

ABSTRACT

Title of dissertation: **SCALABLE TECHNIQUES FOR SCHEDULING AND MAPPING DSP APPLICATIONS ONTO EMBEDDED MULTIPROCESSOR PLATFORMS**

George F. Zaki, Doctor of Philosophy, 2013

Dissertation directed by: **Shuvra S. Bhattacharyya (Chair/Advisor)**
Professor
Department of Electrical and Computer Engineering,
and Institute for Advanced Computer Studies

A variety of multiprocessor architectures has proliferated even for off-the-shelf computing platforms. To make use of these platforms, traditional implementation frameworks focus on implementing Digital Signal Processing (DSP) applications using special platform features to achieve high performance. However, due to the fast evolution of the underlying architectures, solution redevelopment is error prone and re-usability of existing solutions and libraries is limited. In this thesis, we facilitate an efficient migration of DSP systems to multiprocessor platforms while systematically leveraging previous investment in optimized library kernels using dataflow design frameworks. We make these library elements, which are typically tailored to specialized architectures, more amenable to extensive analysis and optimization using an efficient and systematic process.

In this thesis we provide techniques to allow such migration through four basic contributions:

1. We propose and develop a framework to explore efficient utilization of Single In-

struction Multiple Data (SIMD) cores and accelerators available in heterogeneous multiprocessor platforms consisting of General Purpose Processors (GPPs) and Graphics Processing Units (GPUs). We also propose new scheduling techniques by applying extensive block processing in conjunction with appropriate task mapping and task ordering methods that match efficiently with the underlying architecture. The approach gives the developer the ability to prototype a GPU-accelerated application and explore its design space efficiently and effectively.

2. We introduce the concept of Partial Expansion Graphs (PEGs) as an implementation model and associated class of scheduling strategies. PEGs are designed to help realize DSP systems in terms of forms and granularities of parallelism that are well matched to the given applications and targeted platforms. PEGs also facilitate derivation of both static and dynamic scheduling techniques, depending on the amount of variability in task execution times and other operating conditions. We show how to implement efficient PEG-based scheduling methods using real time operating systems, and to re-use pre-optimized libraries of DSP components within such implementations.
3. We develop new algorithms for scheduling and mapping systems implemented using PEGs. Collectively, these algorithms operate in three steps. First, the amount of data parallelism in the application graph is tuned systematically over many iterations to profit from the available cores in the target platform. Then a mapping algorithm that uses graph analysis is developed to distribute data and task parallel instances over different cores while trying to balance the load of all processing units

to make use of pipeline parallelism. Finally, we use a novel technique for performance evaluation by implementing the scheduler and a customizable solution on the programmable platform. This allows accurate fitness functions to be measured and used to drive runtime adaptation of schedules.

4. In addition to providing scheduling techniques for the mentioned applications and platforms, we also show how to integrate the resulting solution in the underlying environment. This is achieved by leveraging existing libraries and applying the GPP-GPU scheduling framework to augment a popular existing Software Defined Radio (SDR) development environment — GNU Radio — with a dataflow foundation and a stand-alone GPU-accelerated library. We also show how to realize the PEG model on real time operating system libraries, such as the Texas Instruments DSP/BIOS. A code generator that accepts a manual system designer solution as well as automatically configured solutions is provided to complete the design flow starting from application model to running system.

SCALABLE TECHNIQUES FOR SCHEDULING AND MAPPING
DSP APPLICATIONS ONTO EMBEDDED MULTICORE
PLATFORMS

by

George F. Zaki

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
2013

Advisory Committee:

Professor Shuvra S. Bhattacharyya, Chair/Advisor

Professor Steven Tretter

Professor Manoj Franklin

Professor Raj Shekhar

Professor Ramani Duraiswami

© Copyright by
George F. Zaki
2013

Dedication

To my family

Acknowledgments

I give sincere thanks to professor Shuvra Bhattacharyya, my main advisor, for his guidance, encouragement, and support. I am especially grateful for his experience and connections that gave me opportunities to work on applicable research projects, for the flexibility he gave me in choosing my research direction, and for his rigorous paper review. Professor Bhattacharyya also helped me in teaching by providing me opportunities to work as his teaching assistant and co-teacher. I learned from professor Bhattacharyya how an advisor can help his students have a fruitful as well as enjoyable PhD experience.

I am thankful to my PhD dissertation committee - professor Steven Tretter, professor Manoj Franklin, professor Raj Shekhar, and professor Ramani Duraiswami for reviewing this thesis and giving me valuable feedback.

I would like to thank Dr. Will Plishker, my Postdoc, for the productive discussions, detailed problem formulations, and hands-on assistance with my experiments.

I am thankful for the support of Texas Instruments Inc., which sponsored this research by allowing me to be a Texas Instruments scholar. Specifically I would like to thank Frank Fruth, Dr. Bogdan Kozanovic, Patel Piyush, and Charles Fosgate for giving me the opportunity for real-world experience and for providing me with the necessary tools, software, and development kits that I used to run experiments for this research.

I am so thankful to members of the Institute of Electronics and Telecommunications-Rennes (IETR), notably Dr. Maxime Pelcat for the fruitful discussions and Karol Desnos for sharing his experiences.

I thank the Laboratory for Telecommunications Sciences for their support, especial-

ly professor Charles Clancy, Tim Oshea, and Nicholas McCarthy.

I am also grateful to the members of the DSPCAD group - Dr. Chung-Ching Shen, Dr. Hojin Kee, Dr. Ruirui Gu, Dr. Nimish Sane, Dr. Hsiang-Huang Wu, Soujanya Kedilaya, Inkeun Cho, Kishan Sudusinghe, Shenpei Wu, Zheng Zhou, Ilya Chukhman, Lai-Huei Wang, Scott Kim, and Shuoxin Lin. The useful discussions we had helped me truly enjoy working in the DSPCAD laboratory.

I would like to give my sincere thanks and gratefulness to my parents and all the members of my family for their continuous sacrifice and encouragement throughout all my life.

Last but not least, I thank my wife Alma Jean for her support, help, patience, and for being my lovely partner during my last years as a doctoral student.

