-Sched, and $b = 11{,}264$. Intuitively, we expect that this is because HEFT takes interprocessor data transfer time into account, and is able to keep slower processors (in this case the CPU) idle and select the processor that is able to finish a task at the earliest possible time, even if that processor happens to be busy at the current time.

3.5 Summary

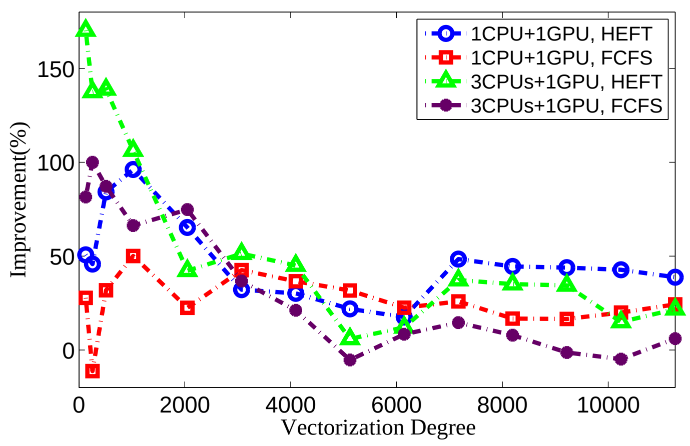
In this chapter, we have presented a new model-based software synthesis framework, called DIF-GPU, that integrates high level dataflow graph specification, vectorization, scheduling, and code generation for heterogeneous CPU-GPU platforms. We have demonstrated the ability of DIF-GPU to synthesize, through its highly integrated design flow, implementations that significantly outperform conventional CPU-GPU mappings (i.e., where all actors for which GPU implementations are available are unconditionally mapped to the GPU). Furthermore, we have demonstrated the utility of DIF-GPU in (a) enhancing application performance through optimized management of interprocessor communication for given scheduling and vectorization configurations, and (b) exploring complex design spaces in the mapping of applications onto CPU-GPU platforms.



(a)



(b)



(c)

Figure 3.14: Speedup of MP-Sched benchmarks using the HEFT and FCFS schedulers. (a) 2x5, (b) 4x4, (c) 6x3.

Chapter 4

Memory-Constrained Vectorization and Scheduling

The increasing use of heterogeneous embedded systems with multi-core CPUs and GPUs presents important challenges in effectively exploiting pipeline, task and data-level parallelism to meet throughput requirements of DSP applications. Hand optimization of code to satisfy these requirements is inefficient and error-prone, and can therefore, greatly slow down development time or result in highly underutilized processing resources. In this chapter, we present dataflow graph vectorization and scheduling methods to effectively exploit multiple forms of parallelism for throughput optimization on hybrid CPU-GPU platforms, while considering system-level memory constraints. We demonstrate that our methods provides significantly improved throughput compared to previous approaches under a given memory constraint. We also present a practical case-study of applying our methods to the implementation of an orthogonal frequency division multiplexing (OFDM) receiver system.

4.1 Introduction

Heterogeneous multiprocessor platforms are of increasing relevance in the design and implementation of many kinds of embedded systems. Among these platforms, *heterogeneous CPU-GPU platforms (HCGPs)*, which integrate multicore central processing units (CPUs) and graphics processing units (GPUs), have been shown to significantly boost throughput for many applications. System-level performance optimization requires efficient utilization of both CPU cores and GPUs on HCGPs. In embedded system designs, multiple system constraints must be met including memory, latency or cost requirements. Manual performance tuning on a case-by-case suffers from inefficiency and can lead to highly sub-optimal solutions. When system constraints or the target platforms are changed, the designer often needs to repeat the same process, which further reduces development productivity, and increases the chance of introducing implementation errors. Therefore, methods for HCGPs that are based on high-level models, and systematically explore parallelization opportunities are highly desirable.

GPUs in HCGPs accelerate computational tasks by supporting large-scale data parallelism with hundreds or thousands of SIMD (single instruction multiple data) processors. GPUs can achieve high throughput gain over CPUs when parallel data is abundant. However, when parallel data is insufficient, GPU performance can be worse compared to CPU cores. For an SDF graph, a sufficient amount of parallel data may not be present to effectively utilize a GPU. In this case, vectorization can be of great utility in improving the degree of exposed data parallelism, and the effective utilization of GPU resources. However, previous research on scheduling and software synthesis from SDF graphs has focused largely on task and pipeline parallelism, therefore providing inadequate support of GPU-targeted design flows. The developments in this chapter are intended to address this gap.

In general, the average time required for an actor firing scales differently in terms of the vectorization factor between a CPU and GPU. Additionally, overheads involving interprocessor communication and synchronization can limit or even negate performance gains achieved through vectorization. Thus, effective throughput optimization for HCGPs requires rigorous joint consideration of vectorization and scheduling. In this chapter, we develop techniques for software synthesis targeted to HCGPs that jointly consider vectorization and scheduling for through optimization of SDF graphs. We refer to this problem of joint vectorization and scheduling as the SDF *vectorization-scheduling throughput optimization* (*VSTO*) problem, or simply as "VSTO".

Our contribution is summarized as follows. First, we formally present the VSTO problem for HCGPs. Second, we develop a novel vectorization and scheduling strategy for the VSTO problem. Third, we demonstrate our approach to VSTO by applying them to an Orthogonal Frequency Division Multiplexing (OFDM) receiver, and to a large collection of synthetic, randomly-generated dataflow graphs.

4.2 Related Work

SDF throughput analysis under resource constraints using explicit state space exploration has been studied in [43]. In [44], the authors present a scheduling algorithm for SDF graphs that applies static topological analysis and vectorization to improve SDF throughput and memory usage on shared-memory, multicore platforms. In [45], a buffer optimization technique for pipelined, multicore schedules is discussed.

Earlier work on SDF vectorization has focused on throughput optimization for single-processor implementation on programmable digital signal processors, and more recently, on multicore implementation. SDF vectorization techniques to maximize throughput for single-processor implementation were developed in [34]. In [46], the authors presented methods to construct vectorized, single-processor schedules that optimize throughput under memory constraints. In [47], the authors presented techniques for maximizing throughput when simulating SDF graphs on multicore platforms. These techniques simultaneously optimize vectorization, inter-thread communication, and buffer memory management. In these works, SIMD architectures are not involved, and vectorization is applied to reduce synchronization overhead and context switching rather than to exploit data-parallelism.

Various studies have targeted automated exploitation of parallelism to map dataflow models onto heterogeneous computing platforms. Design tools that exploit various forms of parallelism using CUDA or OpenCL have been developed in [27, 48, 29]. These tools assume that vectorization has been specified by the designer, and map an actor onto a GPU whenever a GPU-accelerated implementation of the actor is available. For such actors, these tools do not take into account the possibility that CPU-targeted execution may be more efficient. In [26], SDF graphs are automatically vectorized, transformed to single-rate SDF graphs, and then scheduled using Mixed-Integer Programming techniques. However, this approach does not take memory constraints into account. Intuitively, a single-rate SDF graph is one in which all actors are fired at the same average rate. This concept is discussed in more detail in Section 4.3.

When SDF graphs are converted to single-rate graphs, they can be scheduled in the same way that task graphs are scheduled in programming environments such as StarPU [49], FastFlow [30], and OmpSS [31]. These environments support run-time task graph scheduling and parallelization on hybrid CPU-GPU platforms. StarPU, for example, uses the *Heterogeneous Earliest Finish Time* (*HEFT*) heuristic to schedule tasks on HCGPs. However, these programming models cannot directly be applied to multirate SDF graphs; a designer must manually vectorize the graph and convert it to a single-rate SDF graph before working with it in such environments. In addition to requiring such manual transformation, this process limits the flexibility in vectorization and scheduling for SDF execution, which can lead to inefficient memory usage and execution time performance.

In this part of thesis, we go beyond the previous works by jointly considering SDF vectorization and scheduling for HCGPs under memory constraints. To our knowledge, our work is the first to take memory constraints into account in the context of SDF vectorization and scheduling for heterogeneous computing platforms. Our methods are not restricted to single-rate SDF graphs, and are capable of deriving efficient, memory-constrained vectorization configurations for acyclic SDF graphs. Acyclic SDF graphs are suitable for a modeling broad class of applications across many areas of signal processing (e.g., see [7]). Furthermore, we present in this chapter the DIF-GPU framework, which integrates vectorization, scheduling and software synthesis processes for a highly automated workflow. DIF-GPU incorporates techniques for minimizing runtime overhead through compile-time scheduling and incorporation of carefully-designed protocols for interprocessor communication.

The work presented in this chapter extended Chapter 3 significantly by developing:

- memory constrained actor vectorization methods,
- scheduling methods for multirate SDF graphs on HCGPs,
- experiments on a large number of synthetic SDF graphs, and
- a case study of an OFDM receiver application.

These new developments are integrated with DIF-GPU so that implementations that employ these new vectorization techniques can be generated automatically through software synthesis. These new vectorization-integrated software synthesis capabilities are applied in our experimental evaluation (Section 4.6 and Section 4.7).

4.3 Background

The HCGPs that we target in this chapter consist of one multi-core CPU and one GPU each. This class of single CPU / single CPU architectures is widely used in embedded systems. In our targeted class of HCGPs, we refer to the CPU as the *host*, as it controls overall execution flow and manages the associated GPU, and we refer to the GPU as the *device*. The device receives instructions and data from the host.

Additionally, in the target architecture model, the CPU can read and write directly from/to main memory, while the GPU has its own device memory, which is separate from the main memory. When data required for a processor is outside of its directly-accessible memory, the processor copies the data from the other processor's memory to its own memory through a shared bus. Such data copying between the host and the device is referred to as *host-to-device (H2D)* or *device-to-host (D2H)* data transfer, depending on the direction. H2D data transfer can result in large overhead and significantly reduce the performance gain of HCGPs [14].

Signal processing systems represented as SDF graphs are often required to be executed indefinitely — that is, iterated through a number of iterations for which no useful bound is known in advance. To support such indefinite execution, the concepts of consistency and periodic schedules in SDF graphs are important [16]. An SDF graph is *consistent* if it has a *periodic schedule*, which is a sequence of actor executions that does not deadlock, fires each actor at least once, and produces no net change in the number of tokens on each edge. Consistent SDF graphs can be executed indefinitely with finite buffer memory requirements. Furthermore, for each actor $v \in V$ in a consistent SDF graph $G = (V, E)$, there is a unique *repetition count* $q(v)$, which gives the minimum number of firings of v in a periodic schedule. We call a set of actor firings in which each actor v fires exactly $q(v)$ times an *iteration* of G. If $q(v) = 1$ for every actor $v \in V$, then G is called a *single-rate* SDF graph. We refer to an acyclic single-rate SDF graph as a *task graph*. A wide variety of algorithms have been developed for scheduling task graphs onto multiprocessor systems (e.g., see [37]). In cases where the graph G may not be understood from context, we apply a minor abuse of notation and represent the repetition count of an actor v by $q(G, v)$.

Given an SDF graph $G = (V, E)$ and an actor $v \in V$, we denote the sets of input and output edges of v as $in(v)$ and $out(v)$, respectively. Given an edge $e \in E$, we denote the source and sink actors of e by $src(e)$ and $snk(e)$, respectively. We denote as $prd(e)$ the number of tokens produced onto e by each firing of $src(e)$, and similarly, we denote as $cns(e)$ the number of tokens consumed from e by each firing of $snk(e)$. For implementation of G, we assume a static buffer allocation model, where we allocate a FIFO buffer of fixed, finite size ("buffer bound") $\text{size}(e)$ for each edge $e \in E$. When an actor v fires, it must satisfy that (1) for each edge $e_i \in in(v)$, e_i contains at least $cns(e_i)$ tokens, and (2) for each edge $e_o \in out(v)$, e_o contains no more than $(\text{size}(e_o) - prd(e_o))$ tokens. When this condition is met, the actor is said to be *bounded-buffer fireable*, and SDF graph execution following this rule is called *bounded-buffer execution*.

As mentioned in Section 4.2, we assume in this chapter that the input SDF graphs for vectorization and software synthesis are acyclic. Extension of the methods in this chapter to SDF graphs that contain cycles is a useful direction for future work.

We represent the individual processors in the target multiprocessor platform as $P = \{p_1, p_2, \ldots, p_N\}$, where $p_1, p_2, \ldots, p_{N-1}$ represent the available CPU cores, and p_N represents the GPU. When scheduling G onto the platform, actor firings are assigned to processors to be executed. In this context, we say that an actor $v \in V$ is *mapped* onto processor $p \in P$ if all firings of v are assigned to execute on p.

4.4 Problem Formulation

In this section, we formally define the VSTO problem for HCGPs. We begin by defining the concept of actor-level vectorization. Given a consistent SDF graph $G = (V, E)$, and an actor $v \in V$, the *vectorization* of v by a *vectorization degree* (*VCD*) b is defined as a transformation of G that involves the following set of operations: (1) replacing v by v_b, where firing v_b is equivalent to b consecutive firings of v; (2) replacing each edge $e_i \in in(v)$ by an edge e'_i such that $cns(e'_i) = b \times cns(e_i)$; and (3) replacing each edge $e_o \in out(v)$ by an edge e'_o such that $prd(e'_o) = b \times prd(e_o)$. We refer to the actor v_b as the *b-vectorized actor* of v, and the *transformed graph* that results from the vectorization operation as $vect(G, v, b)$.

If G is a consistent, acyclic SDF graph, then $vect(G, v, b)$ is also consistent for any $v \in V$, and any positive integer b. However, in this work, we restrict the set of allowable vectorization degrees to the set $allowable(G, v)$, which is defined as

$$allowable(G, v) = \{n \in \{1, 2, \ldots\} \mid (n \text{ isafactorof } q(v)\}) \text{ or } (n \text{ isamultipleof } q(v)\}). \tag{4.1}$$

If G is understood from context, then we may apply a minor abuse of notation to write $allowable(v)$ in place of $allowable(G, v)$. This restriction enables fast derivation of repetition counts for $G' = vect(G, v, b)$ to facilitate incremental vectorization techniques, where actors are selected for vectorization one at a time according to specific greedy criteria. In particular, if b is a factor of $q(v)$, then $q(G', v) = q(G, v)/b$, while the repetition counts of all other actors are unchanged. Similarly, if b is a multiple of $q(v)$, then $q(G', v) = 1$, while for any other actor $u \neq v$, $q(G'u) = bq(G, u)/q(G, v)$. In Section 4.5, we discuss specific techniques for incremental vectorization that apply these forms of repetition count updates.

On HCGPs, vectorized actors can take advantage of SIMD processors such as GPUs to execute multiple firings of the same actor in parallel. However, in the presence of memory constraints, there are limits to the amount of vectorization that can be applied.

Figure 4.1 shows an example of how vectorization can increase the minimum buffer requirement. By the *minimum buffer requirement* for an SDF graph, we mean the minimum over all periodic schedules of the amount of memory (in units of tokens) required to implement the dataflow edges in a given graph, assuming that separate storage is dedicated for each edge. The figure shows an SDF graph, alternative vectorization configurations for the graph, and the corresponding minimum buffer requirements. The annotation in angle brackets above each actor gives the repetition count associated with the actor.

A lower bound on the minimum buffer requirement for a delayless SDF edge e can be determined by the following equation 4.2 (see [9]):

$$mbr(e) = prd(e) + cns(e) - gcd(prd(e), cns(e)), \tag{4.2}$$

where gcd represents the greatest common divisor operator.

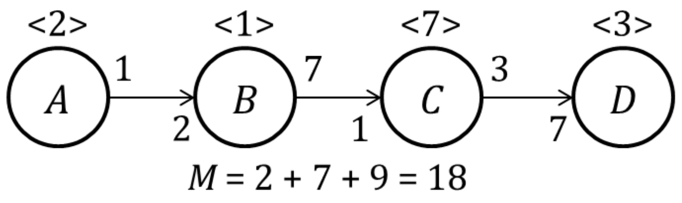
In the example SDF graph, it can be shown that the minimum buffer requirement is equal to the sum (over all edges) of the bound given by Equation 4.2. As we can seen in Figure 4.1(b)(c)(d), different ways of vectorizing actors in the SDF graph result in different increases to the minimum buffer requirement.

To represent SDF graphs with vectorized actors and their relationships with the original graphs, we define *vectorized SDF graphs* (*VSDFs*) as follows.

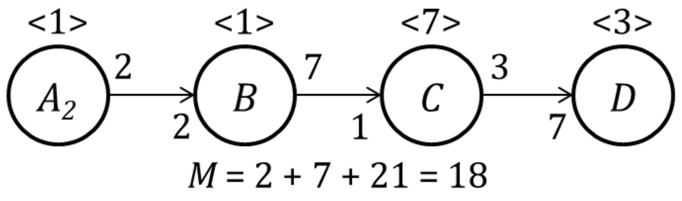
Definition 1. *Suppose that* $G = (V, E)$ *is a consistent SDF graph,* $b_v \in allowable(v, G)$ *is a VCD for each* v*, and* $B = \{(v, b_v) \mid v \in V\}$*. Then the* B-vectorized SDF graph of G *is defined as* $G_B = (V_B, E_B)$*, where (1) each* $v_B \in V_B$ *is the* b_v*-vectorized actor of* v*, (2) each edge* $e_B = (x_B, y_B)$ *in* G_B *is derived from the corresponding edge* $(x, y) \in E$*, and (3) for each* $e_B = (u_B, v_B) \in E_B$*,* $prd(e_B) = b_u \times prd(e)$*, and* $cns(e_B) = b_v \times cns(e)$*, where* $e = (u, v)$*.*

The vectorized graph G_B is an SDF graph. We defined a restricted form of vectorization, called *graph-level vectorization (GLV)*, in which a common "repetitions vector multiplier" $\beta \in \{1, 2, \ldots\}$ is used for all actors in the input graph. That is, $b_v = \beta \times q(G, v)$ for all $v \in V$. In this context, we refer to β as the *graph vectorization degree* (*GVD*). Under GLV , G_B is a single-rate SDF graph. To distinguish with GLV, we refer to the more general form of vectorization, where actors can have different VCDs, as *actor-level vectorization* (*ALV*).

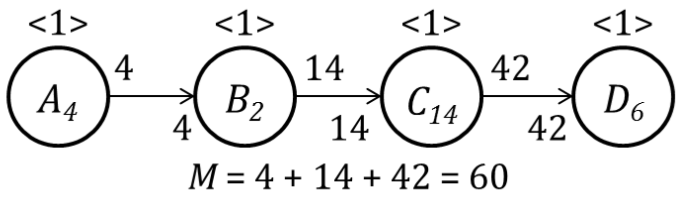
As discussed in Section 4.2, the conventional approach to solving VSTO involves 3 steps: (1) the designer or design tool sets the GVD based on memory constraints, (2) converts the SDF graph into a single-rate SDF graph using GLV, and (3) generates a schedule using task graph scheduling methods. Compared to



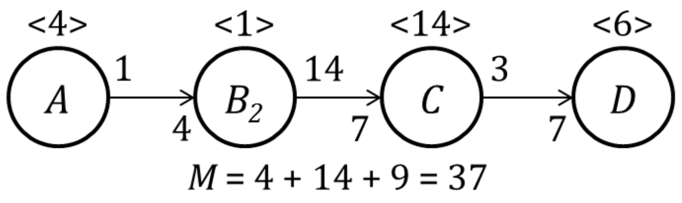
(a)



(b)



(c)



(d)

Figure 4.1: Example of vectorization and minimum buffer requirements. (a) Original graph. (b) Vectorization of A by 2. (c) Vectorization of all actors by 2. (d) Vectorization of B by 2.

ALV, GLV can require significantly larger buffers (see Figure 4.1(c)). The vectorization methods that we present in this chapter go beyond these conventional approaches by considering general ALV solutions instead of being restricted only to GLV solutions.

In addition to involving a larger design space, ALV is is more challenging compared to GLV because the transformed graphs are not single-rate graphs, and thus, conventional task-graph scheduling methods are not applicable at the back-end of the vectorization process. For multiprocessor scheduling of ALV solutions, we introduce in this work a general scheduling strategy, which is suitable for HCGPs, and can loosely be viewed as a variant of the list scheduling strategy. This variant is adapted for memory-constrained, multiprocessor mapping of transformed graphs that result from ALV. This strategy is a static scheduling strategy that operates using compile-time estimates of actor execution times. The general strategy is defined as follows.

Definition 2. *Given a consistent SDF graph $G = (V, E)$, and a multiprocessor target architecture with a set of processors P, the* Σ-scheduling strategy *(1) statically assigns each actor $v \in V$ to a processor $p \in P$, (2) statically determines a buffer bound $buf(e)$ for each edge $e \in E$, and (3) iteratively selects a bounded-buffer firable actor to fire on its assigned processor p as soon as p has completed all executions. An algorithm that conforms to this scheduling strategy completes when all actors in G have been scheduled using the iterative process of Step (3).*

The Σ-scheduling strategy is closely related to the Ω-scheduling strategy, which was introduced in [47]. Both the Σ and Ω strategies satisfy Parts (1) and (2) of Definition 2; the main difference is that in Part (3), Σ schedules actors onto a finite number of processors, while Ω-scheduling assumes an unlimited number of processors. Additionally, in our application of Σ-scheduling, we perform ALV to construct the input graph to the strategy. In contrast, Ω-scheduling in [47] is applied to the original (unvectorized) SDF graph.

To determine the buffer bounds $\{buf(e)\}$ in Σ-scheduling, we apply the Ω-buffering technique defined in [47]. This technique derives the buffer bounds by applying Ω-scheduling, and determining the buffer bounds to be equal to the corresponding buffer sizes $\{buf(e)\}$ that result from Ω-scheduling. We refer to the buffer bound $buf(e)$ for each edge e that is computed in this way as the Ω *buffer bound for* e. It is shown that Ω-buffering sustains maximum throughput for SDF graphs under Ω scheduling [47] so that imposing these bounds imposes no theoretical limitation on throughput. Given an SDF graph $G = (V, E)$, we denote by $\Omega_{buf}(G)$ the total buffer memory cost for G as determined by Ω-scheduling: $\Omega_{buf}(G) = sum_{e \in E}(buf(e))$.

Definition 3. *Suppose that $G = (V, E)$ is a consistent SDF graph, $b_v \in allowable(v, G)$ is a VCD for each v, $B = \{(v, b_v) \mid v \in V\}$, S_B is a periodic schedule for the B-vectorized graph G_B, and $T(S_B)$ is an estimate of the time required to execute a single iteration of S_B. Then from the fundamental properties of periodic SDF schedules [16], we can derive a unique positive integer $J(S_B, G)$, which we call the* blocking factor *of S_B relative to G, such that S_B executes each $v \in V$ exactly $(J(S_B, G) \times q(G, v))$ times. In this context, we define the* relative throughput of S_B

or the throughput of S_B relative to G *by the quotient* $J(S_B, G)/T(S_B)$. *This metric gives the average number of iterations of the original (unvectorized) SDF graph is executed per unit time by the schedule* S_B.