Table of Contents

List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Overview	1
1.2 Contributions	5
1.2.1 Scheduling DSP Systems on Heterogeneous Processors	5
1.2.2 Exposing Parallelism and Dynamic Scheduling Opportunities for DSP Applications	7
1.2.3 Scheduling and Mapping of Partial Expansion Graphs	8
1.2.4 Integration of Scheduling Solutions with Design Environments	10
1.3 Dissertation Organization	11
1.4 Summary	12
2 Background	13
2.1 Dataflow Applications Models	13
2.1.1 Formal Description	13
2.1.2 Dataflow Interchange Format	16
2.1.3 Streaming Application: Software Defined Radio	17
2.1.4 Pre-optimized Kernels: GNU Radio	18
2.2 Architecture Models	20
2.2.1 Contemporary Platforms	20
2.2.2 Model Elements	22
2.3 Multiprocessor Scheduling	24
2.4 Summary	26
3 Scheduling DSP Systems on Heterogeneous GPP and GPU Platforms	27
3.1 Workflow Description	27
3.1.1 Writing Accelerated Kernels	29
3.1.2 Partitioning, Scheduling, and Mapping	30
3.2 Multi-objective Multicore Scheduler	32
3.2.1 From Application Model to Block Processing DAGs	32
3.2.2 Architecture Model	34
3.2.3 Problem Formulation	35
3.3 Multiprocessor MLP Scheduler	37
3.3.1 Basic Variables	37
3.3.2 Constraints	39
3.3.3 Objective	41
3.4 Related Work and Contribution	42
3.5 Summary	44

4	Integration of the Heterogeneous Platform Scheduling Workflow with GNU Radio	45
4.1	FIR Filter Mapping to GPU Architecture	45
4.2	DIF Importer from GNU Radio	48
4.3	GRGPU: GPU Acceleration in GNU Radio	50
4.4	Empirical Results	53
4.4.1	GRGPU Profile	54
4.4.2	Scheduler Empirical Results	56
4.4.2.1	Test Setup	56
4.4.2.2	Design Space Exploration	57
4.5	Related Work and Contribution	60
4.6	Summary	61
5	Partial Expansion Graphs	63
5.1	Introduction	63
5.2	Partial Expansion of Dataflow Graphs	65
5.2.1	Formal Definition	66
5.3	Buffer Manager	68
5.3.1	Slot States	70
5.3.2	Slot Size Selection	72
5.4	Dynamic Scheduling	74
5.5	Code Generation	76
5.6	Evaluation	80
5.7	Related Work and Contribution	84
5.8	Summary	87
6	Scheduling and Mapping of Partial Expansion Graphs	89
6.1	PEG Scheduling	89
6.2	Particle Swarm Optimization	92
6.2.1	PSO Problem formulation	92
6.3	PEG Mapping Heuristic	95
6.4	Generic Implementation	99
6.5	Evaluation	102
6.5.1	Experimental Setup	102
6.5.2	Benchmarks	103
6.5.3	Results	105
6.6	Related Work and Contribution	108
6.7	Summary	110
7	Conclusions and Future Work	111
7.1	Static Systems	111
7.2	Dynamic Systems	114
	Bibliography	117

List of Figures

1.1	Design flow using a model based approach.	5
2.1	Example of a SDF graph.	15
2.2	Expressing parallelism using dataflow graphs.	16
2.3	GPU memory hierarchy.	22
2.4	A typical Texas Instruments evaluation board.	23
3.1	Example of an SDF graph for the mp-sched benchmark and its corresponding BPDAG.	34
3.2	Example of typical off-the-shelf platform consisting of GPPs and GPUs using the PCI bus as communication medium.	35
4.1	Implemented workflow for SDR applications described in GNU Radio. . .	46
4.2	Comparison of total running time versus number of outputs for FIR filter implementation on different processors.	47
4.3	Multirate application exported to DIF.	49
4.4	GRGPU actors within H2D and D2H communicate data using the GPU memory, avoiding unnecessary host/GPU transfers.	53
4.5	MP-sched SDR benchmark.	54
4.6	GRGPU overhead for various benchmarks.	56
4.7	Gantt chart for 2x5 mp-sched graph on 1 GPP and 1 GPU.	58
4.8	Design space for a 2x5 mp-sched graph on 1 GPP and 1 GPU for different blocking factors.	59

5.1	Expansion of a multirate SDF Graph.	65
5.2	Finite state machines for buffer slot states.	71
5.3	Updates of buffer states during graph execution.	73
5.4	PEG model evaluation benchmarks.	80
5.5	Speedups for different sources of parallelism using the PEG strategy. . . .	82
5.6	Efficiency of the PEG strategy for different computation to scheduling ratios.	82
5.7	Comparison between round robin and dynamic scheduling of activations. . . .	84
6.1	PEG-based scheduling workflow.	90
6.2	Input to the mapping heuristic.	97
6.3	Applying the mapping heuristic based on a given graph expansion.	98
6.4	Illustration of a customizable PEG-based implementation.	101
6.5	PEG scheduling evaluation benchmarks.	104

List of Tables

4.1	Solver results for different mp-sched graphs.	58
4.2	Evaluation of our MLP formulation on multicore processors.	60
5.1	Average mean and half confidence interval between the dynamic and round robin scheduling techniques.	84
6.1	PEG scheduling attributes for actors.	103
6.2	PEG Scheduling attributes for edges.	103
6.3	Speedups and expansion of the image registration benchmark	106
6.4	Speedups and expansion for the digital receiver benchmark.	106
6.5	Speedups and expansion for the SDR benchmark.	106
6.6	Mapping heuristic and system-in-the-loop overhead.	107

List of Algorithms

1	Implementation of peeking attribute on a shared memory architecture. . . .	72
2	FAFA dynamic scheduling heuristic	77
3	Particle swarm optimization	93
4	PEG mapping heuristic.	96

Chapter 1

Introduction

1.1 Overview

Implementation of streaming digital signal processing applications, such as Long Term Evolution (LTE), codecs for video and audio players, and Network Intrusion Detection systems (NIDs), has been constantly evolving over the past decades. Such implementation needs special attention to the requirement that these systems execute specific sets of repeated tasks over long periods of product lifetime. Spending time in exploring various design points can be important to derive low cost and efficient solutions that meet the system requirements. The development workflow of these systems starts by choosing appropriate DSP algorithms that achieve the desired system functionality. The algorithms are then examined for implementation, usually starting with some sort of block diagrams that represent the main system kernels and their interconnections. A target platform that is estimated to satisfy the final product specifications is then chosen and design space exploration is conducted.

Multiple platforms can be selected to implement such systems. Factors like time to market (i.e., ease of development), power consumption, system throughput, latency, and other metrics guide the selection of the final processing unit type or types (in the case of heterogeneous platforms). Typical platforms span the range from Application Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGA)s, Programmable

Digital Signal Processors (PDSPs), Graphics Processing Units (GPUs), and General Purpose Processors (GPPs).

Typically, numerous design decisions have to be taken in order to achieve efficient algorithm realizations that meet the given platform capabilities and constraints. For example, an efficient scheduling (mapping and ordering) of the given algorithm kernels to the available computing units is required. Depending on the target platform, scheduling objectives include different aspects to improve final performance metrics, such as latency, throughput, and power consumption, while meeting platform limitations. The scheduling problem is classically known to be NP complete [1]. In practice, designer experience, as well as different heuristics or time-consuming, exact algorithms are applied to derive scheduling solutions.