Intuitively, vectorization improves relative throughput when $T(S_B) < J(S_B) \times T(S)$, where S is the best available minimal-periodic (unvectorized) schedule for S. Such efficiency in the vectorized execution time $T(S_B)$ can be achieved due to improved utilization of processing resources under carefully-optimized GLV and ALV configurations. For example, if a vectorized actor v_b is mapped onto a GPU, it may be possible to process up to b firings of v in parallel to gain significant speed improvement across blocks of b firings. The techniques in this chapter are designed to systematically exploit this kind of execution efficiency under given memory constraints.

A limitation of the techniques developed in this chapter is that they may increase latency, and thus, they may not be suitable for implementations in which latency is a critical performance metric. However, it is envisioned that the methods developed in this chapter provide a useful foundation that can be built upon for latency-aware vectorization. Investigating adaptations of the these methods to take latency constraints into account is an interesting direction for future work.

Based on the definitions introduced in this section, we formulate the VSTO problem as follows.

Definition 4. *Let* $G = (V, E)$ *be a consistent SDF graph,* $P = \{p_1, p_2, \ldots, p_N\}$ *be the set of processors in an HCGP, where* $p_1, p_2, \ldots, p_{N-1}$ *represent the CPU cores,* *and* p_N *represents the GPU. Given a total memory budget* M *(a positive integer), the* vectorization-scheduling throughput optimization problem*, or* VSTO problem *associated with* G *and* P *is the problem of finding a set* B *of vectorization degrees, and a schedule* S_B *for* $G_B = (V_B, E_B)$ *such that the throughput of* S_B *relative to* G *is maximized subject to* $\Omega_{buf}(G_B) \leq M$*.*

We refer to a set of ordered pairs $C = \{(v, c_v) \mid (v \in V) \text{ and } c_v \in \mathit{allowable}(G, v)\}$ as an *ALV configuration* for G. Note that if an actor is not represented within a given ALV configuration (i.e., it does not appear as the first element of any ordered pair in the set), then the actor is assumed to be unvectorized (equivalent to a vectorization degree of 1). Thus, the VSTO problem can be thought of as the problem of jointly determining an ALV configuration B together with a schedule for G_B such that the resulting schedule optimizes throughput subject to a given buffer memory constraint M.

The vectorization formulation and techniques developed in this chapter assume that each SDF edge (FIFO buffer) is implemented in a separate block of memory. Various techniques have been developed in recent years to share memory efficiently among edges in multirate SDF graphs (e.g., see [50, 51]). Extending the techniques in this chapter to incorporate such memory sharing techniques is a useful direction for future work.

4.5 Vectorization and Scheduling with Memory Constraints

As discussed in Section 4.4, vectorization and scheduling are interdependent factors in throughput optimization of SDF graphs on HCGPs. The joint consideration of these factors is highly complex given the high complexity of multiprocessor scheduling in and of itself, along with the large number of alternative ALV configurations that can exist for dataflow graphs that contain significant numbers of actors. In this section, we develop in detail a set of efficient heuristics for integrated vectorization and scheduling that collectively address the VSTO problem for acyclic SDF graphs.

Specifically, in this section we develop three main heuristics called Incremental Actor Vectorization (IAV), N-candidates IAV, and Mapping-Based Devectorization. These three heuristics can be viewed as "peers" in the sense that any one of them may be the preferable choice for a given application. Thus, the designer or a design tool can apply all three of these complementary methods and select the best result for a given application. This is how we have integrated the three heuristics in our DIF-GPU software framework. More details on the integration with software synthesis and associated experimental results are discussed in Section 4.6 and Section 4.7.

4.5.1 Incremental Actor Vectorization

In this section, we define a general approach for searching the space of ALV configurations that is based on selecting and vectorizing actors one at a time based on some specific greedy criteria. We refer to this general approach as *Incremental*

Actor Vectorization (*IAV*). Each iteration of IAV, called an *IAV iteration*, involves the selection and vectorization of a single actor. This results in a sequence of intermediate vectorized graphs, $I_1, I_2, \ldots, I_N$, where I_i is the transformed graph that results from IAV iteration i, and N is the total number of iterations before IAV terminates. The approach is incremental in both the dimensions of actors and vectorization degrees — that is, each IAV iteration selects a single actor v, and increases its vectorization degree to the next highest element of $allowable(G, v)$. Given an actor v that has an associated vectorization degree b_v, we refer to this process of replacing b_v with the next highest element $min(x \in allowable(v) \mid x > b_v)$ as *stepping up* the vectorization of v or just "stepping up v".

In IAV, we define a "score" function to guide the vectorization process. At each algorithm iteration, IAV selects an actor that has the highest score among all actors whose stepping up would not result in a violation of the given memory budget M. Analogous to how different priority functions can be used to select tasks in multiprocessor list scheduling (e.g., see [37]), different score functions can be used to apply different ALV criteria in IAV. This contributes to a novel design space for development of integrated vectorization and scheduling techniques.

The specific score functions that we experiment with in this work first apply Σ-scheduling to generate a schedule $\mu(i)$ of the current I_i (intermediate vectorized graph) onto the target HCGP P, and then use a specific metric to estimate the potential "gain" of each candidate stepping up operation relative to the processor assignment associated with $\mu(i)$. Given a schedule S returned by Σ-scheduling, we define the associated *processor assignment* associated with S and dataflow graph

$G = (V, E)$ as the function $mp_S : V \to P$ such that for each $v \in V$, $mp_S(v)$ gives the processor to which actor v is mapped according to S. The initial schedule $\mu(0)$ is derived by applying Σ-scheduling to the input (unvectorized) graph for IAV.

Algorithm 1 shows a pseudocode description of the IAV approach that employs this mapping-based method of score function formulation. In the remainder of this chapter, we refer to the mapping-based form of IAV shown in Algorithm 1 as "Σ-IAV".

Algorithm 1 Integrated Vectorization and Mapping using Σ-IAV.

```
Function incrementalVectorize($G, P, M$)
    initialize $configs = \emptyset$, $G_B = G$, $B = \{(v, 1) | v \in V\}$
    while memSize($G_B$) $\leq M$ do
        $mp$ = generateMapping($G_B, P$)
        $v^* = \text{argmax}_{v \in V}$ score($B, mp, v$)
        $B(v^*)$ = nextVCD($v$)
        $G_B$ = vectorize($G, B$)
        if memSize($G_B$) $\leq M$ then
            $configs = configs \cup \{(B, mp)\}$
        end
    end
    return $\text{argmax}_{c \in configs}$ throughput($G$, $c$)
```

In Algorithm 1, `generateMapping` is a placeholder for any Σ-scheduling technique that is applied to map a given intermediate vectorized graph onto the targeted heterogeneous platform P. In our implementation of Σ-IAV, we employ a specific Σ-scheduling technique called *Incremental Actor Re-assignment (IAR)* as the `generateMapping` function. The IAR technique is discussed further in Section 4.

The function `throughput` referenced in Algorithm 1 represents a placeholder for any function that is used to estimate the throughput of a mapping that is generated by `generateMapping` for an intermediate vectorized graph. In our implementation of Σ-IAV, we employ an efficient simulation-based approach for this kind of throughput estimation. This simulation approach is discussed further in Section 4.5.5.

We formulate and experiment with two specific score functions in this work. We refer to these score functions as *time-saving* (*TMSV*) and *time-saving-per-byte* (*TMSVPB*). The TMSV score for actor v during IAV iteration i is defined as largest *adjusted execution time reduction* achievable (across all processors in P) when stepping up v. This "adjusted" time reduction is computed relative to the execution of v on $mp_{\mu(i)}(v)$, and is normalized by the vectorization degree. The units of this adjusted time reduction are thus "seconds per unit of vectorization". This score can be expressed as:

$$tmsv(v, i) = \max_{p \in P}\left(\frac{t(v, b, mp_{\mu(i)}(v))}{b} - \frac{t(v, b', p)}{b'}\right), \tag{4.3}$$

where t represents a function that is used to retrieve statically-derived ALV profiling data, b is the current VCD of v (in IAV iteration i), and $b' in allowable(v)$ is the VCD that would result from stepping up v. The function t provides access to profiling data that is collected for a finite subset $profiled(v) \subset allowable(v)$ of contiguous elements in $allowable(v)$ starting at 1. For a given actor v, vectorization degree $b \in profiled(v)$, and processor $p \in P$, $t(v, b, p)$ gives the profiling-derived estimate for the execution time of v on p with vectorization degree b.

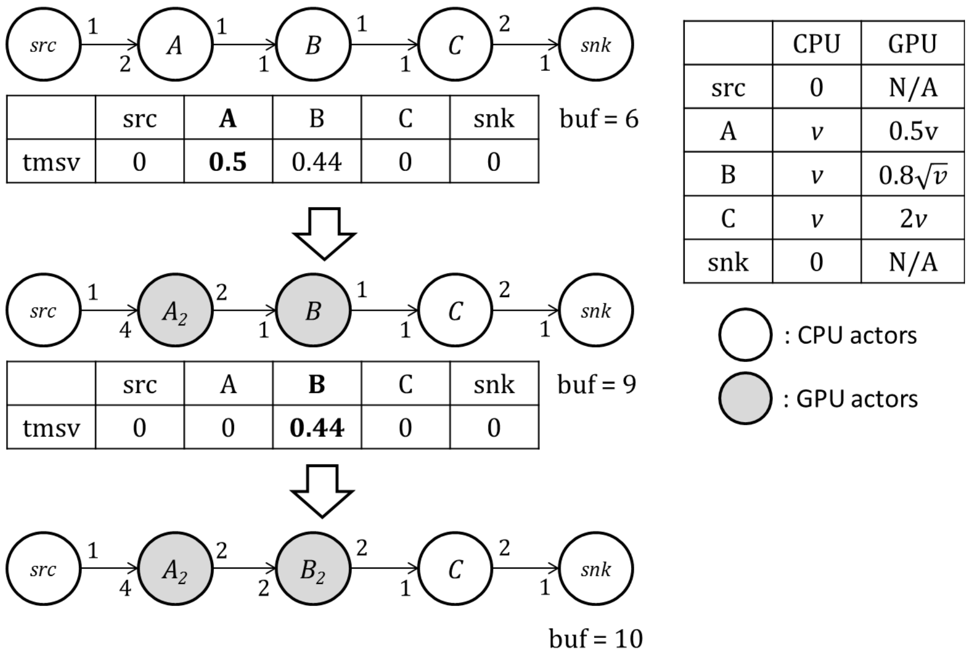
Here, we use "profiling" as a general term that encompasses any method for deriving a compile-time estimate for the execution time of a vectorized actor execution. The specific approach to profiling that we use in our experiments is discussed in Section 4.6.

Intuitively, a lower incremental time required to execute additional firings leads to a higher TMSV score. An extreme (theoretical) case is when an unlimited number of firings of a given actor can be executed in unit time on any available processor type — in this case, $f_s(v, i) = 1/b' - 1/b$. TMSV also favors actors that have lower vectorization degrees since $f_s(v, i) \to 0$ as $b \to \infty$.

Figure 4.2 shows a simple example of vectorization using the TMSV score function. The table in this figure provides analytical models, in terms of the vectorization degree v, that are used to derive the profiling function t. For example, the models estimate that actor A requires approximately $(0.5 \times v)$ units of time to execute.

The IAV process begins with an unvectorized graph and an initial mapping where all actors are mapped to the CPU core. In the first IAV iteration ($i = 0$) shown in Figure 4.2, A has the largest TMSV score, so it is selected, and a new mapping is generated based on the VCDs. In the second iteration, B has the largest TMSV score, so B is vectorized (stepped up), and the mapping is updated again. This process continues until no more vectorization operations can be carried out without exceeding the memory budget M.

Under memory constraints, we expect that it will be more useful to consider the increase in buffer requirements when selecting actors for ALV. This motivates our formulation of the TMSVPB score function. This memory-aware score function can be formulated as:



	src	**A**	B	C	snk
tmsv	0	**0.5**	0.44	0	0

	src	A	**B**	C	snk
tmsv	0	0	**0.44**	0	0

	CPU	GPU
src	0	N/A
A	v	0.5v
B	v	$0.8\sqrt{v}$
C	v	$2v$
snk	0	N/A

Figure 4.2: A simple example to illustrate Σ-IAV using the TMSV score function.

$$tmsvpb(v, i) = \max_{p \in P}\left(\frac{t(v, b, mp_{\mu(i)}(v))/b - t(v, b', p)/b'}{\Omega_{buf}(G_{B'}) - \Omega_{buf}(G_{B(i)})}\right), \tag{4.4}$$

where $B(i)$ represents the current ALV configuration in ILV iteration i, and $B' = B(i) - \{(v, b)\} + \{(v, b')\}$ represents the candidate configuration that results from stepping up v. Thus, the TMSVPB function favors actors whose vectorization results in throughput improvement without excessive increase in buffer requirements.

4.5.2 N-Candidates IAV

Our proposed Σ-IAV approach has two drawbacks — (1) it selects only one actor at each step, and (2) with the TMSV and TMSVPB score functions, the selections are based on actor execution times only, without taking into account the SDF graph topology. We alleviate the first drawback by storing multiple vectorized-graph candidates to consider in each IAV iteration following the very first iteration. In particular, we store N candidate graphs that provide the highest throughput when processed by by Σ-scheduling. Here, N is a parameter that can be controlled by the designer or tool developer.

The second drawback can be addressed by applying Σ-scheduling to optimize throughput over each actor for every candidate graph. That is, for each candidate graph Y that is stored, and each actor v, we apply Σ-scheduling to the transformed graph that results from stepping up v in Y. We then take the best result from all of these Σ-scheduling-based evaluations to determine the vectorization operation that is to be applied in the associated IAV iteration. This approach results in some increase in complexity, but has the potential to perform significantly more thorough optimization at a relatively high level of design abstraction.

We refer to this modified Σ-IAV approach as *N-candidates IAV*. Algorithm 2 provides a pseudocode description of N-Candidates IAV. Here, the notation $c.1$ denotes the first element of the ordered pair c, and $\mathit{configs}[1 : N]$ denotes the list that consists of the first N elements of the list *configs*.

As with our implementation of Σ-IAV, we employ in our implementation of N-candidates IAV the IAR technique (Section 4) as the `generateMapping` function.

Similarly, our implementation of N-candidates IAV incorporates the simulation-based throughput estimation technique that is discussed in Section 4.5.5. This estimation technique corresponds to the function called `throughput` in Algorithm 2.

Intuitively, N-candidates IAV is a greedy method that tries to avoid unsatisfactory search paths by retaining multiple intermediate vectorized graphs during each

Algorithm 2 A pseudocode description of N-candidates IAV.

```
Function nCandidatesVectorize($G = (V, E), P, M, N$)
    initialize $B = \{(v, 1) | v \in V\}$,
    $mp$ = generateMapping($G, P$), $configs = \{(B, mp)\}$, $flag = true$
    while $flag = true$ do
        $flag = false$
        foreach $c \in configs$ do
            foreach $v \in V$ do
                $B' = c.1 - \{(v, b_v)\} \cup \{(v,$ nextVCD$(v, b_v)\}$
                if (visited$(B') = true) and (\Omega_{buf}(G_{B'}) \leq M)$ then
                    $mp$ = generateMapping($G_{B'}, P$)
                    $configs = configs \cup \{(B', mp)\}$
                    $flag = true$
                end
            end
        end
        sortByThroughput ($configs$)
        $configs = configs[1 : N]$
    end
    return $\text{argmax}_{c \in configs}$ throughput($G$, $c$)
```

IAV iteration. Larger values for the parameter N allow more extensive design space exploration at the cost of greater running time. When $N = 1$, N-candidates IAV reduces to IAV with the score function being the estimated throughput ("throughput") of the transformed graph that results from the selected vectorization operation. In our implementation of N-candidates IAV, we estimate throughput using simulation. This simulation approach is discussed further in Section 4.5.5. In Algorithm 2, `throughput`(G, c) represents the estimate of throughput that is derived in this way for a given intermediate vectorized graph G that is based on ALV configuration c.

Other score functions can be used in N-candidates IAV other than throughput. However, in our experiments, we found that among TMSV, TMSVPB, and throughput, the throughput score function produces the best results. Investigation of other score functions in this context is an interesting direction for future work.

In our experiments, we used $N = |V|$ as the number of candidates to be stored.

4.5.3 Mapping-Based Devectorization

N-candidates IAV is an incremental vectorization method that starts with an unvectorized graph, and gradually increases the VCDs of selected actors. In some cases, it may be advantageous to also consider decreasing VCDs during the optimization process. Such decreasing of VCDs can be useful to reduce memory consumption associated with selected actors so that memory can be dedicated to groups of other actors that provide greater throughput benefit through vectorization. A specific form of decrease that we consider in this section is *devectorization*, where an actor with VCD $b > 1$ is transformed to have no vectorization (VCD of unity).

Figure 4.3(a) shows an example of this kind of scenario. Here, S (source), K (sink), F (fork), and C (combine) are computationally simple actors without potential for GPU acceleration, and only very limited potential for speedup through CPU-based vectorization. On the other hand, actors A_1, A_2, A_3, A_4 have GPU-accelerated versions with significant throughput gain. In this case, however, the overall throughput gain is limited by the slowest of the four A_is so that incrementally vectorizing individual A_is does not directly impact throughput gain.

To provide memory efficient vectorization in which this kind of scenario is of dominant concern, we propose another vectorization method called *Mapping-Based Devectorization* (*MBD*). In contrast with ALV-based incremental vectorization, MBD applies GLV to first vectorize all vectorizable actors, and then performs

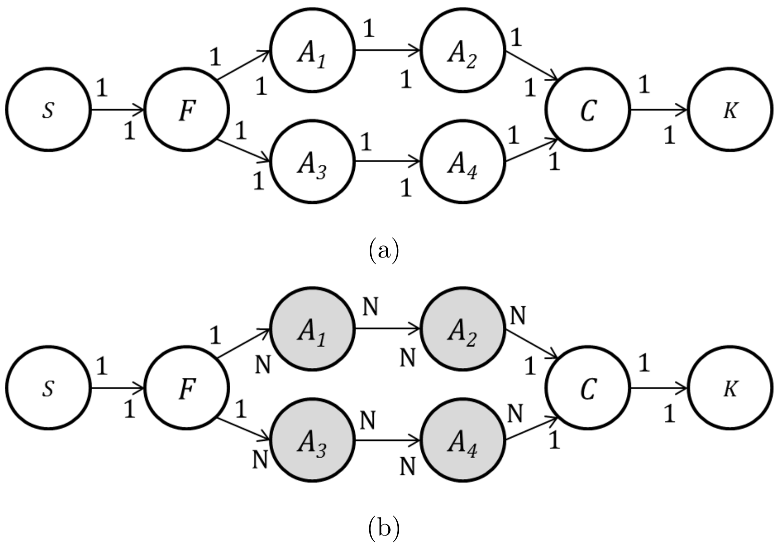


Figure 4.3: An example that illustrates the utility of devectorization. (a) The original graph. (b) The graph with GVD = N and devectorization applied to all CPU-mapped actors — C, F, K, S.

devectorization on the transformed graph derived from GLV. MBD is useful in devectorizing actors that have have relatively low CPU-based performance gain through vectorization, and in jointly considering vectorization improvements produced by groups of actors.

MBD performs GLV, generates a processor assignment A, and then evaluates for devectorization each actor that is mapped to a CPU core in A. If a given devectorization operation does not reduce the original throughput by a pre-defined threshold r, the actor is devectorized. In our experiments, we set the threshold r empirically by experimenting with different values of r. We found in our experiments that $r = 0.95$ achieves the maximum throughput gain for MBD (see Section 4.6).

In principle, the processor assignment A can be generated using any multiprocessor task graph scheduling technique. In our implementation of MBD, we employ the Heterogeneous Earliest Finish Time (HEFT) heuristic (e.g., see [49, 39]) to generate a schedule for the transformed graph that results from GLV, and then we extract the processor assignment from this generated schedule.

Devectorization saves memory from low-impact vectorization of actors that are mapped onto CPU cores. When memory constraints are loose enough to allow GLV, the MBD technique, based on the memory savings achieved through devectorization, may improve throughput by allowing greater GVDs to be applied.

Figure 4.3(b) illustrates the application of MBD. In this example, since actors C, F, K, and S are mapped onto CPU cores, they are devectorized. As a result of this devectorization, the buffer requirements on edges (S, F) and (C, K) are reduced to 1 for each edge.

Algorithm 3 provides a pseudocode description for MBD.

Algorithm 3 Mapping-Based Devectorization (MBD).

```
Function mappingBasedDevectorize($G = (V, E), P, M$)
    initialize $B = \{(v, 1) | v \in V\}$, $mp$ = generateMapping($G, P$),
    $configs = \{(B, mp)\}$, $G_B = G$, $gvd = 1$
    repeat
        $B' = B$, $mp' = mp$
        $B$ = graphVectDegrees($G, gvd$)
        $G_B$ = vectorize($G, B$)
        $mp$ = generateMapping($G_B$)
        $cpu\_actors = \{v \in V | v$ ismappedtoaCPUcore$\}$
        foreach $v \in cpu\_actors$ do
            $B'' = B - \{(v, b)\} \cup \{(v, 1)\}$
            if throughput($G, (B'', mp)$) $\geq r \times$ throughput($G, (B, mp)$) then
                $B = B''$
            end
        end
        $gvd = gvd + 1$
    until memSize($G_B$) $\leq M$;
    return $(B', mp')$
```

4.5.4 Mapping Actors onto HCGPs

The Σ-IAV and N-candidates IAV methods presented in Section 4.5.1 and Section 4.5.2, respectively, both employ Σ-scheduling throughout the optimization process to generate schedules for intermediate vectorized graphs. The Σ-scheduling approach is useful in our iterative optimization context because it provides moderate-complexity, bounded-buffer scheduling of multirate SDF graphs. As mentioned in Section 4.5.1 and Section 4.5.2, we develop a specific Σ-scheduling technique called *Incremental Actor Re-assignment* (*IAR*) for use in both Σ-IAV and N-candidates IAV. In this section, we elaborate on the IAR technique.

In contrast to time-intensive scheduling methods such as Mixed Linear Programming and Genetic Algorithms, IAR is designed with computational efficiency as a primary objective. This is because IAR is invoked repeatedly during each IAV iteration — in particular, it is invoked for each candidate ALV configuration.