A wide range of digital signal processing applications are implemented on reprogrammable software processing units such as programmable digital signal processors, GPUs and GPPs. Building systems on these kinds of *off-the-shelf* processing units has the advantage of relatively shorter time to market, ease of development, code modification and debugging. Classical systems development for these platforms has targeted single core processors. Following Moore's law and the necessity to limit the dissipated power on a single chip, performance gains in these processing platforms has more recently been coming from increasing the number of cores on a single die instead of increasing the frequency of single core. This can effectively increase the computational horsepower on-chip while not adversely affecting power consumption. The complexity of a single core varies from Single Instruction Multiple Data (SIMD) units in GPUs, Very Long Instruction Word (VLIW) configurations in PDSPs, and more complex cores with branch

prediction and runtime speculation in GPPs accompanied with vector processing units.

Following this historical evolution of programmable platforms, developers of signal processing systems are now often required to migrate their libraries of kernels that were originally optimized to target single core processors to newer families of multicore processors [2, 3]. Such migration can be performed using two approaches. The first approach is to change the original serial algorithm for a kernel to a parallel one (e.g., from serial addition to parallel reduction). This method requires the designer to go through much of the development process for a new system and limits the ability to leverage previous code development investments. In order to simplify this operation for many multicore architectures, programming models and environments are being introduced to take advantage of particular processors types, along with their associated forms of memory hierarchy and communication facilities such as CUDA [4], OpenMP, MPI and OpenCL. The second approach, which is called the Model Based Approach (MBA), requires refinement of the original algorithm to a formal model in order to identify data dependencies between the kernels and different sources of parallelism. These sources can be categorized as data, task, and pipeline parallelism. Once this identification process is complete, the resulting application model can be analyzed and implemented either manually or using automated tools to take advantage of the target parallel platform. Many existing tools use MBA approaches, such as Ptolemy [5], StreamIt [6], PREESM [7], CAL [8], and DIF [9].

The first approach has the advantage of efficiently using all architecture features but with the penalty of redeveloping a large portion of system “from the ground up”. This operation has to be repeated with every new architecture. On the other hand, the MBA leverages much of the prior investment in many systems, such as in application

specific design frameworks where fast design flows are facilitated by exploiting the common application structures of particular domains and rich libraries of elements tailored to them. In this approach, required modifications are limited to kernel interface modifications needed to adhere to any new design models that are being applied, and to adaptations for scheduling techniques that are needed to handle new target platform characteristics.

Figure 1.1 shows a high level illustration of a model based design approach for signal processing systems. Here, a system designer starts by choosing appropriate models to represent the application and the targeted platform. These descriptions along with basic objectives and constraints are given as input to the scheduler. The scheduling solution is then passed to a code generator with some adjusted system attributes defined in *scenario files*. Once a design point is chosen and a working solution is implemented, profiling takes place, and then relevant parameters are tuned. Implementation, profiling, and parameter tuning can be iterated repeatedly until the final specifications are met or the designer decides to revisit higher level goals of the targeted implementation.

A major advantage of this kind of design approach is the separation of the application and platform representations. Such separation helps to preserve efforts used to model and implement different system kernels across different platforms. Also as an implementation is derived, various attributes that correspond to the application description, platform description or application-platform relationships can be described to allow the system designer to reason about, fine tune and iterate systematically over alternative solutions.

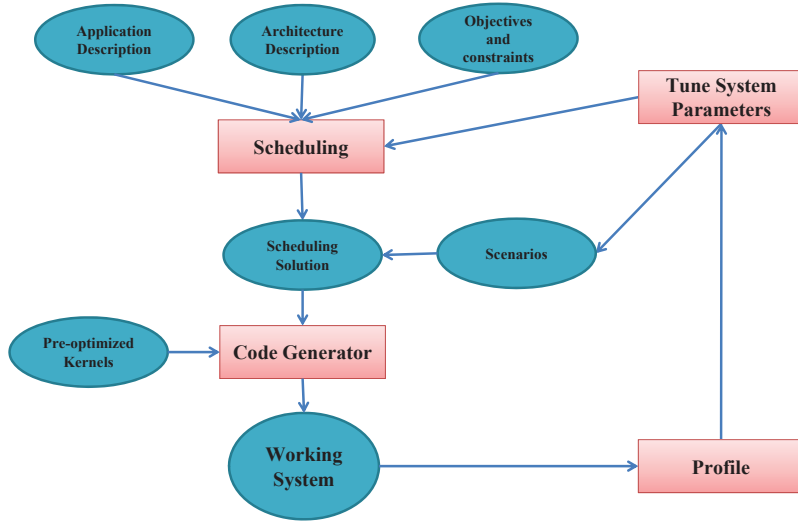


Figure 1.1: Design flow using a model based approach.

1.2 Contributions

In this dissertation, we develop new techniques to perform MBA-based migration of signal processing systems from single-core to multicore platforms. These techniques are developed in order to allow designers to derive efficient solutions on state-of-the-art platforms with high productivity. Starting from formal descriptions of the application and target platform, we investigate techniques and analysis tools to facilitate optimization and explore the design space when deriving an implementation. We demonstrate our new design methods concretely through the following four main contributions.

1.2.1 Scheduling DSP Systems on Heterogeneous Processors

GPUs have mainly been used for image processing and graphics applications that require single instruction multiple data processors. In recent years, they have also been

made available to be used for general purpose applications [4] [10]. With the availability of hundreds of cores in GPUs and vector processing units in GPPs, system designers are left with many decisions about how to split applications between GPUs and GPPs in order to effectively use the underlying platforms while minimizing communication and coordination overhead.

In our first contribution, we develop a framework to schedule and map DSP systems implemented on such heterogeneous platforms. Starting from a formal model, we make use of different types and levels of parallelism that exist in the application to efficiently use the underlying heterogeneous platform. Our workflow starts by exploring different levels of vectorization to efficiently use the SIMD units and increase the system throughput. Then it uses a Mixed Linear Programming (MLP) formulation to model the partitioning problem of the application between GPPs and GPUs. In this formulation, we account for the difference in the execution time of every actor on various processors. Using the information in the input application graph, the set of linear inequalities perform a graph analysis to account for the communication overhead between the processors. The MLP partitioner has the objective of reducing the total graph latency and the MLP solution provides the mapping and ordering of the actors on the given processors. The design space is systematically explored and the system designer can choose a solution that best matches the constraints of the application.