Intuitively, IAR incrementally moves actors in Σ schedules from "busier" (more loaded) processors to less busy ones. Algorithm 4 provides a pseudocode description of the IAR method. IAR initializes the actor assignment by mapping all actors that have GPU-accelerated versions onto the GPU, and all other actors onto a single CPU core. This results in an initial assignment that utilizes at most two processors (the GPU and one CPU core).

Although the MBD algorithm begins by applying GLV, the algorithm produces solutions that are in general ALV solutions. This is because of the application of devectorization later in the algorithm, which in general results in heterogeneous

Algorithm 4 Incremental Actor Re-assignment (IAR).

```
Function generateMapping($G, P$)
    for $v \in V$, initialize $bestMp(v) = p_N$ if
    $t(v, p_N) < \infty$ and $bestMp(v) = p_1$ otherwise
    initialize $bestTh = \texttt{throughput}(G, bestMp)$
    foreach $v \in V$ do
        $mp = bestMp$, $th = bestTh$, $p^* = bestMp(v)$
        $Q = \{q \in P | q \neq p^*\}$
        foreach $p \in Q$ do
            $mp' = mp - \{(v, mp(v))\} \cup \{(v, p)\}$
            $th' = \texttt{throughput}(G, mp')$
            if $th' > th$ then $mp = mp'$, $th = th'$ ;
        end
        if $th > bestTh$ then $bestMp = mp$,$bestTh = th$ ;
    end
    return ($bestMp, bestTh$)
```

vectorization degrees across the set of actors in the input graph.

Then IAR iteratively computes the maximum throughput gain for all actor-processor pairs, and selects the pair that gives the highest throughput at each iteration. In this context, selection of an actor-processor pair (a, p) means that the current processor assignment of actor a will be discarded, and actor a will be assigned ("moved") to processor p. For this selection process, only actors that have not yet been selected during previous iterations are considered. The throughput gain is computed with the aid of the function denoted in Algorithm 4 as `throughput`. This function invokes the simulation-based throughput estimator discussed in Section 4.5.5. Each actor is moved only once during execution of IAR.

4.5.5 Throughput Estimation

For compile-time throughput estimation, we have developed a throughput simulator for SDF graphs that follows bounded-buffer execution semantics (defined in

Section 4.3) with a statically-determined processor assignment, as derived by the Σ strategy introduced in Section 4.4. The inputs to the simulator are: (1) the transformed SDF graph G_v that results from the candidate set of vectorization operations that is under evaluation; (2) the Σ mapping for G_v that is generated by IAR; (3) the Ω buffer bound for each edge in G_v; (4) an estimate of the execution time for each actor in G_v; and (5) an estimate of the data transfer time between the main memory and the device memory on the target platform. In our experiments, the execution time estimates under different vectorization degrees for each actor as well as the data transfer time are derived by using measurements of actor and data transfer execution on the target HCGP.

As described above, this simulator applies SDF bounded-buffer execution semantics. When an actor a is bounded-buffer fireable, its assigned processor p is idle, and no other actors assigned to p are bounded-buffer fireable, actor a is fired on p. When multiple actors are bounded-buffer fireable on an idle processor, we select the actor with the earliest finish time to fire. The throughput simulator naturally incorporates pipelined parallelism, as firings of an actor can be executed whenever they satisfy the bounded-buffer firing condition, without the need to wait until actor firings from previous graph iterations are completed .

4.5.6 Summary

Figure 4.4 summarizes the developments of this section by illustrating relationships among the key analysis and optimization techniques that have been in-

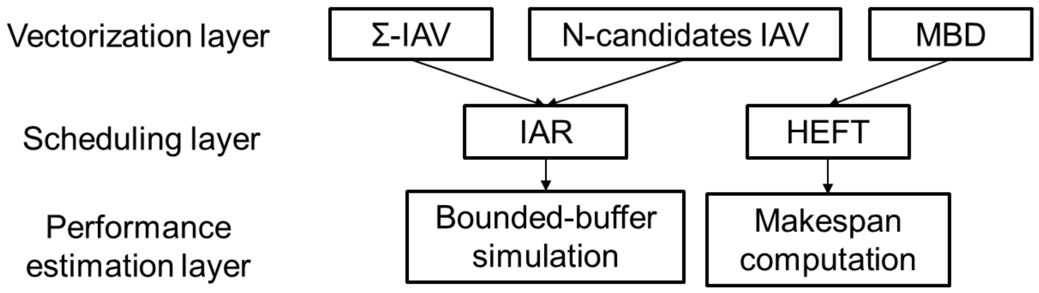


Figure 4.4: Layered structure of vectorization, scheduling, and performance estimation in the proposed design optimization framework.

troduced. Recall that IAV, HEFT, and MBD stand, respectively for incremental actor vectorization, heterogeneous earliest finish time, and mapping-based devectorization. Each directed edge in Figure 4.4 represents usage of one technique (at the sink of the edge) by another (at the source of the edge). For example, IAR is used by Σ-IAV.

In the remainder of the chapter, we develop an experimental evaluation of our proposed new design optimization framework for mapping SDF graphs onto heterogeneous, CPU/GPU platforms, and we study the contributions of the different components shown in Figure 4.4 to the overall effectiveness of the framework.

4.6 Experiments using Synthetic Graphs

In this section, we demonstrate the effectiveness of the models and methods developed in Section 4.5 through experiments that study throughput gain and running time. We compare our methods with the approach of applying graph-level vectorization (GLV) followed by task-graph scheduling.

We use Heterogeneous Earliest Finish Time (HEFT) as the task-graph scheduling method in this comparison. HEFT is a commonly used task-graph scheduling method for HCGPs (e.g., see [49]). The integration of HEFT with GLV can be viewed as a natural way to integrate SDF vectorization and scheduling using conventional techniques. We refer to the combination of GLV and HEFT as the *GLV-HEFT baseline* or simply as *GLV-HEFT*. As implied by this terminology, GLV-HEFT is employed in this experimental study as a baseline for evaluating our proposed methods. The GLV-HEFT baseline applies both GPU acceleration and CPU-GPU multi-processor scheduling. We demonstrate in this section that the ALV and IAR scheduling methods developed in this chapter provide significant throughput gain over this baseline approach under given memory constraints.

4.6.1 Experimental Setup

The vectorization and scheduling techniques that we have developed in this part of thesis have been integrated into DIF-GPU (see Chapter 3) to provide a streamlined workflow that combines actor-level / graph-level vectorization, multi-rate / single-rate SDF scheduling, code generation, and runtime support on heterogeneous computing platforms with multi-core CPUs and GPUs.

In the experiments presented in this section, we employ an HCGP consisting of a quad-core Intel i5-6400 CPU and an NVIDIA Geforce GTX750 GPU. Actor implementations that are developed for multi-core CPU and GPU execution are compiled using GCC 4.6.3 and the NVIDIA CUDA compiler (NVCC) 7.0, respectively.

4.6.2 Synthetic Graph Generation

We use Task Graphs For Free (TGFF) [52] to generate large sets of synthetic SDF graphs with varied size and complexity. From the graph topologies generated by TGFF, we randomly map each graph vertex to a specific DSP actor type that has both a CPU-targeted and GPU-targeted implementation. We perform this vertex-to-actor mapping for all actors in each randomly-generated graph. A broad set of DSP actor types — including actors for cross-correlation, FIR filtering, FFT computation, and vector algebra — are considered when performing this mapping. The GPU-accelerated implementations of these actors provide speedups from 1X to 20X compared to the corresponding multicore CPU implementations. This use of TGFF in conjunction with randomly generated actor mappings helps us to evaluate the performance of our proposed methods on a large variety of graph topologies.

In our experiments, the source and sink actors are selected from a pool of different implementations of data sources and sinks. Because the input/output interfacing functionality in an embedded HCGP is typically implemented on a CPU, we assume that source and sink actors can only be mapped onto CPU cores.

We profile the actors by measuring the execution times of the actors' firings on the target platform under a series of vectorization degrees. This profiled data is then used as input to the evaluated vectorization and scheduling techniques. The profiled data is also used to simulate the vectorization-integrated schedules that are derived from the proposed and baseline techniques. This simulation is based on the throughput simulator presented in Section 4.5.5. We use simulation here to enable efficient, automated comparisons across a large variety of different graph structures. In Section 4.7, we complement this simulation-based evaluation approach with our experimental evaluation of a case study involving an orthogonal frequency-division multiplexing (OFDM) receiver. The evaluation in Section 4.7 is performed by synthesizing software using DIF-GPU for the targeted HCGP platform, executing the synthesized software on the target platform, and measuring the resulting execution time performance.

4.6.3 IAR Scheduling

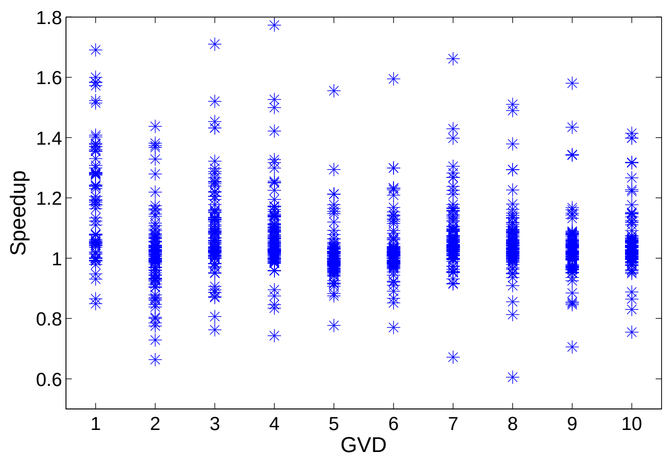
Our first experiment analyzes the impact of IAR scheduling on the targeted HCGP with various degrees of GLV. We first apply IAR scheduling on graph-level-vectorized SDF graphs with a series of GVDs, and compare the simulated throughput gain with HEFT scheduling under the same vectorization settings. We refer to these two methods as GLV-IAR and GLV-HEFT, respectively.

Figure 4.5 shows the speedup of GLV-IAR over GLV-HEFT using (a) 1 CPU core and 1 GPU, and (b) 3 CPU cores and 1 GPU. This comparison is performed based on simulated throughput results measured from 100 synthetic graphs that are generated using the methods described in Section 4.6.2, and vectorized using GVDs that range from 1 to 10.

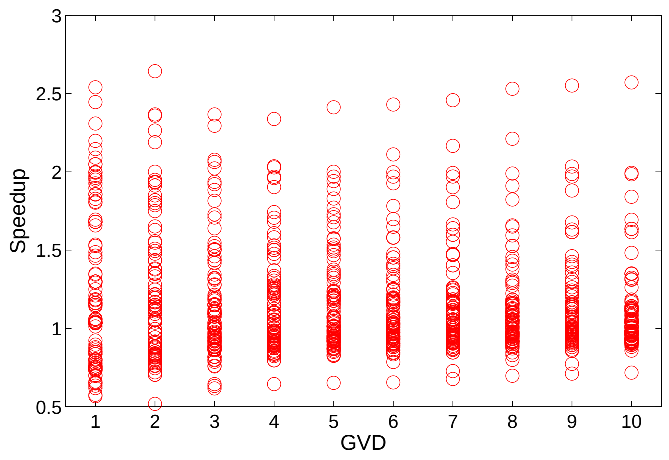
The results shown in Figure 4.5 indicate that GLV-IAR achieves a throughput improvement over GLV-HEFT on average, and that the gain can vary significantly between different input graphs. The results also suggest that the distribution of speedup is not strongly dependent on the GVD that is applied to the graph. While for some input graphs, GLV-IAR can achieve a speedup that significantly exceeds 1, there are other input graphs for which the speedup falls in the range of 0.5X-1.0X. The average speedups over GLV-HEFT are 1.06 and 1.15 for the 1 CPU core / 1 GPU target, and 3 CPU cores / 1 GPU target, respectively. The maximum speedups of GLV-IAR over GLV-HEFT measured in this experiment are 1.78 and 2.64, respectively, for these two target platform configurations.

From these results, we conclude that using IAR scheduling on SDF graphs that are graph-level-vectorized provides some average increase in throughput performance, although this average increase is not dramatic.

However, we emphasize here that the more important advantage of IAR compared to HEFT in our investigated design flow is not its potential for throughput gain, but rather its applicability to multirate SDF graphs. Recall from our discussion in Section 4.4 that HEFT applies to acyclic single-rate graphs (task graphs), while IAR is designed to operate directly on arbitrary acyclic SDF graphs, including both single-rate and multirate graphs. This applicability to multirate graphs enables its efficient integration with ALV, which in turn enables derivation of more memory-efficient vectorization solutions compared to GLV. In Section 4.6.4, we present an experimental study on the utility of IAR for ALV.



(a)



(b)

Figure 4.5: Simulated speedup compared between the vectorized schedules generated by GLV-IAR and GLV-HEFT. Two target platform configurations are considered in this evaluation: (a) 1 CPU core + 1 GPU, and (b) 3 CPU cores + 1 GPU.

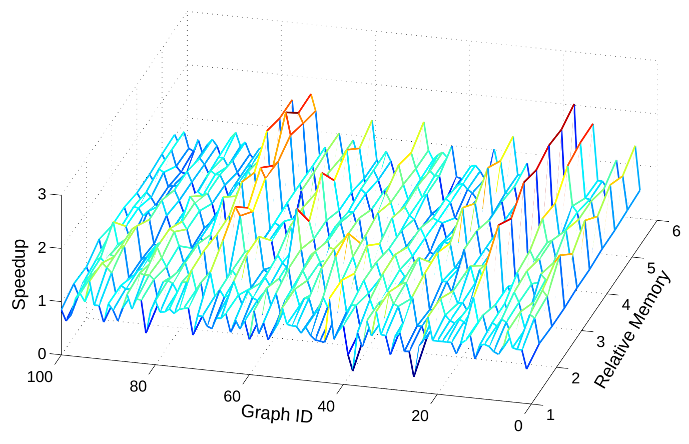
4.6.4 Vectorization

In this section, we apply the different ALV methods introduced in Section 4.5 to a large collection of synthetic SDF graphs, and evaluate the performance of the derived schedules by simulating their bounded-buffer execution. The synthetic graphs are generated using TGFF together with randomized vertex-to-actor mappings, as described in Section 4.6.3. We evaluate the speedup over the GLV-HEFT baseline under different memory constraints.

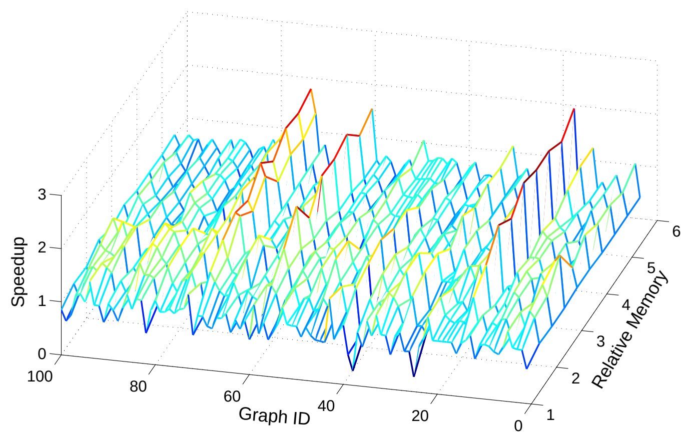
To compare speedups across SDF graphs that have different sizes (i.e., different numbers of actors and edges) and different multirate properties (as defined by the production and consumption rates on the actor ports), we introduce a concept of *relative memory bounds* as a normalized representation for memory constraints. Given an algorithm A for performing GLV, the relative memory bound $M(G)$ for an SDF graph G is defined as $M(G) = M_0 \times \alpha$, where M_0 is the memory cost of the GLV solution derived by Algorithm A when applied to G with GVD = 1, and α is a constant that represents the "tightness/looseness" of the applied memory constraint. In our experiments, we experiment with $\alpha \in \{1.0, 1.5, \ldots, 5, 5\}$ to cover a series of memory constraints ranging, respectively, from tight to loose.

Figure 4.6 shows the simulated speedup that we measured from a set of randomly generated SDF graphs for different techniques for ALV that were introduced in Section 4.5. These results are for a target platform configuration that consists of 1 CPU core and 1 GPU. Here, "TMSV Σ-IAV" and "TMSVPB Σ-IAV" represent the Σ-IAV algorithm with the TMSV and TMSVPB score functions, respectively.

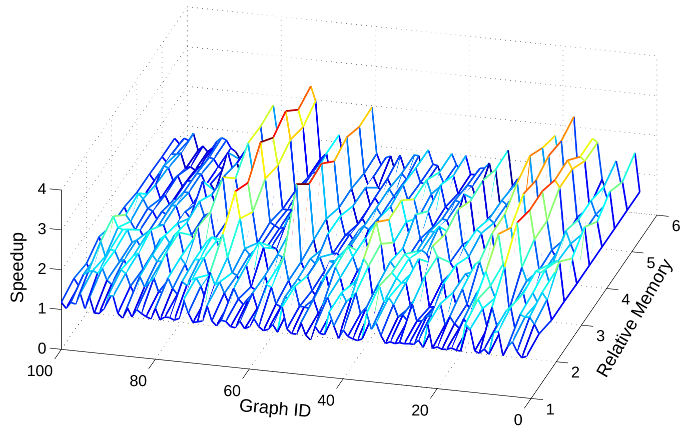
The results in Figure 4.6 show that each of the four ALV methods provides unevenly distributed throughput gains over the test set, where the measured throughput gain ranges from 0.8X to 2.4X. The speedup obtained by the ALV methods can also exhibit significant variation from one SDF graph to another. The Graph IDs along the horizontal axis in Figure 4.6 are arranged in random order.



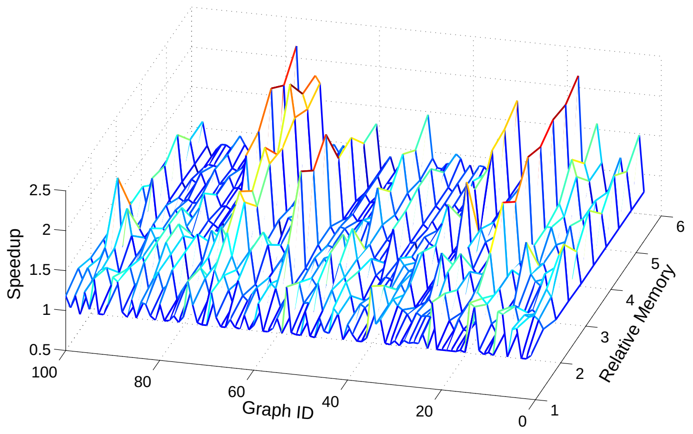
(a)



(b)



(c)



(d)

Figure 4.6: Speedups measured for four different ALV techniques that were introduced in Section 4.5: (a) TMSV Σ-IAV, (b) TMSVPB Σ-IAV, (c) N-candidate IAV, and (d) MBD. The target platform configuration consists of 1 CPU core and 1 GPU.

Table 4.1 shows the average and maximum speedups measured for the four ALV methods — TMSV Σ-IAV, TMSVPB Σ-IAV, N-candidates IAV, and MBD — that are represented in Figure 4.6. As mentioned previously, these speedups are in comparison to baseline solutions that are derived using the GLV-HEFT baseline technique.

The column labeled *ALV-IAR* in Table 4.1 represents the meta-algorithm that results from applying all four of the proposed ALV techniques, and selecting the best result from among the four derived solutions. In Section 4.7, we perform further experimental analysis of the ALV-IAR method, which provides a way to leverage complementary benefits of all of the key ALV techniques introduced in Section 4.5. ALV-IAR is useful, in particular, for design scenarios that can tolerate the relatively large optimization time that is required by N-candidates IAV, which dominates the time required by ALV-IAR.

From this table, we see that N-candidates IAV provides the largest average speedup by a significant margin, and this algorithm also provides the largest maximum speedup. We anticipate that this is because N-candidates IAV uses more vectorization candidate solutions throughout the search process. The other three ALV techniques achieve similar average and maximum throughput gain.

Although the MBD method and the two Σ-IAV methods achieve smaller average speedup compared to NIAV, they run significantly faster (see Section 4.6.5), and can be useful in cases where quicker turnaround time is desired from the software synthesis process. In addition, there are cases where they perform better than NIAV. Thus, when the quality of the derived solutions is of utmost importance, it useful to apply the ALV-IAR meta-algorithm described above. This ALV-IAR method demonstrates average and maximum speedup values of 1.36X and 2.9X on the benchmark set that we have experimented with.

Table 4.1: Average and maximum speedup results of the four investigated ALV techniques compared to the GLV-HEFT baseline technique.

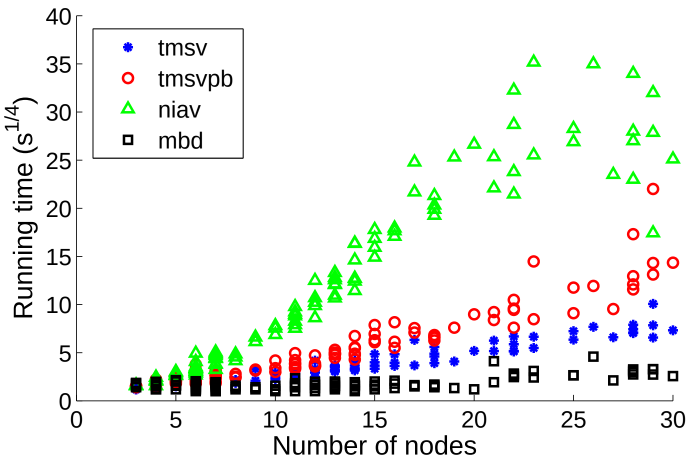
	TMSV	TMSVPB	NIAV	MBD	ALV-IAR
Average	1.17	1.16	1.33	1.16	1.36
Max	2.7	2.6	2.9	2.5	2.9

4.6.5 Runtime

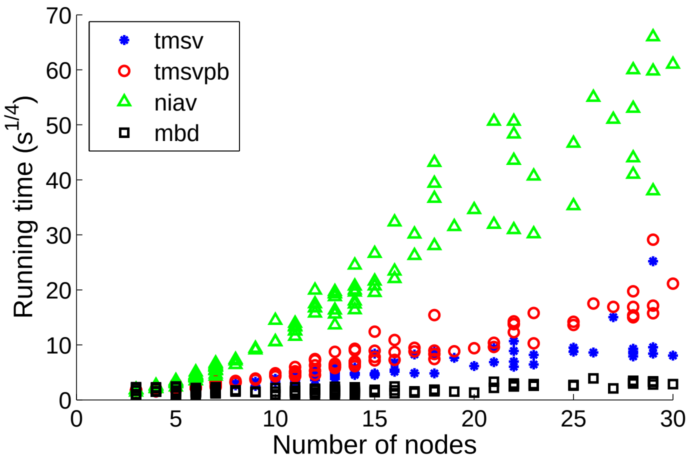
In this section, we compare the measured running times of the four proposed ALV techniques. We tested the running times of the ALV techniques on the same set S_g of randomly generated SDF graphs that we used in the experiments reported on in Section 4.6.3 and Section 4.6.4. The set S_g consists of 120 graphs, where the of number of nodes in a given graph ranges from 3 to 30.

Figure 4.7 shows the measured running times for the four ALV methods with respect to the number of nodes in the input graph. For each of the four ALV methods, there are 120 points plotted in each part of the figure — one point for each graph in S_g. Thus, Figure 4.7(a) and Figure 4.7(b) each depicts a total of $4 \times 120 = 480$ plotted points. From these points, we can observe trends in how the running time increases with the size of the input graph.

Figure 4.7 presents running time results associated with two different memory constraints — $M = 2M_0$ in Figure 4.7(a), and $M = 4M_0$ in Figure 4.7(b) (see the discussion on relative memory bounds in Section 4.6.4). These two memory constraints are used to represent relatively tight and loose memory budgets, respectively. The vertical axes in Figure 4.7 correspond to $s^{1/4}$, where s is the measured running time in seconds. Here, we apply an exponent of (1/4) to help improve clarity in depicting the large number of displayed points.



(a)



(b)

Figure 4.7: Runtime of ALV methods under different memory constraints: (a) $M = 2M_0$, and (b) $M = 4M_0$.

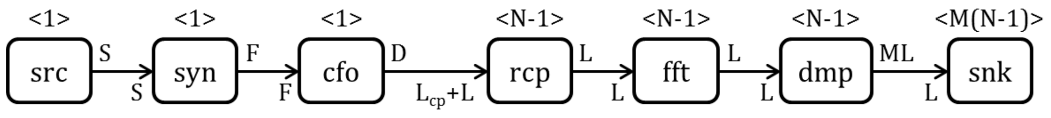
Table 4.2: The running times (in seconds) of the ALV methods on a specific SDF graph with 22 nodes and 33 edges.

	TMSV Σ-IAV	TMSVPB Σ-IAV	NIAV	MBD
$M = 2M_0$	2.0	8.4	320	0.1
$M = 4M_0$	13.0	35.9	3500	0.7

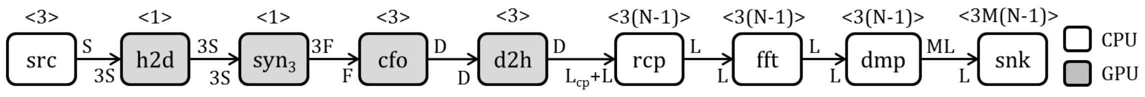
The list of the ALV methods sorted from the fastest to the slowest are: MBD, Σ-IAV with the TMSV score function, Σ-IAV with the TMSVPB score function, and NIAV. Table 4.2 shows the running times of the ALV methods on a specific graph with 22 nodes and 33 edges. This graph is selected randomly to provide further insight into variations in the running time among the four ALV methods.

In our experiments, we find that typically MBD finishes within 1 second, while the running times of the two Σ-IAV methods usually range from several seconds up to a few minutes. We expect that this kind of running time profile is acceptable in many coarse grain dataflow design scenarios in the embedded signal processing domain, where actors typically perform higher level signal processing operations, and therefore the number of nodes in the graphs is limited compared to other types of dataflow graphs that are based on fine-grained actors.

The running time of NIAV is generally the longest among all four methods, and grows rapidly with the number of nodes. In our experiments with an SDF graph having 30 nodes, for example, NIAV takes 3 hours to finish its computation. The fast growth of the running time in NIAV arises because the algorithm maintains information about multiple search paths in the vectorization space, which in turn results in the need to keep track of multiple, intermediate vectorized graphs. Therefore, NIAV is more suitable in situations when the SDF graph is relatively small,



(a)



(b)

Figure 4.8: SDF model of OFDM-RX application. (a) The original graph. (b) The transformed graph that results from vectorization of *syn* by a factor of 3, and insertion of the actors *h2d* and *d2h*.

design turnaround time is not critical, or solution quality is of utmost importance. Additionally, NIAV has various parameters that can be experimented with to trade off solution quality and running time. These parameters can be used to configure NIAV into a form that is more suitable for a specific design context. Deeper investigation into the configurability of NIAV is a useful direction for future work.

4.7 Case Study: OFDM

In this section, we demonstrate the effectiveness of our new ALV-integrated software synthesis framework through a case study involving an *orthogonal frequency-division multiplexing* (*OFDM*) receiver (*OFDM-RX*). The OFDM-RX is an adapted version of the OFDM system described in [53]. Figure 4.8 shows an SDF model for the OFDM-RX application. The value above each actor in Figure 4.8 gives the repetition count of the actor. Table 4.3 lists the actors in this SDF model and describes their corresponding functions. The system can operate with different parameter values, as shown in Table 4.4.

Table 4.3: Actors in the OFDM-RX application.

Actor	Description
src	Read samples of the input signal.
syn	Perform time-domain synchronization.
cfo	Remove carrier frequency offsets.
rcp	Remove cyclic prefix.
fft	Perform Fast-Fourier Transform on symbols.
dmp	Map OFDM symbols into bit stream.
snk	Write bit stream onto the output.

Table 4.4: Parameters in the OFDM-RX application model, along with the settings or ranges ("values") of these parameters that we use in our experiments.

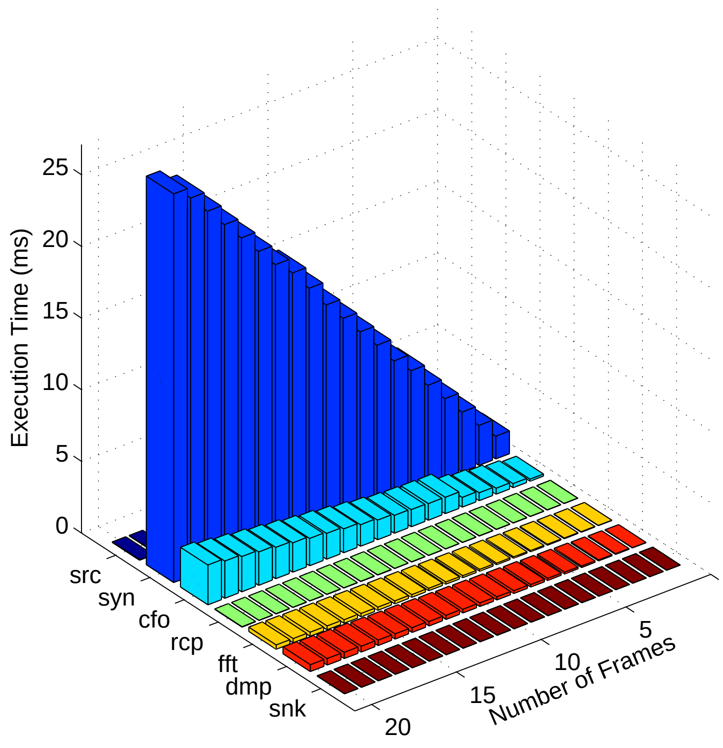
	Description	Values
L	Number of subcarriers per OFDM symbol	[128, 256, 512, 1024]
N	Number of OFDM symbols per frame	10
L_{cp}	Length of cyclic prefix for each OFDM symbol	$(9/128)L$
M	Number of bits per sample	4
D	Length of data excluding training symbols	$(N-1)(L+L_{cp})$
F	Length of a frame	$N(L+L_{cp})$
S	Size of sample stream	$2F$

Multiple forms of data parallelism can be exploited in OFDM-RX at different levels: (1) *Frame Level*: multiple frames can be processed in parallel in *syn* and *cfo*; (2) *Symbol Level*: OFDM symbols can be processed in parallel in *rcp*, *fft*, and *dmp*; (3) *Subcarrier Level*: Computation involved with arrays of subcarriers, such as convolution, FFT computations and vector operations, can be parallelized within each actor.

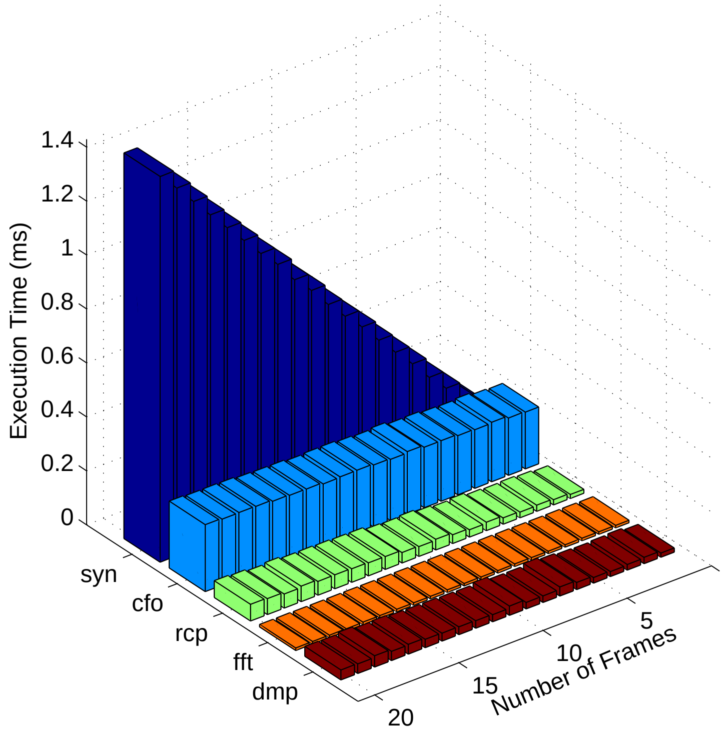
4.7.1 System Implementation and Profiling

We have implemented the OFDM-RX actors using the *Lightweight Dataflow Environment* (*LIDE*), which provides a programming methodology and associated application programming interfaces (APIs) for implementing dataflow graph actors and edges in a wide variety of platform-oriented languages, such as C, C++, CUDA, and Verilog [21, 35]. In our OFDM-RX system, GPU-accelerated implementations are available for all actors other than the *src* and *snk* actors. The *src* and *snk* actors are not mapped to the GPU in our implementation because of input/output operations that are involved in these actors.

We have profiled the execution times for the OFDM-RX actors on both the CPU and GPU. Figure 4.9 summarizes the Average execution Times per SDF graph Iteration (ATSIs) for the actors, as derived through this profiling process. The ATSI $t_T(v)$ for an actor v can be expressed as $t_T(v) = q(v)t(v)$, where q represents the repetitions vector of the enclosing SDF graph, and $t(v)$ represents the average execution time measured for a single firing of v. These execution time estimates are measured on both the CPU and GPU when $L = 256$, and the actors are vectorized to process different numbers of data frames per vectorized invocation. Observe from Figure 4.9 that the distribution of the ATSIs in OFDM-RX are uneven, and that the *syn* and *cfo* actors dominate the execution times both on the CPU and GPU. Also, observe that although actor execution times are roughly proportional to the number of frames N_F, they increase at different rates in relation to N_F — for example, $t_T(cfo)$ on the GPU grows very slowly with increases in N_F, and $t_T(syn)$ grows much faster.



(a)



(b)

Figure 4.9: ATSIs on the CPU and GPU when actors are vectorized to process multiple frames in each firing. (a) CPU. (b) GPU.

4.7.2 Software Synthesis with GLV-HEFT

We first measure the performance improvement achieved by GLV-HEFT when integrated in our DIF-GPU software synthesis framework. Here, we measure the system throughput under 11 different configurations without any memory constraints imposed. These measurements are performed on software implementations that are generated automatically using DIF-GPU integrated with GLV-HEFT.

In contrast to the relative throughput metric (see Section 4.4) that is used as a general performance metric in Section 4.6, the throughput metric we that employ in this section and in Section 4.7.3 is *frames per second*, which is of specific relevance to the OFDM-RX application.

We denote the results (throughput values) from these measurements by $Th_0, Th_1, \ldots, Th_{10}$. Here, Th_0, denotes the throughput when the input graph is not vectorized and all actors are mapped onto a single CPU core. On the other hand, for $b \in \{1, 2, \ldots, 10\}$, Th_b represents the throughput obtained when GLV is applied with GVD $= b$, and HEFT is used to schedule the resulting vectorized graph (GLV-HEFT) [20].

Figure 4.10 shows the speedup in throughput of GLV over the single-CPU implementation. The maximum measured speedups achieved here are 10.1X, 18.1X, 31.9X, 41.1X for $L = 128, 256, 512, 1024$, respectively. Table 4.5 compares Th_0 and Th_{10} in more detail for different values of L.

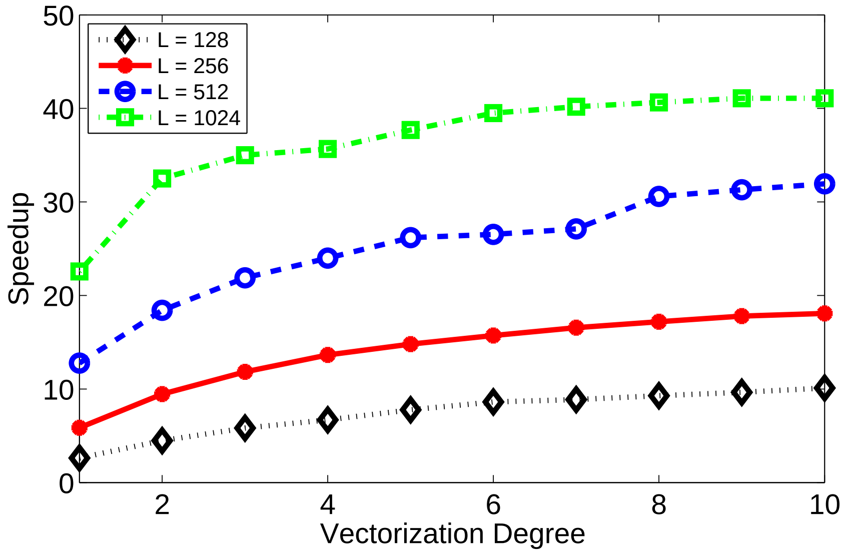


Figure 4.10: Speedup of the OFDM-RX application over a single CPU implementation for different GVD values and different values of the bandwidth parameter L.

Table 4.5: OFDM-RX system throughput.

L	Th_0 (10^3/s)	Th_{10}(10^3/s)
128	1.94	19.6
256	0.62	11.2
512	0.17	5.43
1024	0.044	1.8

4.7.3 Software Synthesis with ALV-IAR

In this section, we perform measurements and comparisons that involve software implementations that are generated automatically using DIF-GPU integrated with ALV-IAR. Recall from Section 4.6 that ALV-IAR applies TMSV Σ-IAV, TMSV-PB Σ-IAV, N-candidates IAV, and MBD, and then selects the best result from among the four derived solutions. The experiments are performed under different memory budgets and different levels of bandwidth L (an application-level parameter). For comparison, we apply DIF-GPU integrated with GLV-HEFT to synthesize software that incorporates vectorized schedules constructed using GLV-HEFT instead of ALV-IAR.

Table 4.6 shows an example of the vectorization degrees and processor assignments derived for OFDM-RX under a specific memory constraint. This memory constraint is selected to represent one that is neither very tight nor very loose. These vectorized scheduling results are derived by ALV-IAR, and the throughput is measured by executing the resulting software implementation that is synthesized by DIF-GPU. The vectorization and processor assignment (mapping) results are shown in Table 4.6 as lists of values that correspond to the graph actors in their topological order (*src*, *syn*, ..., *snk*). The numbers 0 and 1 in the Mapping column represent the CPU-core and GPU, respectively. The results in Table 4.6 show that ALV-IAR produces a 1.2X speedup compared to the baseline technique for the selected memory constraint.

The memory budgets are set to $M = b \log(L) \times 10^5$, where $b = \{1, 2, \ldots, 10\}$.

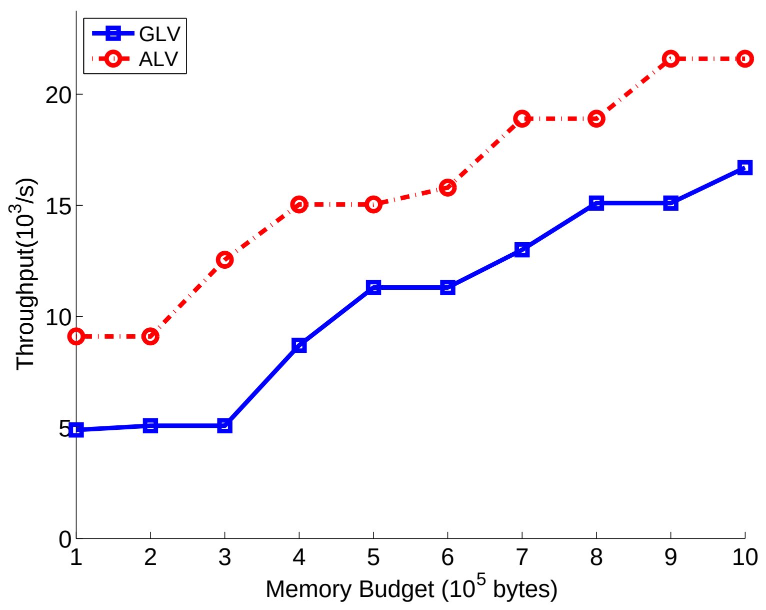
Table 4.6: Vectorization degrees and mapping results generated by ALV-IAR and GLV-HEFT under the memory constraint $M = 2.8$ Mb, and $L = 512$.

Method	Vectorization	Mapping	$Th(10^3/s)$
ALV-IAR	[1,3,12,1,1,1,1]	[0,1,1,0,0,0,0]	3.15
GLV-HEFT	[4,4,4,36,36,36,144]	[0,1,1,1,1,0,0]	2.60

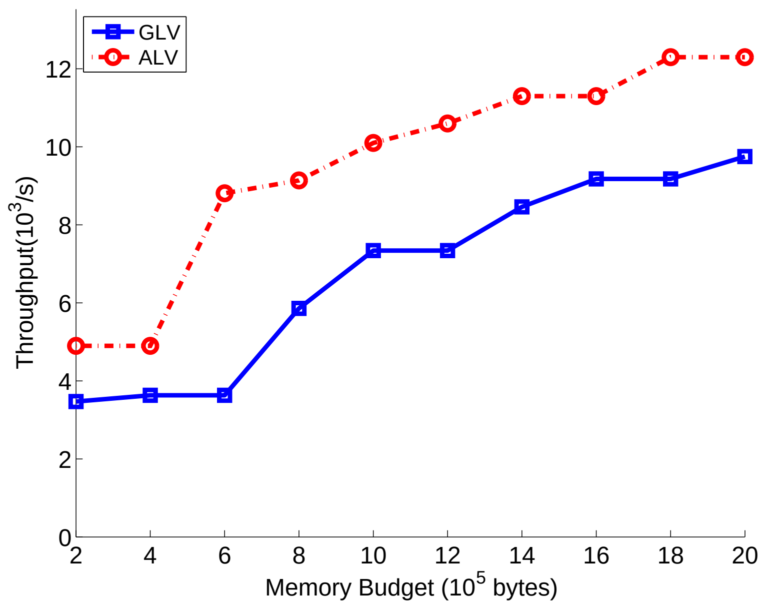
We compare the throughput levels of implementations generated using the two methods — ALV-IAR and GLV-HEFT — as shown in Figure 4.11. The results shown in Figure 4.11 show that using actor-level vectorization and Σ scheduling, we are able to obtain system throughput that consistently exceeds that provided by the baseline method under same memory constraint.

When memory constraints are relatively tight, GLV has difficulty in adequately exploiting data parallelism in the OFDM-RX system. ALV-IAR alleviates this problem by focusing memory resources to vectorize selected, performance-critical actors. Specifically, ALV-IAR successfully identifies *syn* and *cfo* as the two actors that benefit the most from vectorized execution on the GPU. Prioritizing the vectorization of these two actors helps to avoid wasting memory on vectorizations that have relatively little or no impact on overall system performance. This is reflected by a large throughput gain when $b \leq 4$. When the memory constraint is relaxed, the gap in the throughput gain between ALV-IAR and GLV is reduced, as data-parallelism in the system can exploited more effectively by GLV under loose memory constraints.

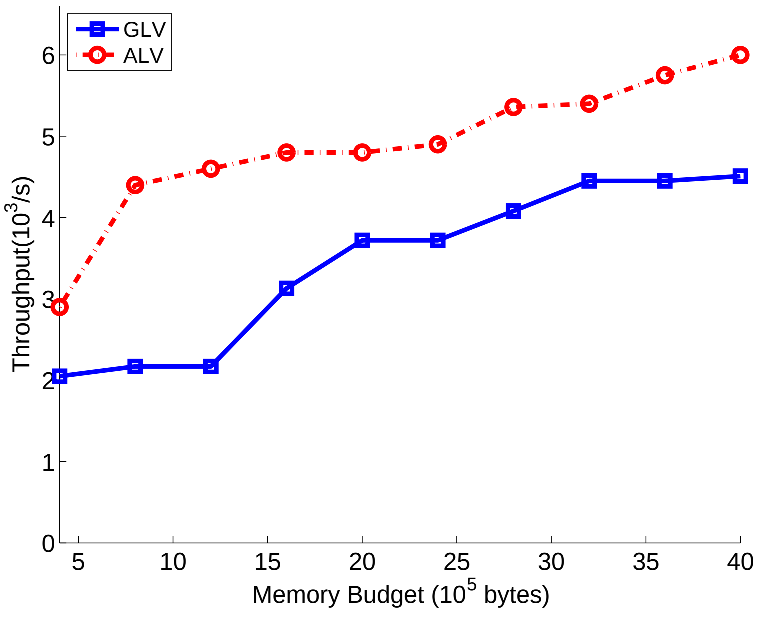
When optimizing the OFDM-RX system, ALV-IAR maps only *syn* and *cfo* onto the GPU, and assigns the other actors to the CPU to utilize pipeline parallelism in the system. Under this mapping, firings of *syn* and *cfo* from subsequent frames can be executed in parallel with firings of *rcp*, *fft*, *dmp* and *snk* from earlier frames.