1.2.2 Exposing Parallelism and Dynamic Scheduling Opportunities for DSP Applications

Synchronous Dataflow (SDF) is among the most commonly used DSP-oriented dataflow models [11]. In SDF, the number of tokens (data values) produced and consumed at each actor (dataflow graph vertex) port is constant across each firing (invocation) of the associated actor. Multiprocessor scheduling for an SDF graph conventionally involves a transformation of the graph to its equivalent Homogeneous SDF (HSDF) form. Such *HSDF expansion* is performed to more fully expose the parallelism in an SDF representation, including task and data parallelism. However, the HSDF expansion transformation may in general produce an exponential increase in dataflow graph size — the numbers of actors and edges (graph size) in the HSDF equivalent graph is in general not polynomially bounded in the size of the corresponding SDF graph. This expansion can thus lead to very slow or memory consuming scheduler performance, which limits the effectiveness of design space exploration and other analysis tasks. Furthermore, considering the typical number of cores in contemporary DSP platforms, the full parallelism exposed by an HSDF expansion may not be beneficial because such parallelism (e.g., tens to hundreds of parallelizable firings per scheduling iteration) may overwhelm the number of available cores (e.g., the latest Texas Instruments TMS320C6678 processor has eight cores).

We present a scalable dataflow graph intermediate representation, called *partial expansion graphs (PEGs)*, and we present an implementation and scheduling strategy that utilizes PEGs to realize DSP systems efficiently on multicore platforms. Intuitively, PEGs provide benefits during the scheduling process by allowing designers and design

tools to tune the trade-offs involving dataflow graph (intermediate representation) complexity and exposed parallelism based on relevant considerations, such as the amount of parallelism that is available in the target platform. This allows scheduler construction and exploration to proceed more effectively compared to always operating on fully expanded HSDF graphs (high complexity / high parallelism exposure) or unexpanded SDF graphs (low complexity / low exposure), which can be viewed as the extremes in the space of possible PEG representations. We develop methods to tune PEG representations based on application and architecture characteristics, systematically integrate PEG-based scheduling into real time DSP operating systems, naturally express diverse forms of parallelism in dataflow graphs, and perform buffer management operations associated with PEG representations to dynamically distribute time-varying firing loads (i.e., when execution times can vary dynamically).

1.2.3 Scheduling and Mapping of Partial Expansion Graphs

Implementation models such as the PEG model help system designers to realize their applications in simpler, more efficient ways. However, complex spaces of design parameters typically need to be explored in order to derive system implementations from such models. We develop three main contributions in this thesis to help address the challenges associated with exploring such complex design spaces for multiprocessor DSP system implementation.

First, we develop methods to optimize the amount and types of parallelism that an application should expose to efficiently run on the platform that it is targeted to. For

an application that has a limited number of expandable (data parallel) actors, an exhaustive search can be accomplished in reasonable time, while in other scenarios, the number of permutations of expandable solutions can increase exponentially, and strategic exploration of candidate solutions is required. Therefore, we address this problem using a probabilistic search technique. In particular, we develop a Particle Swarm Optimization (PSO) [12] approach, where each particle represents a specific amount of partial expansion for the application. The particles collectively explore the solution space and generate different PEG graphs.

Second, given an expanded PEG graph, a mapping solution that places every PEG instance on an available core is required. We provide a mapping heuristic that enables the underlying platform to use diverse forms of parallelism (i.e., allow different types of parallel instances to run simultaneously). The heuristic has two objectives: 1) to utilize data and task parallelism to speedup the application by identifying what we define as Delay Parallel Regions (DPRs), and 2) to map the corresponding DPR instances to different cores. The second objective aims to balance the load across the set of available processor cores in order to make use of pipeline parallelism.

Finally, after using the first two methods, described above, to schedule PEGs, accurate evaluation of mapping candidates has to take place. In the context of probabilistic search methods, such evaluation is often called “fitness evaluation” or “fitness function evaluation.” Given the dynamic nature of PEG applications and the necessity of changing system parameters at runtime, such as in LTE where the system is configured depending on the communication channel status, offline evaluation is not sufficient for accurate performance estimation of an implementation. Therefore, we provide a novel method for

measuring the fitness function by implementing a generic solution that can be customized easily for different types of expansion and mapping techniques. We also implement the PSO engine and the mapping heuristic on the targeted platform in order to close the evaluation loop and update the solution parameters systematically and accurately. Although we use a programmable digital signal processor as the embedded platform in our experiments, our “embedded fitness evaluation” approach can be applied to other kinds of platforms as well.

1.2.4 Integration of Scheduling Solutions with Design Environments

We make use of the proposed mapping workflow for GPP-GPU heterogeneous platforms in the context of system design for software defined radio (SDR), which is an important signal processing application domain. Our experiments specifically build on a popular design environment called GNU Radio [13] for developing SDR applications. GNU Radio provides libraries of kernels and other utilities that facilitate application of MBA-based design methods.

More specifically, we show how our new MBA-based design methods can be applied to integrate GNU Radio with a formal dataflow design framework that provides capabilities for optimized SDR implementation on multicore platforms. We achieve such an integration through a novel method for automatically translating GNU Radio applications into formal dataflow representations. This integration allows GNU Radio to employ a variety of dataflow-based schedulers that effectively target heterogeneous multicore platforms.

Dataflow formalisms provide our design flow with a structured, portable application description that can be applied for systematic vectorization, latency optimization, and other design objectives. These objectives can ultimately be incorporated into an SDR application through a GPU specific library of SDR actors. For this work, we have constructed GRGPU, which is such a library written for GNU Radio. This integration enables new target specific optimizations for performance improvement and provides enhanced retargetability. We demonstrate this new model based design flow for GNU radio on a standard SDR benchmark.

Furthermore, we have developed a code generator to automate the realization of DSP systems modeled using PEG representations. Our code generator makes use of available Application Programming Interfaces (APIs) on digital signal processors, and automatically generates deployable code for optimized application coordination and scheduling.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. In the next chapter, descriptions of dataflow models, DSP applications and environments, architecture models, and our targeted class of multiprocessor scheduling problems are given. In Chapter 3, we explain the implementation of our model based approach to migrate and schedule DSP applications onto heterogeneous multiprocessor platforms. In Chapter 4, we cover implementation details, integration with GNU Radio, and design space exploration techniques associated with the framework proposed in Chapter 3. Formal definitions, operation and

evaluation of partial expansion graphs are presented in Chapter 5, while scheduling methods for PEG models are proposed in Chapter 6. Finally, conclusions and directions for future work are discussed in Chapter 7.

1.4 Summary

In this chapter, we have motivated trade-offs between manually porting legacy code to new multiprocessor platforms, and automating this process using model based approaches. We discussed how to facilitate different parts of MBAs through the four contributions of this dissertation.