GLV
ALV
Throughput(10^3/s)
Memory Budget (10^5 bytes)
0
5
10
15
20
1
2
3
4
5
6
7
8
9
10

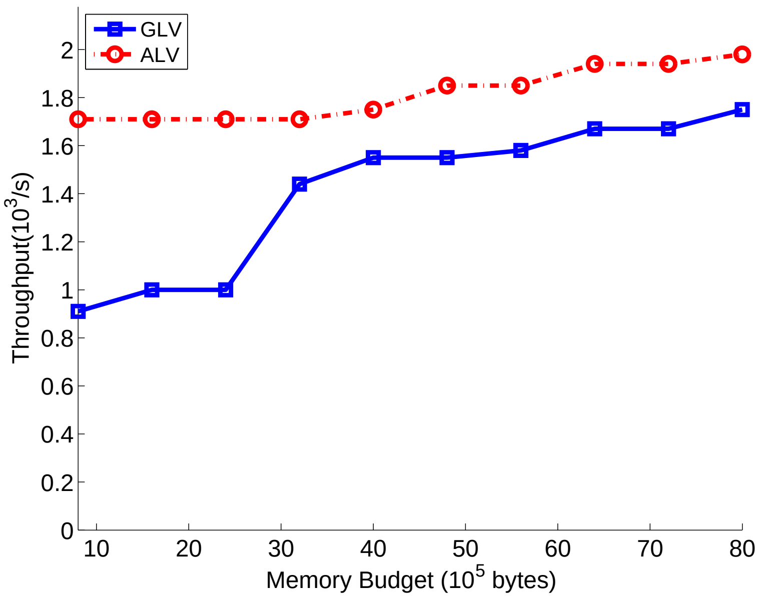
(a)


GLV
ALV
Throughput(10^3/s)
Memory Budget (10^5 bytes)
0
2
4
6
8
10
12
2
4
6
8
10
12
14
16
18
20

(b)



(c)



(d)

Figure 4.11: Memory-constrained throughput of OFDM-RX systems with different levels of memory budget M and bandwidth L using ALV-IAR compared to the GLV-HEFT baseline: (a) $L = 128$, (b) $L = 256$, (c) $L = 512$, (d) $L = 1024$.

In these experiments, the maximum measured speedup values of ALV-IAR over GLV-HEFT are 2.66X, 2.45X, 1.94X and 1.71X for $L = 128, 256, 512, 1024$, respectively. The maximum speedup values of ALV-IAR compared to a single-core, unvectorized CPU baseline implementation are 11.1X, 19.8X, 33.8X, and 47.6X, for $L = 128, 256, 512, 1024$, respectively.

In summary, the throughput improvement obtained by HCGP acceleration using the methods developed in this work facilitates real-time, memory constrained processing of OFDM signals that can benefit a variety of software-defined radio and cognitive radio applications.

4.8 Summary

In this chapter, we have investigated memory-constrained, throughput optimization for synchronous dataflow (SDF) graphs on heterogeneous CPU-GPU platforms. We have developed novel methods for Integrated Vectorization and Scheduling (IVS) that provide throughput- and memory-efficient implementations on the targeted class of platforms. We have integrated these IVS methods into the DIF-GPU Framework, which provides capabilities for automated synthesis of GPU software from high-level dataflow graphs specified using the dataflow interchange format (DIF). Our development of novel IVS methods and their integration into DIF-GPU provide a streamlined workflow for automated exploitation of pipeline, data and task level parallelism from SDF graphs. We have demonstrated our IVS methods through extensive experiments involving a large collection of diverse, synthetic SDF graphs, as well as on a practical embedded signal processing case study involving a wireless communications receiver that is based on orthogonal frequency division multiplexing. The results of our experiments demonstrate that our proposed new methods for IVS provide significant improvements in system throughput when mapping SDF graphs onto CPU-GPU platforms.

Chapter 5

Parameterized Sets of Modes

As previously described in Chapter 1, reconfigurable signal processing systems present challenges at many levels of design, including configuration, control and system-level optimization. To meet requirements of bandwidth, flexibility and reconfigurability, systematic methods to model and analyze cognitive radio designs on signal processing platforms are desired. To help address these challenges, we present in this chapter a novel dataflow modeling technique, called *parameterized set of modes* (*PSM*). PSMs allow efficient representation, manipulation and application of related groups of processing configurations for functional design components in signal processing systems. PSMs lead to more concise formulations of actor behavior, and a unified modeling methodology for applying a variety of techniques for efficient implementation. In the following sections, we develop the formal foundations of PSM-based modeling, and demonstrate its utility through two case studies involving the mapping of reconfigurable wireless communication functionality into efficient implementations.

5.1 Introduction

Recent developments in embedded systems and applications have motivated new research towards design methodologies for configurable, high-performance embedded software. As an example, Cognitive Radio (CR) enables a wireless transceiver to cognitively manage its wireless spectrum for improved agility and efficiency. Flexibility and reconfigurability of the implementation at various layers, including RF, baseband, and MAC layers, with cross-layer modeling and control, will be important to realize the efficiency potential of spectrum sharing. Realizing the potential of cognitive radio will also require transceivers to dynamically reconfigure communication parameters based on multidimensional criteria, including channel conditions, link performance, and user requirements. Meanwhile, increasing bandwidths and data rates pose new challenges to the baseband (BB) processing chain, as well as to radio frequency (RF) processing. Therefore, *Software Defined Radio* architecture is needed for Cognitive Radio systems to meet the requirements of configurability, agility, etc., and to efficiently utilize various kinds of high-performance computing devices, ranging from multi-core programmable digital signal processors (PDSPs), streaming SIMD extensions (SSE), to general purpose graphics processing units (GPGPUs).

On the other hand, practical and systematic approaches to reconfiguration based on programmable paradigms are still lacking. For example, software-based adaptive configuration of radio frequency chains is still in its infancy, but is a key ingredient of the frequency agile radios needed for cognitive devices and flexible RF spectrum use. The trend of increasing diversity and flexibility in both the functionality and the computational platforms of wireless systems results in complex design spaces that must be considered during design and implementation. The complexity of these design spaces and their novel constraints strongly motivate the development of new design methodologies.

To express dynamics in complex signal processing applications like Cognitive Radios, a number of dynamic dataflow models have been proposed, including *parameterized synchronous dataflow* (*PSDF*) [54], *Boolean dataflow* (*BDF*) [32], and *core functional dataflow* (*CFDF*) [8]. PSDF provides semantics to manipulate application parameters in dataflow models at run-time. BDF introduces special control actors to allow data-dependent invocation of actors. CFDF applies the concept of actor "modes", where different modes can have differing dataflow behavior, and mode transitions can be data-dependent. CFDF is tailored to natural design of actors with dynamic functionality, and facilitates prototyping of dataflow applications, as well as identification of more specialized dataflow behaviors [18], such as BDF, cyclo-static dataflow (CSDF) [17] or synchronous dataflow (SDF) [16].

When using CFDF, a designer specifies the behavior of the different modes of each CFDF actor, and the transitions among these modes. However, as the number of modes grows and the mode transitions become more complex, CFDF formulations can become unwieldy in terms of actor specification, analysis and implementation. In this chapter, we present a novel modeling method, called *parameterized set of modes* (*PSM*), which is a high-level abstraction that efficiently represents parameterized functionality within groups of related modes for CFDF actors. PSMs enable novel ways for representing, manipulating and applying related groups of actor modes that lead to more concise formulations of actor behavior, and a unified modeling methodology for applying a variety of techniques for efficient implementation. We develop the formal foundations of PSM-based modeling, and demonstrate its utility through two case studies involving the mapping of reconfigurable wireless communication functionality into efficient implementations.

Material described in this chapter has been published in [55].

5.2 Related Work

A technique called *mode grouping* for CFDF graphs has been developed in [56]. It is demonstrated that mode grouping can improve scheduling results by aiding the discovery of statically schedulable subgraphs. In [57], CFDF is applied in simulation of dynamic communication systems. CFDF modeling is also applied as the semantic basis for the lightweight dataflow design environment, which is introduced and applied to design and implementation of wireless communication systems in [58]. These works apply the CFDF model in various useful ways, but are unable to streamline their associated analysis or implementation when manipulating groups of modes that are related through parameterization. The mode-based parameterization techniques introduced in this chapter are developed to bridge this gap.

Various research efforts have been directed towards integrating dynamic behavior into dataflow models. In [59], a design framework called SysteMoc is developed for applying dataflow structures, similar to those used in CFDF, involving guarded invocations and state transitions specified by finite state machines (FSMs). The work also includes design space exploration and code synthesis for FPGA platforms. In [60], SysteMoc is applied to perform dynamic partial re-configuration of SDF graphs that are mapped on FPGAs. In [61], a dataflow based analysis method is proposed for SDR applications. This method adopts the concept of “SDF scenarios” to incorporate some degree of dynamism for better estimation of system resource requirements and throughput. Moreover, methods for quasi-static scheduling of statically-schedulable sub-graphs within larger dynamic dataflow graphs are explored in [62].

In the context of the related work described above, the main contributions of this chapter are described as follows. We enhance the CFDF model of computation by introducing the concept of parameterized set of modes (PSM), which incorporates dynamic parameterization into actor modes, thereby increasing the effectiveness with which designers can design and implement CFDF-based, dynamic dataflow models for signal processing systems. PSM-based modeling of actors provides a common framework for integrated specification, analysis and implementation that deeply integrates mode- and parameter-based actor characterizations. Although we develop the PSM model in the context of CFDF in this chapter, we envision that the ideas can be adapted to related dataflow modeling and programming techniques, such as, for example, SysteMoc [59] and CAL [63]. Exploring and applying such adaptations is a useful direction for future work.

5.3 Formulation of PSMs

In this section, we define the concept of *parameterized set of modes* (*PSM*), for incorporating dynamically parameterized modes efficiently into the CFDF modeling framework.

5.3.1 Notation

To develop the PSM concept precisely, we first introduce some notation and review the definition of the CFDF model of computation. For a given dataflow graph actor A, we denote the set of input ports of A by $in(A)$. We also denote the set of nonnegative integers by N, and the set of Boolean values by B. We denote the values in B as *true* and *false*.

When using PSMs, we allow CFDF actors to have arbitrary sets of parameters. Following notation similar to that of parameterized dataflow graphs [54], we denote the set of parameters of a given actor A as $param(A)$, and for each parameter in $p \in param(A)$, we denote the set of permissible values of p as $domain(p)$. At any given point during dataflow graph execution, an actor parameter p has associated with it a unique parameter value $v \in domain(p)$, which is referred to as the *configuration* of p at that point in time. A *configuration* for A can then be specified as a set of configurations for all of the parameters in $param(A)$. Some combinations of possible parameter values may be considered invalid because they do not make sense together. The set of all valid configurations for A is denoted as $DOMAIN(A)$. At a given point during execution, the specific configuration for A that is determined by its current parameter values is referred to as the *active configuration* of A. Similarly, the specific mode that a CFDF actor is in during a given firing is referred to as the *active mode* for the actor.

If S_1 and S_2 are sets, then by $S_1 \subset S_2$, we mean that S_1 is a subset (not necessarily a proper subset) of S_2. Thus, S_1 can be empty, equal to S_2, or a proper subset of S_2.

5.3.2 Motivation for Parameterized Sets of Modes

The CFDF formulation can become unwieldy when working with parameterized actors that have large parameter sets, especially if one or more actor parameters can affect the production and consumption rates of an actor. For example, consider a parameterized downsampler actor that provides an $N : 1$ downsampling of its input signal. Such an actor requires N distinct modes in its CFDF specification even though the operation of all N alternative modes have closely related (parameterized) functionality. Using the PSM concept introduced in this section, we can group all of these related modes together into a single *mode set* σ, where the individual mode in σ that is active during any given actor firing is determined uniquely by the actor parameter set (in this case, by the parameter N).

As a slightly more elaborate example, consider an actor S that can function either as a downsampler or an upsampler depending on its configuration. Such an actor could be useful, for example, as part of a programmable, multistage subsystem for sample rate conversion. This actor can be parameterized with two parameters u and N, where u is Boolean-valued and indicates whether or not S functions as an upsampler (if $u = \mathit{false}$, then the actor functions as a downsampler), and N provides the upsampling or downsampling factor. Using the PSM concept, this actor can be specified precisely using two mode sets — one for the upsampling-related modes, and the other for the downsampling-related modes. In any given mode set, the production and consumption rates are determined uniquely by the actor parameters. For example, in the mode set associated with upsampling ($u = true$), $N = 3$ yields a consumption rate of 1 and production rate of 3.

Intuitively, a *PSM-enhanced CFDF* specification, or *PSM-CFDF* specification, allows an actor's modes to be grouped into "clusters" or sets that have related functionality, and are therefore efficient to work with as distinct units — e.g., in terms of design tasks such as specification, analysis, optimization, profiling, and integration. In general, the actor groups may overlap, but collectively, they should "cover" the entire set of modes of the associated CFDF actor. Furthermore, the actor groups in a PSM-based specification should be related uniquely to the actor modes through the parameters of the given actor.

5.3.3 Formal Definition of PSM-CFDF

Given a PSM-CFDF actor A with mode set M_A, a PSM ρ for A is a 3-tuple $\rho = (S, C, f)$, where $S \subset M_A$, $C \subset DOMAIN(A)$, and $f : C \to S$. The set C, denoted as $psa\_domain(\rho)$, can be viewed as the set of possible actor configurations when the actor is firing in mode set S. The set S, denoted $psa\_modeset(\rho)$, is the set of modes in actor A that is associated with ρ — i.e., whenever A fires in PSM ρ, it fires one of the modes within $psa\_modeset(\rho)$. Finally, the mapping f, denoted F_ρ, specifies the unique mode within $psa\_modeset(\rho)$ that is active whenever A executes in mode set S and a given actor configuration is active.

Given a PSM-CFDF actor A with mode set M_A, and a set R of PSMs for A, we say that R *covers* A if every mode in M_A is contained in the mode set of at least one element of R — that is, if

$$M_A = \bigcup_{\rho \in R} psa\_modeset(\rho). \tag{5.1}$$

A PSM-CFDF actor A is a CFDF actor with an associated set R of PSMs that covers A, and a family of mappings $\{psa\_next_{r,c} : I(F_r(c)) \to R \mid r \in R \text{ and } c \in DOMAIN(A)\}$. Here, for a given mode $m \in M_A$, $I(m)$ denotes the set of all possible combinations of inputs — i.e., all possible n-tuples of token vectors, where $n = |in|$, and the size of (number of elements in) each token vector is equal to the consumption rate of the corresponding port in mode m.

In other words, for each pair (r, c), there is a mapping $psa\_next_{r,c}$, called the *next PSM function of PSM r under actor configuration c*, that determines uniquely a specific mode $m\prime$ for any given input data set for that mode; this mode $m\prime$ can be interpreted as the *next PSM* for the actor — i.e., the PSM that should be active for the next firing of A.

For a PSDF-CFDF actor A, we denote the associated set of PSMs at $PSMset(A)$, and the associated family of mappings as $mappings(A)$.

The next PSM function is related to the invoking function of A, as defined by CFDF semantics. In particular, for a given actor firing, the next mode, as determined by the invoking function, should agree with (be an element of) the next PSM, as determined by $psa\_next(r, p)$. For details on the CFDF invoking function, we refer the reader to [8].

The concept of PSM is a level of abstraction that helps the designer to better understand and expose connections between the actor's parameters and modes. PSM analysis can be combined with various processes in a design framework, such as scheduling and processor selection, to name a few. By grouping into a single PSM the modes of an actor that share some common property, a designer can manipulate the associated modes and apply aspects of the property in an integrated and systematic way.

5.3.4 PSM Transition Graph

For a PSM-CFDF actor, the next PSM function defines the range of modes in which the actor executes in its next invocation. The structure of transitions among PSMs therefore can provide valuable information about the actor's dynamic behavior. These transitions can be expressed formally by a construction that we call the *PSM transition graph.*

The PSM transition graph for a PSM-CFDF actor A is a directed graph $G_{psm} = (V_{psm}, E_{psm})$, where V_{psm} is the set of vertices and E_{psm} is the set of edges. The set of vertices is in one to one correspondence with the PSMs of A; the PSM transition graph vertex associated with PSM r is denoted as $v_{psm}(r)$. Two PSM transition graph vertices $v_{psm}(x)$ and $v_{psm}(y)$ are connected by a directed edge $e = (v_{psm}(x), v_{psm}(y))$ if there exist an input vector ν and a configuration c such that $y = psa\_next_{x,c}(\nu)$. Such an edge e is annotated with a *label*, $label(e) = c$. Note that multiple edges can have the same label if different next PSMs are "reachable" from the same current PSM and same configuration under different input vectors. Compared to finite state machine (FSM) representation of state transitions, the PSM transition graph contains higher level information on the structure of PSMs. Such higher level structure may be difficult to extract or intuitively understand from conventional FSM-style representations (i.e., where each mode corresponds to a separate FSM state), especially when the number of modes is large or their connections are irregular.

Figure 5.1(b) shows an example of a PSM transition graph. Further details about the actor in this example are discussed in Section 5.5.4.

5.3.5 Implementation Considerations

When implementing a PSM-CFDF actor, we do not anticipate that designers will typically need to explicitly implement the mappings (mathematical functions) F_ρ and $psa\_next\{r, p\}$. These mappings are useful as analytical tools, but their explicit realization in software is not in general essential for the PSM-CSDF model — e.g., an actor designer would not need to provide a software function/method or hardware description language module that is dedicated to implementing each of these mappings. Instead, for example, critical aspects of F_ρ may be validated through unit testing, and the next PSM may be determined as a by-product of actor firing — e.g., through an actor-level application programming interface (API) that is used by schedulers to invoke the actor.

5.3.6 Application Example

In this section, we show an example of applying PSM-CFDF concepts in actor design for a reconfigurable OFDM demodulator that is geared towards cognitive radio systems. Such systems can involve significant amounts of parameterization in actor designs. Figure 5.1(a) shows a parameterized demodulator actor that supports different operational modes, including QPSK and QAM16. The actor maps the B samples into an $M \times B$ bit stream. This actor has two parameters: M for the number of bits per sample, and B for the vectorization degree (see [34] for fundamental developments on actor-level vectorization for signal processing dataflow graphs). Since M represents the number of bits for each symbol, $M = 2, 4$ correspond, for example, to QPSK, QAM16, respectively. B can take on any integer value between 1 and B_{max}, where B_{max} is the maximum vectorization degree (e.g., as a designer or design tool might set based on memory constraints). The parameter B allows symbols to be buffered and processed together in batches (block processing). For example, if $B = 1$, then each actor invocation processes a single input symbol; if $B = 10$, then 10 symbols are buffered and processed together in one invocation.

The de-mapper in Figure 5.1(a) is modeled as a PSM-CFDF actor A as follows. Actor configurations are specified in the form (M, B). The set of modes of the actor is given as:

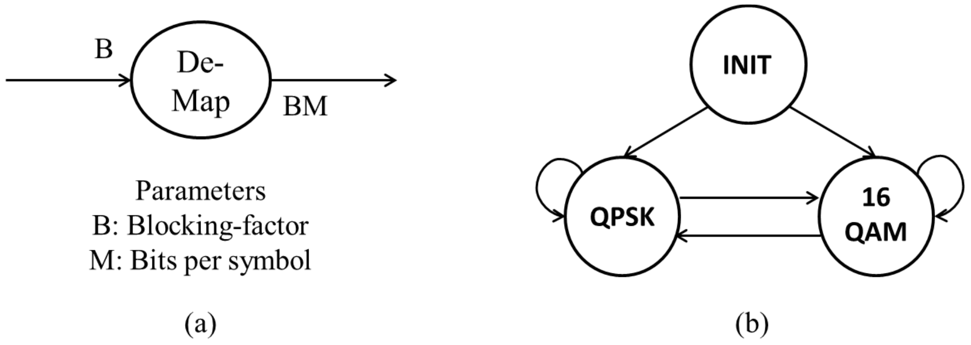


Figure 5.1: An example of a PSM-CFDF actor: OFDM demapper example. (a) Actor interface. (b) PSM transition graph.

$$M_A = \{INIT, QPSK_1, QPSK_2, \ldots, QPSK_{B_{max}}, \\ QAM16_1, QAM16_2, \ldots, QAM16_{B_{max}}\} \tag{5.2}$$

Based on the functionality, M_A can be clustered into 3 PSMs: $\{\rho_i = (S_i, C_i, f_i) \mid i = 1, 2, 3\}$, where $S_1 = \{QPSK_n \mid 1 \leq n \leq B_{max}\}$, $C_1 = \{(2, n) \mid 1 \leq n \leq B_{max}\}$, $f_1(M, B) = QPSK_B$; $S_2 = \{QAM16_n \mid 1 \leq n \leq B_{max}\}$, $C_2 = \{(4, n) \mid 1 \leq n \leq B_{max}\}$, $f_2(M, B) = QAM16_B$; $S_3 = \{INIT\}$, $C_3 = \{(m, n) \mid m = 2, 4; 1 \leq n \leq B_{max}\}$, $f_3(M, B) = INIT$.

Based on this decomposition into PSMs, Figure 5.1(b) illustrates the PSM transition graph for the demapper actor. Upon initialization or reset, the actor enters the *INIT* mode, the only mode in ρ_3. After initialization, the actor enters a mode in ρ_1, or ρ_2, based on the configuration. For any mode in ρ_1, the ratio of the production rate $prd(A)$ to the consumption rate $cns(A)$ is 2. Similarly, for any mode in ρ_2, $prd(A)/cns(A) = 4$. To avoid clutter in the diagram, edge labels are not shown.

5.3.7 Summary

In this section, we have presented an enhancement to the framework of CFDF modeling called parameterized sets of modes (PSM), and we have introduced the PSM-CFDF approach to the modeling of dynamic dataflow actors with dynamically variable parameters. To illustrate the approach, we have presented a detailed example of an OFDM demapper actor that is modeled in terms of PSM-CFDF semantics. This example and its associated PSM transition graph representation concretely illustrate the novel form of higher level modeling structure that is exposed by the PSM modeling concept and the associated PSM-CFDF design methodology.

5.4 PSM-level Static Scheduling for CFDF Graphs

In this section, we demonstrate the application of PSM to efficient scheduling of CFDF-based programs.

A general scheduling approach for CFDF graphs is the so-called *canonical scheduling* approach discussed in [18]. In canonical scheduling, a sequential ordering L of the dataflow graph actors is constructed [18]. At run-time, the scheduler iteratively traverses the list L, and upon visiting each actor A, the scheduler checks the enabling condition (availability of sufficient input data) for A, and invokes A if the enabling condition is satisfied. This scheduling approach is useful in the sense that it is very general (applicable to any CFDF graph), easy to understand, and easy to implement. However, the efficiency of canonical scheduling can be relatively low because of the frequency with which enabling conditions must be checked.