First, we developed a workflow to schedule digital signal processing systems on heterogeneous platforms consisting of general purpose processors and graphics processing units. Second, we discussed how to use the partial expansion graph as an implementation model that expresses a carefully tuned amount of parallelism from an application model depending on the target platform. Third, we discussed how to schedule and map PEG graphs while profiling the solution using a novel method of implementing a generic solution on the targeted embedded platform. Fourth, we discussed how to complete the MBA design loop by implementing code generators for our proposed workflow, and integrating our scheduling solutions in the context of contemporary DSP design environments and platforms.

Chapter 2

Background

In this chapter, we describe different application and platform models and how they are used during the application-to-architecture mapping process.

2.1 Dataflow Applications Models

Dataflow models are widely used in design, analysis and implementation of DSP systems. Different models exist to match various types of applications as synchronous dataflow (SDF) [14], cyclo-static dataflow (CSDF) [15] for static models and Boolean dataflow (BDF) [16], core functional dataflow [17] for dynamic applications. A dataflow model of an application captures important data dependency information between system modules.

2.1.1 Formal Description

A dataflow graph G consists of a set of vertices V and a set of edges E . The vertices or *actors* represent computational functions, and edges represent FIFO buffers that can hold data values, which are encapsulated as *tokens*. Depending on the application and the required level of model-based decomposition, actors may represent simple arithmetic operations, such as multipliers or more complex operations as turbo decoders.

A directed edge $e(v_1, v_2)$ in a dataflow graph is an ordered pair of a source actor

$v_1 = src(e)$ and sink actor $v_2 = snk(e)$, where $v_1 \in V$ and $v_2 \in V$. When a vertex v executes or *fires*, it consumes zero or more tokens from each input edge and produces zero or more tokens on each output edge. Synchronous data flow is a specialized form of dataflow where for every edge $e \in E$, a fixed number of tokens is produced onto e every time $src(e)$ is invoked, and similarly, a fixed number of tokens is consumed from e every time $snk(e)$ is invoked. These fixed numbers are represented, respectively, by $prd(e)$ and $cns(e)$. Some implementations of SDF graphs also support non destructive consumption (*peeking*) of an integer number of tokens from an input edge. This integer (non-negative) number is represented by the attribute $peek(e)$ ($peek(e) = 0$ means that peeking is not employed on e). Peek attributes are useful for actors that read histories of tokens such as Finite Impulse Response (FIR) filters. Homogeneous Synchronous Data Flow (HSDF) is a restricted form of SDF where $prd(e) = cns(e) = 1$ for every edge e .

Given an SDF graph G , a *schedule* for the graph is a sequence of actor invocations. A *valid schedule* guarantees that every actor is fired at least once, there is no deadlock due to token underflow on any edge, and there is no net change in the number of tokens on any edge in the graph (i.e., the total number of tokens produced on each edge during the schedule is equal to the total number consumed from the edge). If a valid schedule exists for G , then we say that G is *consistent*. For each actor v in a consistent SDF graph, there is a unique *repetition count* $q(v)$, which gives the number of times that v must be executed in a minimal valid schedule (i.e., a valid schedule that involves a minimum number of actor firing).

This minimal schedule executes a unit of execution that we refer to as one *graph iteration* of the given SDF graph. Even though they are formulated in the context of

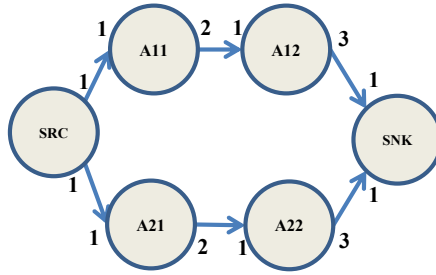


Figure 2.1: Example of a SDF graph.

sequential schedules, the concepts of repetition counts and graph iterations are also fundamental for multiprocessor scheduling of SDF graphs. Furthermore, associated with any valid schedule S , there is a unique positive integer B , called the *blocking factor* of S , such that S invokes each actor v exactly $B \times q(v)$ times [11]. This operation is also known as *vectorization* of S .

In general, a consistent SDF graph can have many different valid schedules, and these schedules can differ widely in the associated trade-offs in terms of metrics such as latency, throughput, code size, and buffer memory requirements [18]. Figure 2.1 shows a typical SDF graph to model the mp-sched benchmark, which we describe later in Chapter 4. The repetition counts for this example are: $(SRC, 1)$, $(A11, 1)$, $(A21, 1)$, $(A12, 2)$, $(A22, 2)$, $(SNK, 6)$.

Using the dataflow model of an application, different design space exploration techniques can be applied. Such techniques can be useful to detect and exploit possibilities for parallel execution across actors, schedule actors for efficient execution, and derive buffer bounds. Some of these analysis techniques can be particularly useful when the DSP system is targeted to a multicore platform.

Figure 2.2 shows three sources of parallelism that can be naturally expressed when

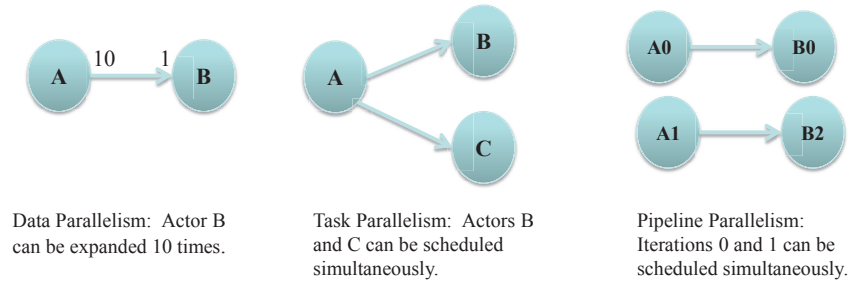


Figure 2.2: Expressing parallelism using dataflow graphs.

using dataflow graphs to model streaming applications. A multirate edge e is an edge that has different values for $prd(e)$ and $cns(e)$. This means that the final repetition counts for the source and the sink actors of e will be different. These edges can express data (loop) parallelism in an application. Task parallelism is shown in a *broadcast* actor when data flowing through the graph goes through two different chains that originate at a common actor. Finally, pipeline parallelism can be found when multiple instances of valid schedules are executed simultaneously.

2.1.2 Dataflow Interchange Format

The Dataflow Interchange Format (DIF) [9] is a standard language and associated software package that provide for mixed-grain specification, analysis and synthesis for dataflow-based design and implementation of signal processing systems. DIF provides a unified framework to facilitate technology transfer of applications among different DSP design tools. In order to achieve this goal, the DIF front end includes various tools to automate the importing and exporting of application between the DIF environment and other dataflow-based design tools.