5.4.1 Statically Schedulable Regions

Static schedules, where the sequence of actor firings is deterministic and unconditional (not guarded by actor-level checking of enabling conditions) can be significantly more efficient and predictable compared to dynamic scheduling approaches, such as canonical scheduling. Even if the overall dataflow graph does not allow for static scheduling (due to the presence of dynamic dataflow), it may be possible to identify "statically schedulable regions" of the graph — i.e., parts of the graph that can be scheduled statically. Such regions can be scheduled using efficient static scheduling techniques, which have been developed extensively in the literature (e.g., see [7]), and then the static schedules for the different regions can be integrated through a "top-level" dynamic scheduling mechanism.

In this section, we develop PSM-based methods for constructing and applying statically schedulable regions for efficient implementation of CFDF graphs. The concept of statically schedulable regions itself is not new, and has been studied in depth, for example, in the implementation of CAL programs [19]. Our contribution in this section, which we refer to as *PSM-level static scheduling*, is to demonstrate methods for integrating the concepts of PSMs and statically schedulable regions, therefore combining the benefits of both approaches, and enabling structure exposed from PSMs to help guide the construction of efficient schedules. More specifically, in our development of PSM-level static-scheduling, we utilize information about actor parameters to form hierarchical PSMs, where each hierarchical PSM is constructed based on combinations of actor modes that share common scheduling properties.

In the remainder of this section, we outline our proposed PSM-level static scheduling approach and present experimental results on an application example.

5.4.2 PSM-level Static Scheduling

PSM-level static scheduling is a hierarchical scheduling technique, where subgraphs within a dataflow specification are combined into *hierarchical actors*, and execution of a hierarchical actor corresponds to execution of a schedule for the associated subgraph. If H is a hierarchical actor with associated subgraph G, we say that H *encompasses* G, and G is the *nested subgraph* of H.

In the class of CFDF-PSM specifications addressed in this work, a hierarchical actor contains a set of modes, and can also contain a set of PSMs, just as non-hierarchical (leaf-level) actors. In the case of a hierarchical actor H, each mode m of H corresponds, respectively, to a mapping $Z_m : V_e \rightarrow \gamma$, where $G_e = (V_e, E_e)$ denotes the graph encompassed by H, γ is the set of all actor modes across all actors in V_e, and $Z_m(v) \in M_v$ for all v. Recall here that M_v represents the set of modes for a given actor v.

Intuitively, execution of H in a given mode $m \in M_H$ corresponds to execution of the encompassed graph with all actors operating in the modes specified by Z_m. The duration (termination criterion) of such an execution is a design issue associated with the construction of H, similar in some ways to the concept of "subsystem iteration" in parameterized dataflow [54]. In this chapter, we assume that each execution of H in a given mode m corresponds to execution of a minimal static periodic schedule of the SDF graph, denoted $G_{sdf}(H, m)$, that results from fixing the actors in G_e based on the mode assignments specified by Z_m. Exploration of other kinds of termination criteria in this context is a useful direction for further work.

In our development of PSM-level static scheduling in this chapter, we assume that the hierarchical actors employed are provided as part of the specification — i.e., as part of the design hierarchy. Another interesting direction for future work is in the development of automated methods to group (cluster) subgraphs into hierarchical actors for PSM-level static scheduling.

5.4.3 Construction of SDF Scheduling PSMs

Building on the concepts introduced in Section 5.4.2, we introduce a simple method to partition the mode set M_H of a hierarchical actor H in a manner that facilitates construction of statically schedulable regions. This leads to a unique partitioning of M_H into a set of PSMs that we refer to as *SDF scheduling PSMs*. The method is useful in systematically decomposing the structure of a hierarchical PSM-CFDF actor in a manner that that captures subsystem-level, multi-mode behavior that is common in cognitive radio systems.

The process of constructing SDF scheduling PSMs operates by iterating through all modes in H, and dividing the modes into subsets (PSMs) $S_1, S_2, \ldots, S_k$, where all modes in a given S_i correspond to the same SDF repetitions vector for the encompassed graph $G(e)$. In other words, if $m_1, m_2 \in S_i$, and $a \in V_e$, then

$q_1(a) = q_2(a)$, where q_1 and q_2 denote, respectively, the SDF repetitions vectors of $G_{sdf}(H, m_1)$ and $G_{sdf}(H, m_2)$. The resulting mode sets $S_1, S_2, \ldots, S_k$ are then parameterized with one more scheduling parameters that can be configured and adapted based on considerations such as the given performance constraints, repetitions vectors q_i, and structure of $G(e)$. This process depends on fundamental properties of the SDF repetitions vector and requires that the set of SDF graphs $\{G_{sdf}(H, m) \mid m \in M_h\}$ satisfy SDF consistency conditions. For details on SDF fundamentals and consistency conditions, we refer the reader to [16].

In cognitive radio systems, actors can often be configured statically or dynamically by various parameters, resulting in large sets of possible actor modes. If the actors' mode spaces are viewed independently, the total number of possible mode combinations to consider can grow exponentially, making the system unwieldy and inefficient for scheduling analysis. The integration of PSM techniques to hierarchical CFDF modeling techniques, as introduced in this section, introduces an alternative, more compact designs space — the design space of scheduling parameters for the PSMs $S_1, S_2, \ldots, S_k$ — that facilitates efficient scheduling, including the application of SDF scheduling techniques to statically schedulable regions.

5.4.4 Synthetic Example

To illustrate the PSM-level static scheduling technique introduced in Section 5.4.2 and Section 5.4.3, Figure 5.2 shows a synthetic CFDF graph with 2 parameters, p_1 and p_2. Intuitively, the parameters p_1 and p_2 control (select) the

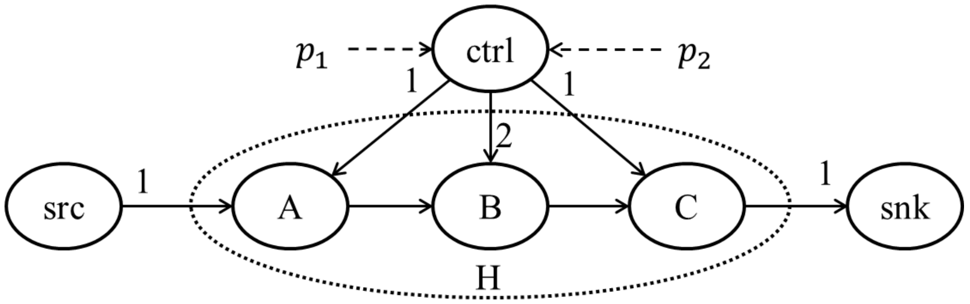


Figure 5.2: A synthetic CFDF graph that is used to illustrate PSM-level static scheduling concepts.

modes of A and C, respectively, and p_1 and p_2 together control the mode of B. The parameter values and their corresponding actor modes, production rates, and consumption rates are shown in Table 5.1. Here, the special actor *ctrl* reads parameter values from an input source (e.g., a file), checks their validity, and sends them as tokens to A, B and C.

Now suppose that H is a hierarchical actor that encompasses the subgraph associated with actors A, B, and C. The actors enter "initialization modes" A_0, B_0 and C_0, respectively, upon system reset, and wait for parameter tokens that are passed from *ctrl*. After receiving the parameter values, the actors continue to their respective operational modes, as specified by the received parameters, until all data from *src* has been processed.

Analyzing the repetitions vectors in M_H, and the mode space of H, and constructing SDF scheduling PSMs leads to the PSMs outlined in Table 5.2. The common repetitions vectors in the same scheduling PSM allows a common static schedule to be applied across all modes in that PSM. For example, for PSM_1, the static schedule $\sigma_1 = ABC$ can be applied as the schedule for H. Similarly, for all

Table 5.1: Details of actor parameters, modes, and dataflow rates.

Actor	Configuration	Mode	Prod	Cons
A	N/A	A_0	0	(0,1)
	$p_1 = 0$	A_1	1	(2,0)
	$p_1 = 1$	A_2	1	(1,0)
B	N/A	B_0	0	(0,2)
	$(p_1, p_2) = (0, 0)$	B_1	2	(1,0)
	$(p_1, p_2) = (0, 1)$	B_2	2	(2,0)
	$(p_1, p_2) = (1, 0)$	B_3	1	(2,0)
	$(p_1, p_2) = (1, 1)$	B_4	1	(1,0)
C	N/A	C_0	0	(0,1)
	$p_2 = 0$	C_1	1	(2,0)
	$p_2 = 1$	C_2	1	(4,0)

Table 5.2: Scheduling PSMs of the hierarchical actor H.

PSMs	Mode of H	Mode of ABC	**q**
PSM₁	H_0	$A_0\ B_0\ C_0$	(1,1,1)
	H_1	$A_1\ B_1\ C_1$	(1,1,1)
	H_2	$A_2\ B_4\ C_2$	(1,1,1)
PSM₂	H_3	$A_1\ B_2\ C_2$	(1,1,2)
	H_4	$A_2\ B_3\ C_1$	(1,1,2)

modes in PSM_2, we can apply the static schedule $\sigma_2 = AB(2C)$. Here, we apply looped scheduling notation, where a parenthesized term of the form (mX), where m is a non-negative integer (or a symbolic expression that resolves to a non-negative integer) and X is a sequence of actor firings, represents the successive execution m times of the sequence X. For background on the construction and manipulation of looped schedules for synchronous and parameterized dataflow graphs, we refer the reader to [64, 54].

For the entire application graph in this example, we can apply the schedule $\sigma_{top} = src\sigma_H(nsnk)$, where n is the mode-dependent firing rate (iteration count) for snk, and σ_H is configured dynamically as σ_1 or σ_2 based on the currently-active scheduling PSM.

We constructed the PSMs and schedules outlined here by hand, and based on these constructions, we implemented this synthetic application graph using the lightweight dataflow environment (LIDE), which is a tool for experimenting with dataflow techniques in arbitrary simulation- or platform-oriented languages, such as C, CUDA, MATLAB, and Verilog [58, 35]. Specifically, in our experiments we employed LIDE-C and LIDE-CUDA, which are C- and CUDA-oriented versions of the LIDE environment, respectively.

We implemented each actor as a simple sample rate converter that inserts or discards tokens to achieve the specified dataflow rates. The experiment is carried out using a desktop computer equipped with an Intel Core i7-2600K 8-core CPU, and 16GB memory. Figure 5.3 shows the execution time of the graph using CFDF canonical scheduling and PSM-level static scheduling. For our implementation of PSM-level static scheduling, we used the hierarchy of schedules σ_{top}, σ_1, and σ_2 defined above. In this example, the average execution time improvement of PSM-level static scheduling among the different modes of H is 11.9%.

Although it is based on a synthetic dataflow graph, the simplicity of this example helps to demonstrate concisely and concretely the proposed PSM-level static scheduling approach, and the potential for performance improvement using the approach.

Figure 5.3: Execution time comparison between canonical scheduling and PSM-level static scheduling for the synthetic example of Figure 5.2.

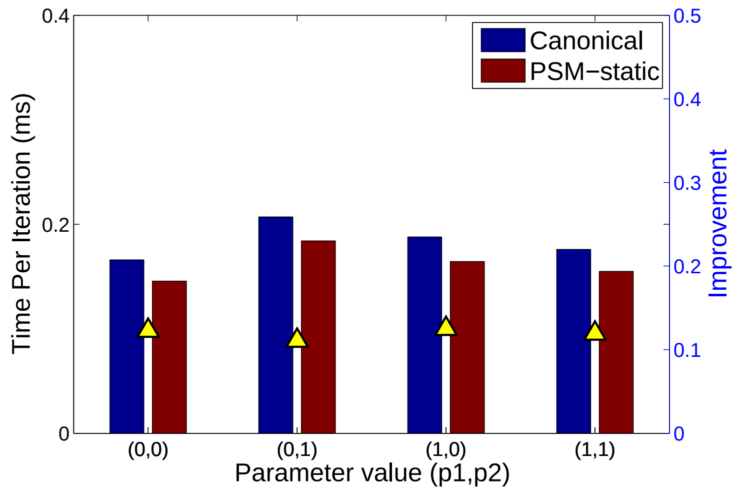


Table 5.3: Dynamic actors in the rqam application.

Actor	Mode	Prod	Cons
	INIT	0	1
src	src_1	1	0
	src_2	2	0
	INIT	0	(0,1)
T	*BPSK*	1	(1,0)
	QPSK	1	(2,0)
	$16 - QAM$	1	(4,0)

5.4.5 Application Example

In this section, we demonstrate a practical example of PSM-level static scheduling that is relevant to the cognitive radio domain. Figure 5.4 shows a dynamically configurable modulator that supports multiple source rates and multiple Phase-Shift-Keying (PSK) and Quadrature Amplitude Modulation schemes. The hierarchical actor R encompasses a subgraph that contains two CFDF actors *src* (using a minor abuse of notation), and T, whose modes are shown in Table 5.3. Here, r and m specify the source rate and the modulation scheme, respectively.

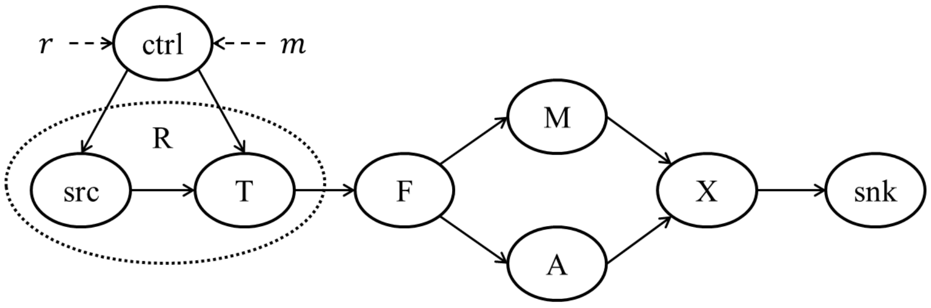

r
ctrl
m
R
src
T
F
M
A
X
snk
src: multi-rate source T: lookup-table F: duplicate
M: magnitude, A: Angle X: muliplier snk: data sink ctrl: control

Figure 5.4: A dynamically configurable modulator in CFDF.

Table 5.4: PSMs of the hierarchical actor R in the rqam application.

PSM	Mode of R	Mode of *src T*	**q**
PSM$_1$	R_0	*INIT*,*INIT*	(1,1)
	R_1	*src$_1$*, *BPSK*	(1,1)
	R_2	*src$_2$*, *QPSK*	(1,1)
PSM$_2$	R_3	*src$_1$ QPSK*	(2,1)
	R_4	*src$_2$*, $16-QAM$	(2,1)
PSM$_3$	R_5	*src$_1$* $16-QAM$	(4,1)
PSM$_4$	R_6	*src$_2$ BPSK*	(1,2)

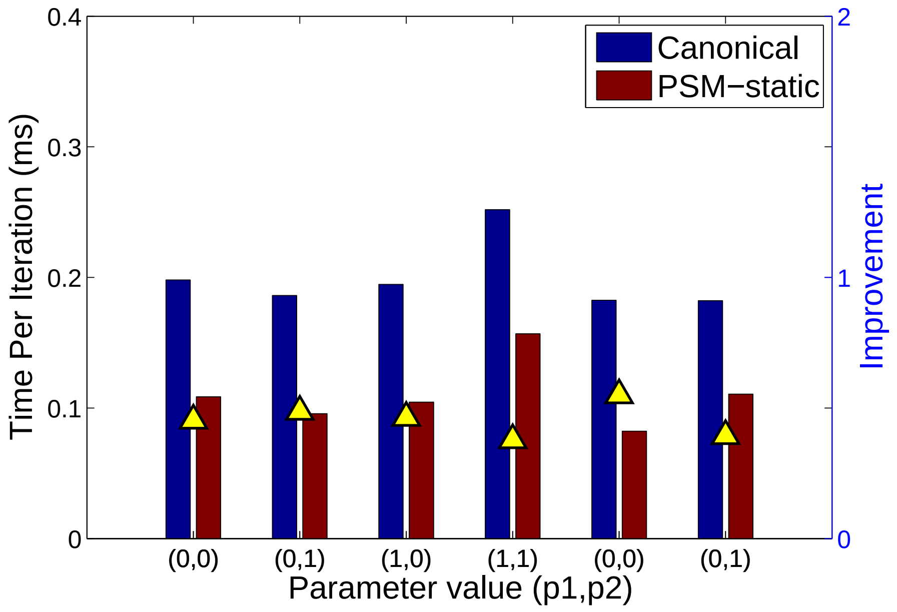


Figure 5.5: Execution time comparison between canonical scheduling and PSM-level static scheduling for the rqam application.

Using PSM-level static scheduling, we derive 4 PSMs, as shown in Table 5.4. The static schedule for each PSM is then constructed by hand, implemented in LIDE, and compared with canonical scheduling, as in Section 5.4.4. We see from the results that in this example, the performance improvement from applying PSM-level static scheduling is higher compared to that of the small, synthetic example in Section 5.4.4. In terms of the execution time per graph iteration (i.e., per minimal periodic scheduling iteration of the derived SDF subgraphs), PSM-level static scheduling outperforms canonical scheduling by an average of 45.4%, as shown in Figure 5.5. Here, the average is taken across the 6 operational modes for the hierarchical actor R.

5.4.6 Summary of PSM-level Static Scheduling

In this part of chapter, we have demonstrated a specific method, called PSM-level static scheduling, for applying the PSM modeling approach. There are many possible ways of applying PSMs in the design process, and the method presented in this section can be viewed as a specific way that we have studied and experimented with to help validate the utility of the PSM model. Although the PSM-level static schedules experimented with in this section were constructed by hand, their foundation in the PSM and CFDF formalisms makes them amenable to derivation through general, automated techniques. Development of such automated tool support for PSM-level static scheduling and other applications of PSMs is a useful direction for further investigation.

5.5 PSM-level Processor Selection for Heterogeneous Platforms

In this section, we demonstrate the application of PSMs to mapping actors in a CFDF-based dataflow program onto a heterogeneous platform. The targeted platform here consists of a general purpose CPU (called "host"), and a graphics processing unit (GPU) that is used to accelerate selected actors. The GPU is controlled by the host, and has a separate memory address space.

5.5.1 Overview

The execution of an actor in this environment on the GPU device generally involves three steps: host-to-device data transfer, on-device execution, and device-to-host data transfer. The data transfers between processors can result in significant overhead, which makes it unfavorable in some scenarios, such as when the amount of data to be processed is relatively small. Thus, the selection of actors to execute on the GPU (processor assignment) is an important problem for performance optimization.

We first formulate a general version of the processor assignment problem that is addressed in this section, and we describe our PSM-level processor selection approach in this general context. Then we present experimental results for PSM-level processor selection on the specific CPU-GPU heterogeneous platform described above.

5.5.2 PSM-level Processor Selection

Suppose that we have a CFDF graph $G = (V, E)$, and a target platform consisting of a (possibly heterogeneous) processor set $P = \{p_1, p_2, \ldots, p_n\}$. Also, for an actor A in G, let M_A denote the set of CFDF modes of A. The objective of PSM-level processor selection is to derive a set of PSMs and a "top-level" quasi-static schedule with the goal of optimizing a pre-defined performance metric. More specifically, PSM-level processor selection involves the following tasks:

- for each actor A, derivation of a set of n PSMs,
  $\mathit{selection}(A) = \nu(A, 1), \nu(A, 2), \ldots \nu(A, n)$, where each $\nu(A, i)$ represents the subset of modes in M_A that are to be assigned (during graph execution) to processor p_i;

- construction of a "top-level", quasi-static schedule that executes actors in G based on the dynamically-determined processor assignment defined by $\{selection(A)\} \mid A \in V$ together with the current parameter values (actor configurations) of the actors in V.

In our development of PSM-level processor selection in the remainder of this section, our targeted performance metric is throughput. However, the proposed processor selection framework can be readily targeted to other metrics, such as latency or memory utilization or to composite metrics, such as latency-constrained throughput optimization, and memory-constrained latency optimization.

5.5.3 Profile-based Selection

In this section, we develop a profile-driven approach to PSM-level processor selection. We refer to this approach as *profile- and PSM-based processor selection* (*PAPPS*). In PAPPS, a three-dimensional "profile table" is used to characterize the performance of specific actor modes on specific processors. In particular, for a given mode $m \in M_A$ for an actor A, and a given processor $p \in P$, $profile(A, m, p)$ provides an estimate of the execution time of mode m for actor A on processor p. The profile table entries for a given actor can be obtained, for example, by iteratively (e.g., through appropriate simulation scripts) executing the actor on each processor in every mode and averaging the results for each mode.

After the profile table is constructed, PSMs for each actor A are formed by grouping together modes that perform best on a specific processor with ties being broken arbitrarily. Thus, for each actor A and each $i \in \{1, 2, \ldots n\}$, we have that

$$\nu(A, i) = \bigcup \{\{m\} \mid i = \mathit{argmin}_j \mathit{profile}(A, m, p_j)\}. \tag{5.3}$$

In the PAPPS approach, ties with respect to the *argmin* function in Equation 5.3 are resolved arbitrarily (as implied earlier), although more sophisticated schemes can be envisioned that take ties or "near-ties" (multiple alternatives that have competitive performance) into account in strategic ways. Such exploration of more sophisticated PSM-based processor assignment schemes is an interesting direction for further work.

Once the PSMs are constructed based on Equation 5.3, a top-level, quasi-static scheduler is used to visit actors according to some scheduling policy, and to execute each visited actor A using a target processor that is (dynamically) selected based on the currently-active PSM for A. In other words, each time an actor A is visited by the scheduler, the current mode m of A is examined to determine the active PSM (i.e., the unique $\nu(A, i)$ that contains m), and then processor p_i is selected as the processor on which to execute the next firing of A.

Canonical scheduling, described in Section 5.4, is a general policy that can be used as the top-level scheduling policy in this context. However, in some cases, static analysis of the parameterized application structure can be applied to streamline the policy — for example, by statically fixing the order of schedule traversal in a way that eliminates or greatly reduces the need for run-time enable condition checking. We demonstrate a simple example of such static-analysis-based streamlining in

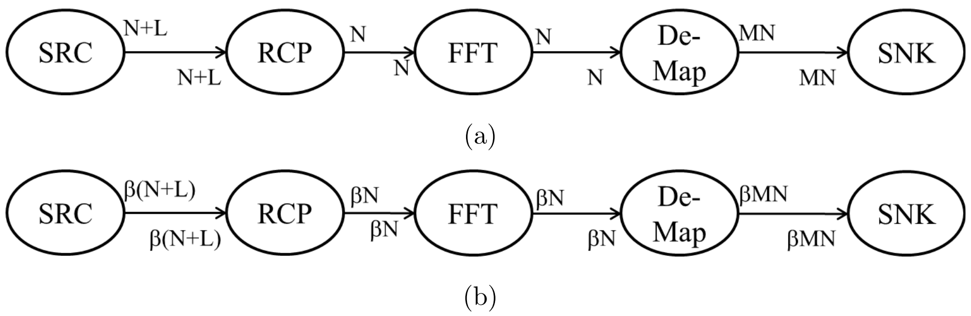


Figure 5.6: PSM-CFDF model of a configurable OFDM demodulator. (a) Original dataflow graph. (b) Vectorized dataflow graph.

Section 5.5.4.

5.5.4 OFDM Demodulation

To demonstrate the PAPPS approach, we have applied it to an OFDM demodulator and a heterogeneous CPU/GPU implementation platform, as described in Section 5.5.2. Orthogonal frequency division multiplexing (OFDM) is used extensively in high-speed wireless communication systems because of its spectral efficiency, robustness in terms of multi-path propagation, and high bandwidth efficiency [65]. The OFDM demodulator is one of the fundamental subsystems of LTE and WiMAX wireless communication systems.

Figure 5.6 illustrates a runtime-reconfigurable OFDM demodulator that is modeled as a CFDF graph. Here, actor *SRC* represents a data source that generates random values to simulate a sampler. In a wideband OFDM system, information is encoded on a large number of carrier frequencies, forming an OFDM symbol stream. In baseband processing, a symbol stream can be viewed in terms of consecutive vectors of length N. The symbol is usually padded with a cyclic prefix (CP) of length

L to reduce inter-symbol inference (ISI) [66]. In Figure 5.6, the CP is removed by actor *RCP*. Then, actor *FFT* performs a fast Fourier transform (FFT) to convert the symbol stream to the frequency domain.

In practical systems, further processing, such as frequency domain synchronization and channel estimation, is required to remove various channel effects. In this case study, however, we use a simpler design that directly performs symbol demapping to illustrate the PAPPS methodology. Actor *Demap* is a parameterized symbol demapper that performs M-ary QAM demodulation, with a configurable QPSK configuration ($M = 2$ or $M = 4$). The output bits are collected by the data sink (actor *SNK*).

For the targeted CPU/GPU platform described in Section 5.5.1, all of the actors in our OFDM demodulation system have CPU implementations, and some of the actors have GPU implementations.

Each actor A has a parameter, called the vectorization degree and denoted by $\beta(A)$, which is the number of OFDM symbols to be processed in a single activation (scheduler visit) of the actor. If the actor A is understood from context, then we sometimes drop the "(A)" and simply write β. Vectorization of signal processing dataflow graph actors, also referred to as "block processing", is useful in optimizing throughput, which is the targeted objective in our development of PSM-level processor selection (see Section 5.5.2) [34]. Here we assume that the same demapping scheme can be applied to all symbols to be processed in one activation, so that SIMD processing can be applied in vectorized executions.

In addition to β, actors in this design have a parameter M, which prescribes

Table 5.5: Actor parameters in the OFDM demodulator system.

Parameter	Domain
β	$\{1, 10, \ldots, 100\}$
N	$\{512, 1024\}$
M	$\{2, 4\}$

the number of bits per symbol. For example, if $M = 4$ and $\beta = 10$, this means that the system is operating in a mode that uses QAM16 as the demapping scheme, and executes actors in blocks of 10 firings each. A third actor parameter is the OFDM symbol length, which we denote by N.

The parameter values in this example determine the mode of each actor, and the actor mode determines the production and consumption rates. Note that this is not always the case in CFDF actors, where, for example, the next mode for an actor can be different from the current mode even though there is no change in parameter settings (e.g., see [8]). However, because there is no such dynamics involved with next mode determination in this example, the actors can be mapped into corresponding parameterized synchronous dataflow (PSDF) actors [54]. The example, therefore demonstrates the applicability to PSM techniques to PSDF graphs.

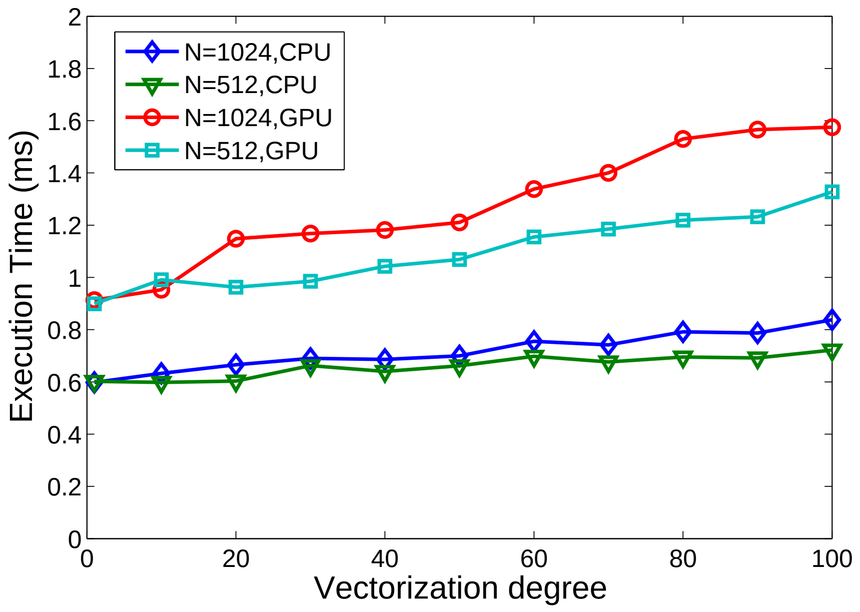
Table 5.5 shows the valid parameter values for the actors in our OFDM demodulation system. The mode set of *Demap* is given by Equation 5.2 in Section 5.3.6. Similarly, for other actors, valid combinations of parameter values lead uniquely to their mode settings. These details for the other actors are omitted here for brevity.

5.5.5 Application of PAPPS to the OFDM Demodulation System

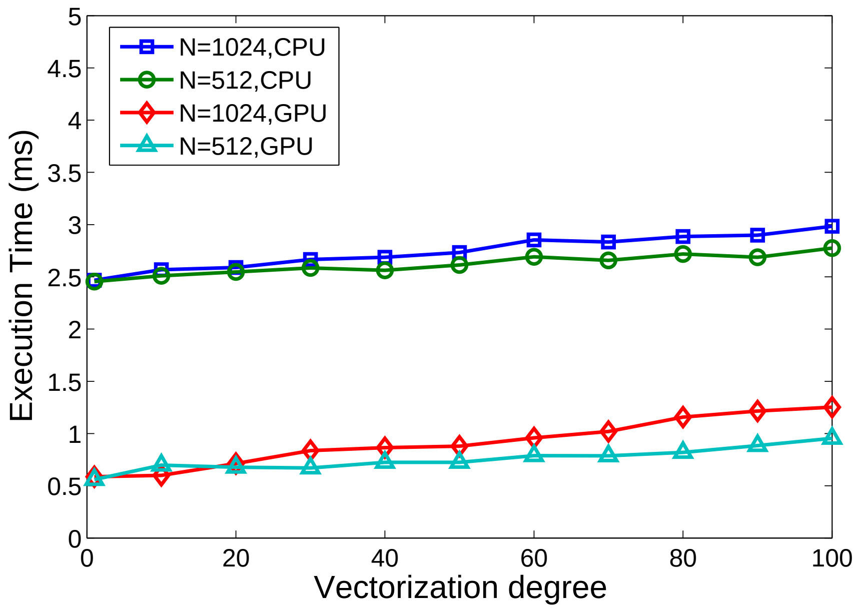
The PSM-CFDF actors *RCP*, *FFT* and *Demap* are each implemented on both the CPU and GPU processors. Following the profiling approach described in Section 5.5.3, each actor A is profiled in every mode in its mode set M_A for both the CPU and GPU implementations. The results are then used to construct the profile table *profile*.

In our experiments, an NVIDIA GTX680 GPU with 2GB memory and an Intel Core I7 3.4GHz CPU with 8GB memory are used for GPU implementation and CPU implementation, respectively. Figure 5.7 illustrates the profile table *profile* for the actors. The maximum latency for all vectorization degrees considered is less than 8 ms, which is tolerable in many software defined radio contexts. In the case of *RCP*, which removes the cyclic prefix from the received signal, the CPU implementation performs better in all settings. This is due to the small amount of computation performed in this actor compared to the large CPU-GPU memory transfer overhead. As a result, *selection*(RCP) contains only one non-empty PSM; the PSM associated with the GPU has no modes.

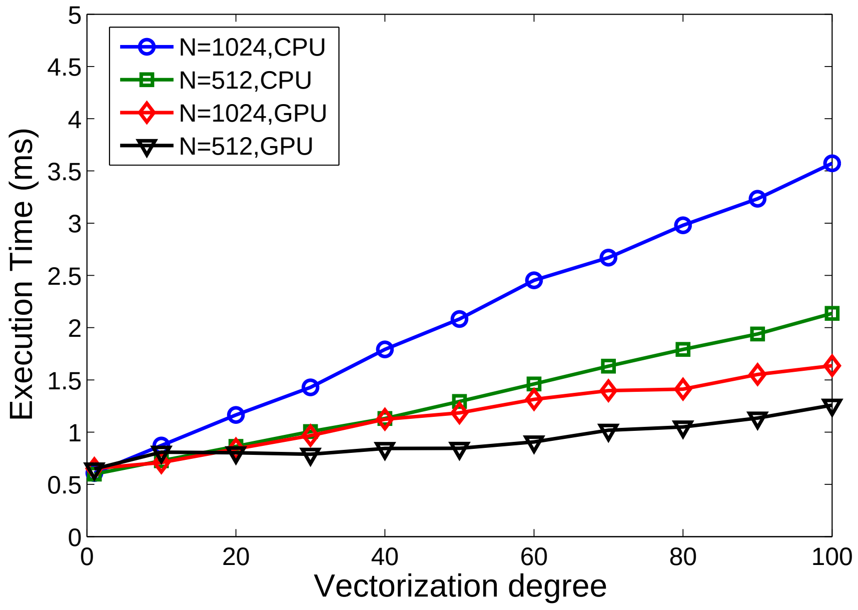
For the *FFT* actor, the GPU implementation always performs better than the CPU implementation in the same mode. Thus, for this actor, the PSM associated with the CPU has no modes. For the *Demap* actor in the 16-QAM modes ($M = 4$), the GPU implementation outperforms the CPU implementation for all values of the vectorization degree β. In the QPSK modes ($M = 2$), there is less difference in performance, and the CPU implementation generally performs better for lower β values, while the GPU implementation performs better for higher β values. The smaller computational load in the QPSK modes makes the memory transfer overhead more significant, which leads to a smaller performance gain from the GPU. In summary, the *Demap* actor has two non-empty PSMs $\nu(Demap, p_1)$ and $\nu(Demap, p_2)$.


N=1024,CPU
N=512,CPU
N=1024,GPU
N=512,GPU
Execution Time (ms)
Vectorization degree
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
0
20
40
60
80
100

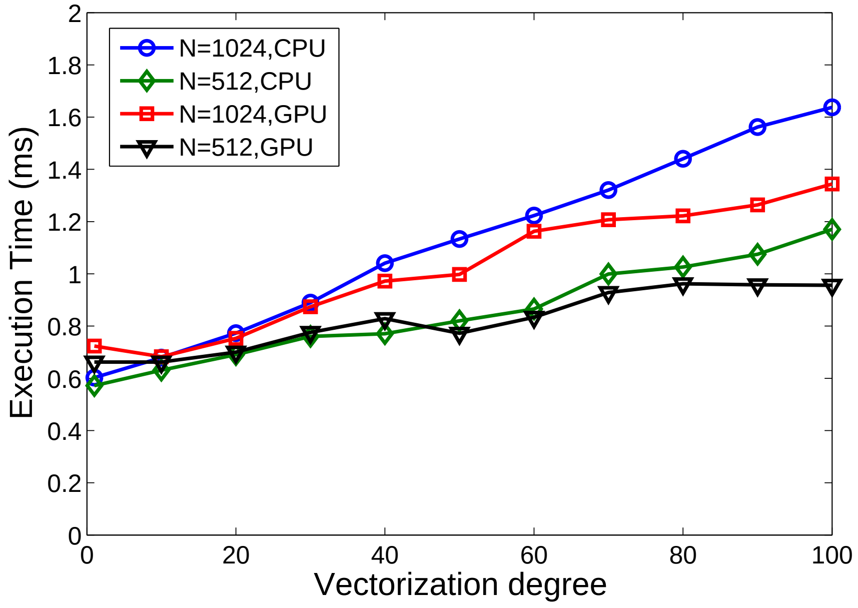
(a)


N=1024,CPU
N=512,CPU
N=1024,GPU
N=512,GPU
Execution Time (ms)
Vectorization degree
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
20
40
60
80
100

(b)



(c)



(d)

Figure 5.7: Actor profiles for application of PAPPS to the OFDM demodulator: (a) *RCP* actor; (b) *FFT* actor; (c) *Demap* actor in 16-QAM modes; (d) *Demap* actor in QPSK modes.

Table 5.6 shows the grouping of actor modes into PSMs when applying the

Table 5.6: PSM grouping based on CPU and GPU performance profiles for processor selection. PSM_1 and PSM_2 are the sets of modes that have shorter execution times for CPU- and GPU-based execution, respectively.

Actor	PSM_1	PSM_2
RCP	$N = 512, 1024;\ 1 \leq \beta \leq 100$	$\varnothing$
FFT	$\varnothing$	$N = 512, 1024;\ 1 \leq \beta \leq 100$
$Demap$	$\{N = 1024, M = 4,\ \beta = 1\}$	$\{N = 1024, M = 4,\ 10 \leq \beta \leq 100\}$
	$\{N = 512, M = 4,\ \beta = 1, 10\}$	$\{N = 512, M = 4,\ 20 \leq \beta \leq 100\}$
	$\{N = 1024, M = 2,\ \beta = 1, 10\}$	$\{N = 1024, M = 2,\ 20 \leq \beta \leq 100\}$
	$\{N = 512, M = 2,\ 1 \leq \beta \leq 40\}$	$\{N = 512, M = 2,\ 50 \leq \beta \leq 100\}$

PAPPS method based on the achieved profiling results illustrated in Figure 5.7.

We have implemented the OFDM demodulator system on the targeted CPU/GPU platform using a PAPPS-based processor selection scheme based on the PSMs illustrated in Table 5.6. We streamlined the top-level scheduler (see Section 5.5.3) by observing that even though the production and consumption rates of actors can vary based on the active actor modes, the variations in this application are interdependent such that the dataflow graph exhibits SDF behavior, and furthermore, the repetitions vector remains constant. In particular, the repetitions vector is specified by $q(A) = 1$ for each actor A regardless of what actor modes are active. This allows us to implement the top-level scheduler without any run-time checks for actor enabling conditions. Note, however, that even though SDF techniques are employed, the derived scheduler should not be viewed as a form of static scheduling because the processor assignment can change dynamically.

As in the case study of Section 5.4, we implemented the top-level scheduler by hand. This scheduler implementation incorporates the PAPPS method for dynamic processor selection based on the PSM decompositions illustrated in Table 5.6. Building on the developments of this section to construct automated scheduler derivation for PAPPS-based implementation is an interesting direction for future work.

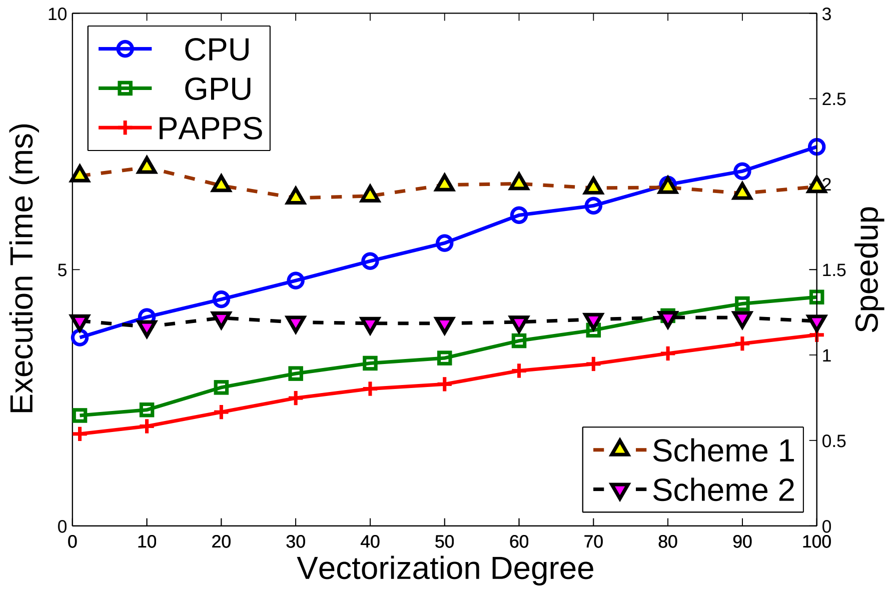
5.5.6 Experimental Results

We compared the application throughput of alternative implementations in terms of the execution time per (vectorized) application iteration, where an application iteration in this context corresponds to the processing required for $(\beta \times N)$ symbols of the enclosing OFDM system. Because we compare alternative processor selection schemes with β fixed for each comparison point, this method of throughput comparison does not favor any particular kind of scheme.

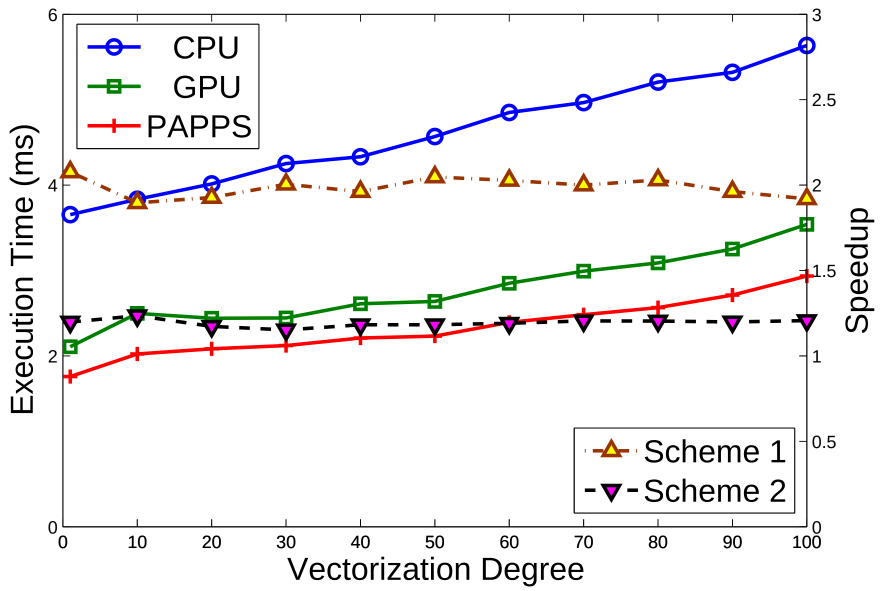
Figure 5.8 shows the execution time per application iteration for three types of processor selection schemes: (1) all actors are assigned to the CPU ("CPU"), (2) *RCP*, *FFT* and *Demap*, the most computationally-

intensive actors, are assigned to the GPU ("GPU"), and (3) processor selection is performed dynamically using our implemented PAPPS-based scheduler ("PAPPS"). Solid lines represent execution times while dashed lines represent the speedup obtained by using the PAPPS approach. The brown dashed line with an "up-triangle" represents the speedup of PAPPS over CPU (scheme (1)); the black dashed line represents the speedup of PAPPS over GPU (scheme (2)). The speedups achieved by using PAPPS, compared to methods (1) and (2), are also shown in the figure. The average speedup achieved by PAPPS in this application over a CPU implementation is more than 1.5X. In the setting where the largest amount of data is present

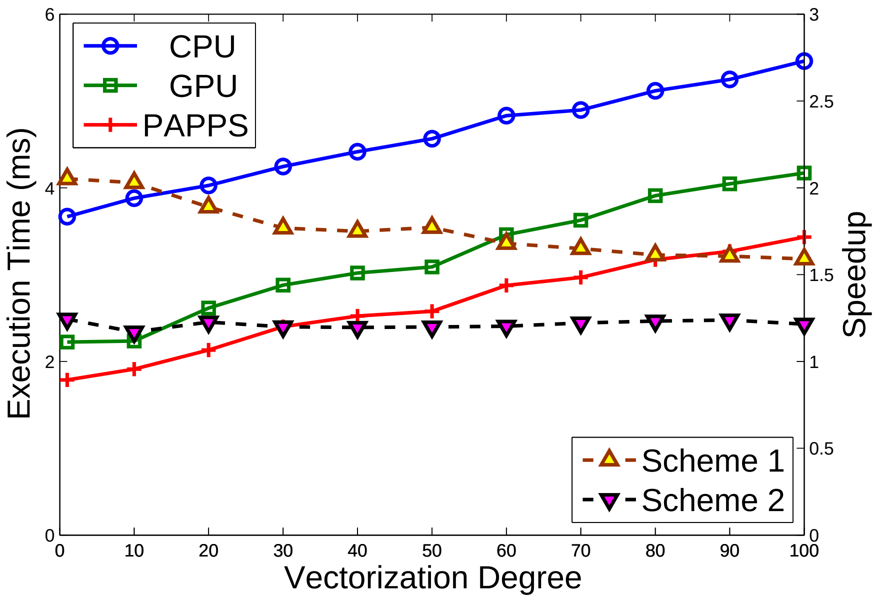
(1024-FFT and 16-QAM), the average speedup is more than 2X over all vectorization degrees. The achieved speedup is limited by the cost of data transfer between CPU and GPU memory for each actor. This data transfer overhead has been taken into account in the reported speedup values.