To facilitate rapid prototyping, designers can focus on describing the dataflow be-

havior of their application using the DIF package, which comes with a set of algorithms and engines to analyze and optimize different application properties. These features of DIF, together with a set of library module implementations that are targeted to a specific platform, can be used to derive high quality embedded software implementations with a high degree of automation [17].

The DIF Language (TDL) provides a language for representing DSP-oriented dataflow application graphs. A distinguishing characteristic of TDL is support for a variety of different dataflow models of computation, and for integrating subsystems in different representations so that individual subsystems can be represented and analyzed in terms of dataflow techniques that are tailored towards the behaviors and constraints associated with those subsystems.

2.1.3 Streaming Application: Software Defined Radio

In recent years, we have witnessed rapid growth in the computational capacity for fixed and floating point arithmetic in processors. This has allowed radio tasks that could once only be implemented in dedicated analog circuits, analog/digital ASICs, or FPGA logic to now be achievable using software. Additionally numerous modern wireless communications standards have chosen to exploit the low cost of computational resources and have begun to significantly drive up the complexity of waveforms in order to achieve improved spectral efficiency and coverage.

This increase in complexity has led to two choices for those building radios for these standards: continue to fabricate vastly larger ASICs or explore software-based solutions.

With the increases in application complexity, ASIC-based solutions must consume many more gates to implement all of the functions that the radio may need to perform, and requires complex hardware state machines to orchestrate the interaction of the various specialized subsystems. Software defined radio avoids much of these problems by reusing computing resources on a more fine grained level. This reuse is achieved by simply implementing all of the necessary routines in software, and implementing flow graphs for signal processing routines in software [19] [20] [21].

As SDR applications have the parallelism and performance demands to be suitable for many of these nascent multicore architectures, this creates the potential of many unique targets to be considered when going to implementation. SDR applications have different levels of parallelism that can be exploited on multicore platforms, but design and programming difficulties have inhibited the adoption of specialized multicore platforms like graphics processors (GPUs). Porting code to new architectures can be an error-prone time-consuming process that involves careful tweaking of signal flow parameters to achieve satisfactory performance.

2.1.4 Pre-optimized Kernels: GNU Radio

GNU Radio [13] is a software development framework that provides software defined radio developers a rich library and a customized runtime engine to design and test radio applications. GNU Radio is extensive enough to describe audio radio transceivers, distributed sensor networks, and radar systems, and fast enough to run such systems on off-the-self radio hardware and general purpose processors. Such features have made

GNU Radio an excellent rapid prototyping system, allowing designers to come to an initial functional implementation quickly and reliably. GNU Radio was developed with general purpose programmable systems in mind. Often initial SDR prototypes were fast enough to be deployed on general purpose processors or needed few custom accelerators. As new generations of processors were backwards compatible with software, GNU Radio implementations could track with Moore's Law. As a result, programmable solutions have been competitive with custom hardware solutions that required longer design time and greater expense to port to the latest process generation.

GNU Radio is an open-source engine and has a collection of many common radio primitives. It allows users to specify a directed acyclic graph (DAG) by using a Python script to instantiate previously-compiled blocks and interconnect them at runtime. These blocks represent common signal processing operations, ranging from digital filters to modulators to forward error correction. A hierarchical flow graph mechanism is provided to allow primitive signal processing blocks to be rapidly connected together to form a full flow graph. A typical flow graph begins with a data source block, proceeds sequentially down a number of signal processing blocks, and then terminates in a data sink block.

Between each block is a buffer that is managed transparently. Buffers are generally refined into implementations that are appropriate for the targeted architecture and operating system. Most commonly the buffer implementations utilize memory allocated on the heap and are carefully matched to the system page size. Blocks contain buffer readers and writers that maintain their appropriate pointers into each of their input and output buffers, all of which is typically transparent to the user.

GNU Radio currently has two automated run-time schedulers. The original one is single-threaded. A topological sort of the blocks is executed in order, where each actor is executed until its input buffer is exhausted. When the last block is completed, execution resumes at the first block. The multithreaded scheduler instead instantiates each block in its own thread, where mutexed buffered FIFO queues are used to pass data between them. The second scheduler involves more system overhead, but allows GNU Radio to run efficiently on multicore processor architectures.

2.2 Architecture Models

Many types of configurable platforms have been proposed to enhance throughput and power consumption efficiency for DSP applications. Hardware platforms can be more customizable for a given protocol specification but as the degree of freedom increases, leveraging common architecture parameters becomes more challenging compared to software platforms. In this section, the basic elements that customize these platforms are described and an overview is given on contemporary devices.

2.2.1 Contemporary Platforms

Multiprocessor platforms include graphics processors, multicore general purpose processors, tile architectures, and multicore digital signal processors. Even for off-the-shelf computing platforms, a heterogeneous mix of multiprocessor devices is likely, including at least one GPU and a multiprocessor GPP.

Figure 2.3 illustrates the architecture and memory hierarchy of a typical CUDA

GPU. This device consists of a number of streaming multiprocessor (SMs), where each SM consists of multiple scalar processors (SPs). Following a Single Program Multiple Thread (SIMT) paradigm, CUDA kernels can be configured into grids of blocks where every block consists of a grid of threads. Every thread block or CUDA block gets assigned to an SM, and threads within a block are executed using SPs. Currently an SM has eight SPs where every group of 32 threads are simultaneously scheduled. To launch a kernel that runs on a CUDA device, the host processor has to configure the number of CUDA blocks, as well as the number of threads for each block. These numbers vary for different applications depending on the amount of data parallelism that can be achieved.

The CUDA work flow consists of serial code running on the host machine and a parallel kernel running on the device. Initially, the input data resides on the host memory. Special functions are provided to copy the data from the host to device memory, where the latter can be accessed by all the CUDA blocks. All SPs within an SM share 16KB or 48KB of shared memory depending the compute capability. The shared memory has lower latency and higher throughput than the device memory. Shared memory is cleared after a kernel completes, making it necessary for the programmer to copy the partial results back to the device memory between successive dependent kernels. Also, only threads within a CUDA block can synchronize inside a kernel, whereas different thread blocks can synchronize after a kernel launch.

Figure 2.4 shows another multiprocessor example, which is based on a family of state-of-the-art Texas Instruments evaluation modules. The platform consists of several fixed or floating point PDSP very long instruction word cores, where every core has its own configured L1 cache or memory. New platforms also have shared L2 memories,

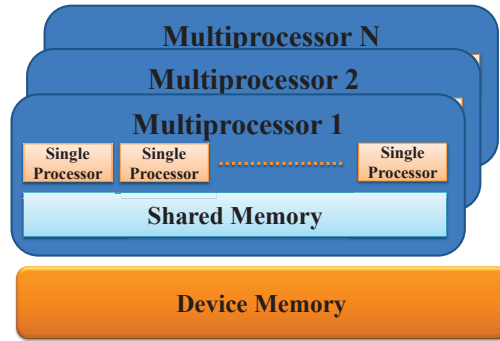


Figure 2.3: GPU memory hierarchy.

for which data access has comparable latency to private L2 memories. An Enhanced Direct Memory Access (EDMA) switch fabric can be used to communicate between the cores and the external world. The TI programming model is rich, with various tools and application programming interfaces to create and schedule different types of threads, perform Inter Processor Communication (IPC), and facilitate system development with pre-optimized kernels.

In this dissertation, we are targeting a heterogeneous mix of off-the-shelf computing platforms that are used in common workstations and equipped with pre-optimized signal processing software packages such as GNU Radio. We also enhance the generality and cross-platform aspect of our approach by targeting additional DSP platforms, such as the Texas Instruments TMS320C6472 (six cores) evaluation modules equipped with real time operating systems.

2.2.2 Model Elements

When mapping systematically from an application model to an architecture model, the following parameters are important to consider in relation to the targeted architecture

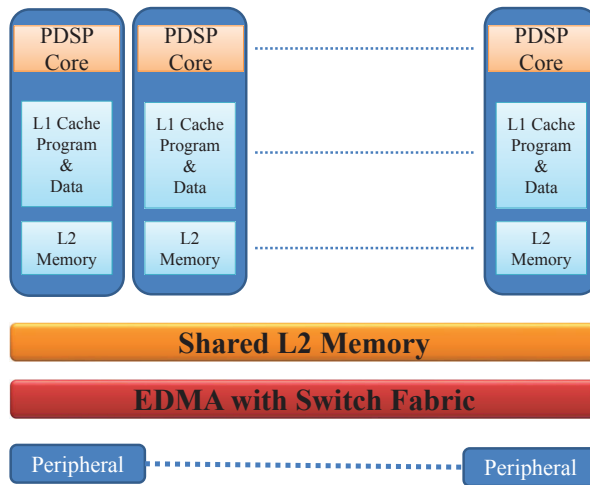


Figure 2.4: A typical Texas Instruments evaluation board.

model.

- The number and type of processing units. Processing units can vary from complete processors with elaborate instruction sets to customized hardware acceleration units. The number of processors and functional units gives us bounds on the amount of parallelism that we can exploit from the application graph. Also, common DSP functions can make use of of dedicated hardware that guides the process of actor-to-processing-unit assignment.
- Memory hierarchy and communication schemes. The access speed, size and latency per transaction of different memory levels affect the performance of DSP functions. Values of these parameters give us guidelines on how much computation to perform per memory transaction in order to mask communication latency. Second, the memory hierarchy affects where to place the FIFO buffers required between actors (i.e., multiple data dependent actors can take advantage of communication through

fast but small memories).

Fully utilizing such a multiprocessor setup requires the identification of appropriate parallelism in the application domain and the implementation of such parallelism efficiently on the targeted platform. More significant challenges are posed to achieve efficient multiprocessor mappings of DSP applications on heterogeneous platforms. When deriving mappings onto heterogeneous multiprocessors, designers must in general balance computational loads, efficiently use the given processor types, and account for communication costs. In many DSP systems, this problem is further complicated by multirate application descriptions, and the need to satisfy latency constraints, as well as constraints on throughput and power. In the next section we go over the basic decisions that a multiprocessor scheduler must address and different design trade-off.

2.3 Multiprocessor Scheduling

A variety of multiprocessor scheduling techniques can be applied to DSP applications. In this section, we discuss the different tasks involved in multiprocessor implementation, and then discuss issues related to the exploitation of parallelism for different types of applications.

In [22], Kwok provides a survey of many scheduling algorithms and proposes new scheduling techniques that provide efficient solutions over the state of the art, and can scale across different kinds of parallel platforms.

Multiprocessor implementation involves a number of different tasks, which are enumerated below. A related decomposition of multiprocessor implementation, but one that

is focused specifically on the scheduling aspect, is presented in [23].

1. Clustering and assignment — this involves grouping subsets of actors into *clusters*, and assigning each cluster to a processing unit.
2. Ordering — Ordering can take place at two levels. The first is ordering the execution of clusters that form the graph, and the second is ordering the execution of actors within a single cluster.
3. Buffering — this includes calculating the sizes (capacities) for the FIFO buffers that are required between different clusters and different actors.
4. Synchronization — two types of synchronization can take place. In the first form, called *barrier synchronization*, all of the processing units wait until they reach a certain point in the flow of execution. Once all processors arrive at this point, they exchange results and then resume concurrent execution. This type of synchronization usually appears in Single Program Multiple Data (SIMD) programming models. The other form of synchronization occurs between multiple actors to avoid buffer overflow and underflow.

Depending on the application graph type, the first three tasks can be performed at compile-time or at run-time. Compile-time clustering, assignment, and buffering provide low-overhead implementation, and can be applied by extracting statically schedulable regions from the application [24]. Analysis of such regions can allow designers to explore important trade-offs among context switching, parallel execution, and memory management efficiency.

2.4 Summary

In this chapter, we have reviewed general literature for basic components in MBAs. We have given a detailed explanation for dataflow graph models for DSP systems. We have shown how different forms of parallelism can be expressed in such graphs, and reviewed DIF as one of the known design tools to express and compile such graphs. We have discussed GNU Radio, an open source design environment that uses MBAs to realize software defined radio applications. We have explained the architectures of two contemporary platforms — Texas Instruments programmable digital signal processors and CUDA GPUs — followed by definitions of the basic tasks of multiprocessor scheduling and mapping.

Chapter 3

Scheduling DSP Systems on Heterogeneous GPP and GPU Platforms

3.1 Workflow Description

As we discussed in the previous chapter, model based design has proved to be useful in providing application developers with fast, flexible development environments, as well as the ability to target heterogeneous processing units. Software based implementations have the ability to change to accommodate new standards while implicitly taking advantage of Moore's law through increases in processor performance. In this chapter, we describe the steps of our proposed workflow in detail, and provide a new mixed linear programming formulation for heterogeneous multiprocessor scheduling. Preliminary versions of this work have been presented in [25] and [26].

The design flow of our model based approach to migrating DSP applications onto heterogeneous platforms is described in the following steps.

1. Designers develop a model of their DSP application using an appropriate model of computation, and with no consideration at this stage for the underlying platform. As the domain specific environment has an execution engine and a library of DSP kernel components, designers can validate correct functionality of their application. Architecture models for the underlying platforms can also be built independently of the application at this stage.