CPU
GPU
PAPPS
Scheme 1
Scheme 2
Execution Time (ms)
Speedup
Vectorization Degree
0
5
10
0
0.5
1
1.5
2
2.5
3
0
10
20
30
40
50
60
70
80
90
100

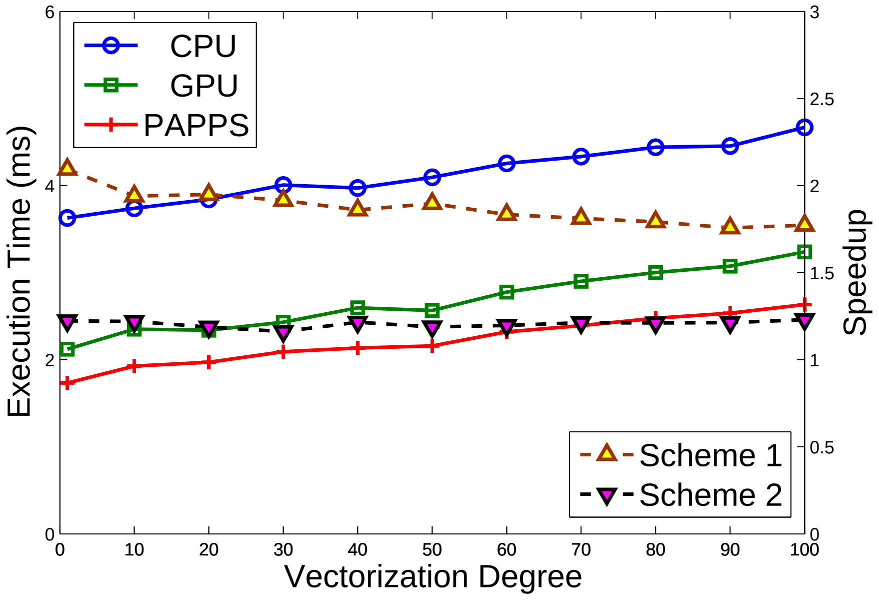
(a)


CPU
GPU
PAPPS
Scheme 1
Scheme 2
Execution Time (ms)
Speedup
Vectorization Degree
0
2
4
6
0
0.5
1
1.5
2
2.5
3
0
10
20
30
40
50
60
70
80
90
100

(b)



(c)



(d)

Figure 5.8: Execution time and speedup under three types of processor selection schemes for the OFDM demodulator system. (a) 1024-pt FFT, 16-QAM; (b) 512-pt FFT, 16-QAM; (c) 1024-pt FFT, QPSK; (d) 512-pt FFT, QPSK.

Compared to the GPU implementation scheme (scheme (1)), the PAPPS scheme achieves an average of 20% improvement in throughput over the GPU scheme. However, the vectorization step applied in our implementation generally results in increased latency for the system. In wireless communication applications, latency is a critical design constraint (e.g. see [67]), and thus, vectorization should be applied carefully to ensure that excessive latency does not result.

In our experiments, the vectorization degree is set to be no more than 100. As shown in Figure 5.8, this results in a maximum latency of 8ms, which is reached when $N = 1024$ and $\beta = 100$. This is at a tolerable level of latency for many kinds of software radio systems. For example, 8ms is only a small fraction of the typical 250ms end-to-end delay for data packets, which is described for the communication systems discussed in [68]. In cases where there are more stringent latency constraints, the vectorization degree can be bounded more tightly to trade off throughput performance for decreased latency.

The experiments presented in this section along with the other examples discussed in this chapter are provided to give a concrete idea of the kind of approaches that are supported by the PSM framework. These can be viewed as representative examples that help to give a sense of the diverse possibilities for applying the proposed methods. Further study into applying these methods and developing design optimizations that build on them is a useful direction for future investigation.

5.6 Summary

In this chapter, we have introduced a new dataflow modeling technique called parameterized set of modes (PSM) and demonstrated its relevance and application to design and implementation of signal processing systems for cognitive radio applications. PSMs enable novel ways for representing, manipulating and applying related groups of actor modes that lead to more concise formulations of actor behavior, and a unified modeling methodology for applying a variety of techniques for efficient implementation. To demonstrate the utility and versatility of PSMs in signal processing system design processes, we have developed two case studies involving mapping of important kinds of reconfigurable wireless communication subsystems into efficient implementations. The PSM methods introduced in this chapter allow implementation techniques like those introduced in the case studies to be developed according to a common modeling framework, which allows such techniques to be better understood, integrated, and optimized. Several useful directions for future work have also emerged from the developments of this chapter, including the investigation of automated techniques for applying PSMs to efficient static region derivation and to processor selection on heterogeneous platforms.

Chapter 6

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.

6.1 Conclusions

In this thesis, we have developed novel dataflow modeling, scheduling and vectorization techniques that are geared towards high-performance software synthesis for hybrid CPU-GPU computing platforms. Our contributions are summarized into three major parts listed as follows.

Firstly, we have developed a new model-based software synthesis framework, called DIF-GPU, that integrates high level dataflow graph specification, vectorization, scheduling, and code generation for heterogeneous CPU-GPU platforms. We have demonstrated the ability of DIF-GPU to synthesize, through its highly integrated design flow, implementations that significantly outperform conventional CPU-GPU mappings (i.e., where all actors for which GPU implementations are available are unconditionally mapped to the GPU). Furthermore, we have demonstrated the utility of DIF-GPU in (a) enhancing application performance through optimized management of interprocessor communication for given scheduling and vectorization configurations, and (b) exploring complex design spaces in the mapping of applications onto CPU-GPU platforms.

Secondly, we have investigated memory-constrained, throughput optimization for synchronous dataflow (SDF) graphs on heterogeneous CPU-GPU platforms. We have developed novel methods for Integrated Vectorization and Scheduling (IVS) that provide throughput- and memory-efficient implementations on the targeted class of platforms. We have integrated these IVS methods into the DIF-GPU Framework, which provides capabilities for automated synthesis of GPU software from high-level dataflow graphs specified using the dataflow interchange format (DIF). Our development of novel IVS methods and their integration into DIF-GPU provide a streamlined workflow for automated exploitation of pipeline, data and task level parallelism from SDF graphs. We have demonstrated our IVS methods through extensive experiments involving a large collection of diverse, synthetic SDF graphs, as well as on a practical embedded signal processing case study involving a wireless communications receiver that is based on orthogonal frequency division multiplexing. The results of our experiments demonstrate that our proposed new methods for IVS provide significant improvements in system throughput when mapping SDF graphs onto CPU-GPU platforms.

Finally, we have introduced a new dataflow modeling technique called parameterized set of modes (PSM) and demonstrated its relevance and application to design and implementation of signal processing systems for cognitive radio applications. PSMs enable novel ways for representing, manipulating and applying related groups of actor modes that lead to more concise formulations of actor behavior, and a unified modeling methodology for applying a variety of techniques for efficient implementation. To demonstrate the utility and versatility of PSMs in signal processing system design processes, we have developed two case studies involving mapping of important kinds of reconfigurable wireless communication subsystems into efficient implementations. The PSM methods introduced in this thesis allow implementation techniques like those introduced in the case studies to be developed according to a common modeling framework, which allows such techniques to be better understood, integrated, and optimized.

6.2 Future Work

Our PSM- and DIF-GPU-based methods target design and optimization for next-generation wireless communication systems from two different aspects: modeling flexibility and exploitation of parallelism. The modeling and optimization techniques developed in this research currently support the core functional dataflow (CFDF) and synchronous dataflow (SDF) models of computation.

In the scope of this thesis, the concept of parameterized sets of modes (PSMs) is applied to achieve performance improvement for restricted classes of multiprocessor platforms. Our approach to PSM-level static scheduling is targeted to efficient scheduling on single-processor architectures, while our approach to PSM-level processor selection is targeted to architectures that are composed of one multi-core CPU and one GPU.

Compared to our development of PSM-based design methods, DIF-GPU naturally supports a broader range of heterogeneous computing platforms, while it

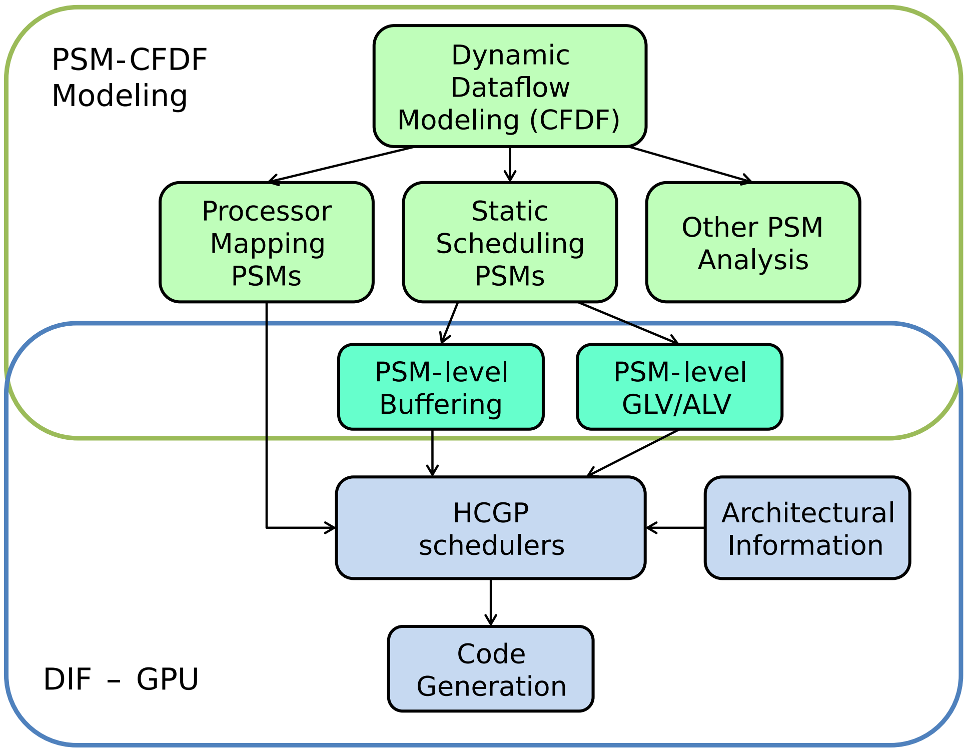


Figure 6.1: Architecture for integration of PSM modeling and optimized software synthesis using DIF-GPU.

assumes a more restricted model of computation — SDF – for application design.

A useful direction for future work therefore centers on integration of the flexible and compact modeling provided by PSMs and the extensive capabilities for software synthesis and design optimization that are provided by DIF-GPU. Figure 6.1 represents a possible architecture for an extended framework that integrates both dynamic dataflow modeling and multiprocessor signal processing performance optimizations in this manner. Here, ALV and GLV stand for actor-level vectorization and graph-level vectorization, respectively.

Two key problems involved in developing this envisioned new framework are the following:

- PSM-level buffering. Developing effective, automated methods for PSM-level buffer analysis is important for providing efficient memory management when integrating PSM models into the DIF-GPU design flow.

- PSM-level vectorization. PSMs for dataflow graph actors represent a lower level of abstraction compared to actors — a single actor can encapsulate any number of PSMs. Extending concepts and methods of actor-level vectorization to the level of PSMs may be a promising direction to derive vectorization methods for dynamic dataflow models that are represented using PSM- and CFDF-based techniques.

The design methods and tools centering on PSMs and DIF-GPU developed in this thesis have laid a foundation for investigation into such new directions for synthesis and optimization from dynamic dataflow representations. For example, as described in Chapter 5.4.2, PSM-level static scheduling groups hierarchical actor modes into PSMs that have the same repetition vectors associated with their encapsulated subgraphs. This property allows the hierarchical actors to be vectorized in the same way for each mode in the corresponding PSMs, which can in turn be translated into performing graph-level vectorization on the encapsulated subgraphs. As another example, consider the PSM-level processor selection scheme (PAPPS) that was introduced in Chapter 5. This scheme can provide designers with important information about algorithm-to-architecture mapping decisions. A possible application of this information is to generate efficient initial mappings, which can then be further refined by more specialized dynamic schedulers. For example, such "back-end schedulers" may incrementally adapt schedules as run-time data is collected and analyzed about their performance and bottlenecks.

We envision that the development of PSM-related methods for efficient vectorization and scheduling on CPU-GPU platforms can significantly improve parallelization of dynamic dataflow models for this important class of platforms. For example, advances in this area can contribute to multidimensional design optimization of dynamic, data driven application systems (DDDAS) — e.g., see [69, 70].

Bibliography

[1] Erik G Larsson, Ove Edfors, Fredrik Tufvesson, and Thomas L Marzetta, "Massive mimo for next generation wireless systems," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 186–195, 2014.

[2] Cenk M Yetis, Tiangao Gou, Syed A Jafar, and Ahmet H Kayran, "On feasibility of interference alignment in mimo interference networks," *IEEE Transactions on Signal Processing*, vol. 58, no. 9, pp. 4771–4782, 2010.

[3] Mohsen Nader Tehrani, Murat Uysal, and Halim Yanikomeroglu, "Device-to-device communication in 5g cellular networks: challenges, solutions, and future directions," *IEEE Communications Magazine*, vol. 52, no. 5, pp. 86–92, 2014.

[4] Boyd Bangerter, Shilpa Talwar, Reza Arefi, and Ken Stewart, "Networks and devices for the 5g era," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 90–96, 2014.

[5] Shanzhi Chen and Jian Zhao, "The requirements, challenges, and technologies for 5g of terrestrial mobile telecommunication," *IEEE Communications Magazine*, vol. 52, no. 5, pp. 36–43, 2014.

[6] Xuemin Hong, Jing Wang, Cheng-Xiang Wang, and Jianghong Shi, "Cognitive radio in 5g: a perspective on energy-spectral efficiency trade-off," *IEEE Communications Magazine*, vol. 52, no. 7, pp. 46–53, 2014.

[7] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, second edition, 2013, ISBN: 978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online).

[8] 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.

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

[10] *CUDA C Programming Guide*, September 2015, Version 7.5.

[11] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*, Addison-Wesley, 2011.

[12] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013.

[13] Pierre-Henri Horrein, Christine Hennebert, and Frédéric Pétrot, “Integration of gpu computing in a software radio environment,” *Journal of Signal Processing Systems*, vol. 69, no. 1, pp. 55–65, 2012.

[14] C. Gregg and K. Hazelwood, “Where is the data? why you cannot debate CPU vs. GPU performance without the answer,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 134–144.

[15] 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.

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

[17] 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.

[18] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya, “Heterogeneous design in functional DIF,” in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2008, pp. 157–166.

[19] R. Gu, J. Janneck, M. Raulet, and S. S. Bhattacharyya, “Exploiting statically schedulable regions in dataflow programs,” in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Taipei, Taiwan, April 2009, pp. 565–568.

[20] Shuoxin Lin, Yanzhou Liu, William Plishker, and Shuvra S. Bhattacharyya, “A design framework for mapping vectorized synchronous dataflow graphs onto cpu-gpu platforms,” in *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, New York, NY, USA, 2016, SCOPES ’16, pp. 20–29, ACM.

[21] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya, “A lightweight dataflow approach for design and implementation of SDR systems,” in *Proceedings of the Wireless Innovation Conference and Product Exposition*, 2010.

[22] M. I. Gordon, W. Thies, and Saman Amarasinghe, “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs,” in *Symposium on Architectural Support for Programming Languages and Operating Systems*, 2006.

[23] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, “Software pipelined execution of stream programs on GPUs,” in *Proceedings of the International Symposium on Code Generation and Optimization*, 2009, pp. 200–209.

[24] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," in *Symposium on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 381–392.

[25] A. Hagiescu, H. P. Huynh, W.-F. Wong, and R. S. M. Goh, "Automated architecture-aware mapping of streaming applications onto GPUs," in *Proceedings of the International Symposium on Parallel and Distributed Processing*, 2011, pp. 467–478.

[26] G. Zaki, W. Plishker, S. S. Bhattacharyya, C. Clancy, and J. Kuykendall, "Integration of dataflow-based heterogeneous multiprocessor scheduling techniques in GNU radio," *Journal of Signal Processing Systems*, vol. 70, no. 2, pp. 177–191, February 2013, DOI:10.1007/s11265-012-0696-0.

[27] F. Ciccozzi, "Automatic synthesis of heterogeneous CPU-GPU embedded applications from a UML profile," in *Proceedings of the International Workshop on Model Based Architecting and Construction of Embedded Systems*, 2013.

[28] H. Jung, Y. Yi, and S. Ha, "Automatic CUDA code synthesis framework for multicore CPU and GPU architectures," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., vol. 7203 of *Lecture Notes in Computer Science*, pp. 579–588. Springer Berlin Heidelberg, 2012.

[29] L. Schor, A. Tretter, T. Scherer, and L. Thiele, "Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL," in *Proceedings of the IEEE Workshop on Embedded Systems for Real-Time Multimedia*, 2013, pp. 41–50.

[30] M. Goli, M. T. Garba, and H. González-Vélez, "Streaming dynamic coarse-grained CPU/GPU workloads with heterogeneous pipelines in FastFlow," in *Proceedings of HPCC-ICESS*, 2012, pp. 445–452.

[31] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, 2011.

[32] 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.

[33] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, "DIF: An interchange format for dataflow-based design tools," in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, 2004, pp. 423–432.

[34] S. Ritz, M. Pankert, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.

[35] C. Shen, L. Wang, I. Cho, S. Kim, S. Won, W. Plishker, and S. S. Bhattacharyya, "The DSPCAD lightweight dataflow environment: Introduction to LIDE version 0.1," Tech. Rep. UMIACS-TR-2011-17, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011, http://hdl.handle.net/1903/12147.

[36] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, Inc., 1999.

[37] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, CRC Press, second edition, 2009, ISBN:1420048015.

[38] G. Teodoro, R. Sachetto, O. Sertel, M. N. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira, "Coordinating the use of GPU and CPU for improving performance of compute intensive applications," in *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–10.

[39] Haluk Topcuoglu, Salim Hariri, and Min-you Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, 2002.

[40] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real time DSP," in *Proceedings of the Global Telecommunications Conference*, 1989, vol. 2, pp. 1279–1283.

[41] E. Blossom, "GNU Radio: tools for exploring the radio frequency spectrum," *Linux Journal*, June 2004.

[42] K. van der Veldt, R. van Nieuwpoort, A. L. Varbanescu, and C. Jesshope, "A polyphase filter for GPUs and multi-core processors," in *Proceedings of the Workshop on High-Performance Computing for Astronomy*, 2012, pp. 33–40.

[43] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi, "Throughput analysis of synchronous data flow graphs," in *Proceedings of the International Conference on Application of Concurrency to System Design*, June 2006.

[44] Jongsoo Park and William J Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 1–10.

[45] Yuankai Chen and Hai Zhou, "Buffer minimization in pipelined sdf scheduling on multi-core platforms," in *17th Asia and South Pacific Design Automation Conference*, Jan 2012, pp. 127–132.

[46] M. Ko, C. Shen, and S. S. Bhattacharyya, "Memory-constrained block processing for DSP software optimization," *Journal of Signal Processing Systems*, vol. 50, no. 2, pp. 163–177, February 2008.

[47] C. Hsu, J. Pino, and S. S. Bhattacharyya, "Multithreaded simulation for synchronous dataflow graphs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 16, no. 3, pp. 25–1–25–23, June 2011.

[48] W. Lund, S. Kanur, J. Ersfolk, L. Tsiopoulos, J. Lilius, J. Haldin, and U. Falk, "Execution of dataflow process networks on opencl platforms," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 618–625.

[49] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Journal of Concurrency and Computation: Practice & Experience*, vol. 23, no. 2, pp. 187–198, February 2011.

[50] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee, "Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 3, 2013.

[51] K. Desnos, M. Pelcat, J.-F. Nezan, and Slaheddine Aridhi, "Buffer merging technique for minimizing memory footprints of synchronous dataflow specifications," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2015, pp. 1111–1115.

[52] Robert P Dick, David L Rhodes, and Wayne Wolf, "Tgff: task graphs for free," in *Proceedings of the 6th international workshop on Hardware/software codesign*. IEEE Computer Society, 1998, pp. 97–101.

[53] Jackson W Massey, Jonathan Starr, Seogoo Lee, Dongwook Lee, Andreas Gerstlauer, and Robert W Heath, "Implementation of a real-time wireless interference alignment network," in *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*. IEEE, 2012, pp. 104–108.

[54] 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.

[55] S. Lin, L.-H. Wang, A. Vosoughi, J. R. Cavallaro, M. Juntti, J. Boutellier, O. Silvén, M. Valkama, and S. S. Bhattacharyya, "Parameterized sets of dataflow modes and their application to implementation of cognitive radio systems," *Journal of Signal Processing Systems*, vol. 80, no. 1, pp. 3–18, July 2015.

[56] W. Plishker, N. Sane, and S. S. Bhattacharyya, "Mode grouping for more effective generalized scheduling of dynamic dataflow applications," in *Proceedings of the Design Automation Conference*, San Francisco, July 2009, pp. 923–926.

[57] N. Sane, C.-J. Hsu, J. L. Pino, and S. S. Bhattacharyya, "Simulating dynamic communication systems using the core functional dataflow model," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Dallas, Texas, March 2010, pp. 1538–1541.

[58] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya, "A lightweight dataflow approach for design and implementation of SDR systems," in *Proceedings of the Wireless Innovation Conference and Product Exposition*, Washington DC, USA, November 2010, pp. 640–645.

[59] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich, "A SystemC-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. Article ID 47580, 22 pages, 2007.

[60] Jonathan Piat and Jeremie Crenne, "Modeling dynamic partial reconfiguration in the dataflow paradigm," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2014, pp. 1–6.

[61] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal, "Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications," in *Proceedings of the International Symposium on System-on-Chip*, 2011, pp. 14–21.

[62] J. Falk, C. Zebelein, C. Haubelt, and J. Teich, "A rule-based quasi-static scheduling approach for static islands in dynamic dataflow graphs," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 3, 2013.

[63] J. Eker and J. W. Janneck, "CAL language report, language version 1.0 — document edition 1," Tech. Rep. UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley, December 2003.

[64] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 21, no. 2, pp. 151–166, June 1999.

[65] O. Edfors, M. Sandell, J.-J. van de Beek, D. Landstrom, and F. Sjoberg, "An introduction to orthogonal frequency division multiplexing," Tech. Rep., Lulea University of Technology, Sweden, September 1996.

[66] J.-J. van de Beek, M. Sandell, M. Isaksson, and P. O. Borjesson, "Low-complex frame synchronization in OFDM systems," in *Proceedings of the International Conference on Universal Personal Communications*, 1995, pp. 982–986.

[67] Thomas Schmid, Oussama Sekkat, and Mani B Srivastava, "An experimental study of network performance impact of increased latency in software defined radios," in *Proceedings of the second ACM international workshop on Wireless network testbeds, experimental evaluation and characterization.* ACM, 2007, pp. 59–66.

[68] T. Blajić, D. Nogulić, and M. Družijanić, "Latency improvements in 3G long term evolution," in *Proceedings of the International Convention on Information and Communication Technology, Electronics and Microelectronics*, 2006.

[69] K. Sudusinghe, I. Cho, M. van der Schaar, and S. S. Bhattacharyya, "Model based design environment for data-driven embedded signal processing systems," in *Proceedings of the International Conference on Computational Science*, Cairns, Australia, June 2014, pp. 1193–1202.

[70] K. Sudusinghe, Y. Jiao, H. Ben Salem, M. van der Schaar, and S. S. Bhattacharyya, "Multiobjective design optimization in the lightweight dataflow for DDDAS environment (LiD4E)," in *Proceedings of the International Conference on Computational Science*, Reykjavik, Iceland, June 2015, pp. 2563–2572.