2. If actors of interest are not in the available libraries for the selected target platform, the designer writes accelerated versions of these actors (e.g., targeting the C, CUDA or Verilog languages). Actor implementation at this stage focuses on exposing parallelism in a parameterized way so that it can be tuned to match the targeted hardware.
3. Through automated or manual processes (or some combination), instantiated actors are assigned to specific processing elements in the targeted heterogeneous platform. The available processing element types can include, for example, GPPs, programmable digital signal processors, GPUs or hardware accelerators. This step may be revisited often as part of an iterative system-level design space exploration process.
4. Mapping results derived in the previous steps are utilized by integrating the original application description with platform-specific libraries, and with software that is automatically synthesized based on the selected mapping configurations. These steps of integration and code generation combine to produce implementations that can be experimented with on the targeted platform, and deployed as final solutions when they are validated to satisfy the given application constraints.

The following sections cover the first three steps in detail, specifically as they relate to our approaches for augmenting the design flow to accommodate new target platforms and design environments, such as CUDA in the GNU Radio environment. In Chapter 4, the last step is explained for the same programming environment.

3.1.1 Writing Accelerated Kernels

GNU Radio gives the designer a dataflow-based facility to describe the application graph. Actors are individually accelerated using GPU specific tools. If an actor of interest is not present in the GPU accelerated library, the developer switches to the GPU customized programming environment, which in our case is CUDA. Also, other tools such as OpenCL and [27] can also be used to implement such actors. We show in 3.2 that actors are later profiled to be scheduled on the heterogeneous processors. The designer is still saddled with difficult design decisions, but these decisions are localized to a single actor. System level design decisions are orthogonal to this step of the design process. While we do not aim to replace the programming approach of the actors functionality, the following design strategy lends itself to later design space exploration by the developer.

As with other GPU programming environments, in CUDA designers must divide their applications into levels of parallelism: threads and blocks, where threads represent the smallest unit of a sequential task to be run in parallel and blocks are groups of threads. In our experience, SDR actors vary in how to use thread level parallelism, but tend to realize block level parallelism with parallelism at the sample level. The ability to tightly couple execution between threads within a block creates a host of possibilities for the basic unit of work within a block, be it processing a code word, multiplying and accumulating for a tap, or performing an operation on a matrix. Because blocks are decoupled, only fully independent tasks can be parallelized. For SDR those situations tend to arise between channels or between samples on a single channel. Some samples may overlap between blocks to support the processing of a neighboring sample, but this redundancy is

often more than offset by the performance benefits of parallelization.

The performance of this parallelization strategy is strongly influenced by the number of channels or the size of a chunk of samples that can be processed at one time. When the application requests processing on a small chunk of sample, there are few blocks to spread across a GPU leaving it under utilized, while large chunks enable high-utilization. The performance difference between small and large chunks is non-linear due to the high fixed latency penalty that both scenarios experience when transferring data to and from the GPU and launching kernels. When chunks are small, GPU time is dominated by transfer time, but when chunks are larger, computation time of the kernel dominates, which amortizes the fixed penalty delay. As the application dictates these values, actors must be written in a parameterized way to accommodate different size inputs.

3.1.2 Partitioning, Scheduling, and Mapping

Once actors are written, system level design decisions must be made, such as assigning which actors are to invoke GPU acceleration. With some applications, the best solution may be to offload every actor that is faster on the GPU than it is on the GPP. But in some cases, this greedy strategy fails to recognize the work that could occur simultaneously on the GPP, while the host thread with the kernel call waits for the GPU kernel to finish. A general solution to the problem would consider application features such as rates of firings, dependencies, and execution times on each platform of each actor, as well as architectural features such as the number and types of processing elements, memories, and topology.

When the application can be extracted into a formal dataflow model, schedulers will not only respect these constraints but are able to optimize for buffer assignments. The applicability of such techniques for specialized multicore platforms are still open research, and this design flow enables greater experimentation with them for SDR applications. Manual scheduling and mapping is likely to continue to dominate smaller, more homogeneous mappings, but a grounding in dataflow opens the door for new automation techniques.

When targeting a GPU or other SIMD platform, vectorization must also be considered. More vectorization tends to lead to higher utilization of the platform (and therefore higher throughput), but often at the expense of increased latency and buffer memory requirements. Also an accelerator typically requires significant latency to move data to or from the host processor, so sufficient data must be burst to the accelerator to amortize such overheads. Ideally, application designers would be simply presented with a Pareto curve of latency versus vectorization trade-offs so that an appropriate design point can be selected. However, vectorization generally influences the efficiency of a given mapping. Thus, to fully unlock the potential of heterogeneous multiprocessor platforms for DSP systems, an automated way of arriving at quality solutions is desirable. In the next section, a scheduler that accept the application and architecture descriptions and generates a varieties of solutions that targets heterogeneous multiprocessors platforms equipped with SIMD units is explained.

3.2 Multi-objective Multicore Scheduler

Automated techniques are useful as starting points for leveraging multiprocessing platforms consisting of GPPs and GPUs, with SSE [28] acceleration and CUDA acceleration. An important criterion to arriving at quality solutions in this context is the ability to explore a variety of design points efficiently and accurately.

3.2.1 From Application Model to Block Processing DAGs

Increasing the throughput of individual actors can be performed By applying actor-level vectorization (also referred to as *block processing*) [29], we can process the maximum possible number of tokens per actor execution. For an SDF graph, this objective can be achieved by using a *flat schedule* of the input graph. A flat schedule can be generated by deriving a topological sort, and invoking every actor v a number of times equal to $B \times q(v)$. While flat schedules have the potential to improve processor utilization and throughput, such schedules generally suffer from high memory usage. However, in this proposal, our objective is to increase the utilization of SIMD cores and furthermore, the available memory on a typical GPU is not a constraint for the class of SDR applications that we are targeting. Given an acyclic SDF application graph G , our scheduling approach first generates a directed acyclic graph (DAG), which we call a *block processing DAG (BPDAG)* T . T is isomorphic to G , meaning that the sets of vertexes and edges are in one-to-one correspondence with one another. Each vertex t in T represents a vectorized version of a specific vertex v in G with some vectorization factor k (i.e., t represents k successive invocations of v). We refer to each vertex in a BPDAG as a *task*.

For platforms that consist of both GPPs and GPUs, different levels of parallelism can generally be exploited in order to improve throughput. First, a fine grain level of data parallelism can be applied by utilizing the SIMD cores available in GPUs and vector operation accelerators in GPPs (if available). This level can be exploited using vectorization. A more coarse grain form of task parallelism is applied by mapping parallel tasks of the application graph onto the available set multiple processors. Both forms of parallelism may generally be exploited more effectively when $B > 1$, where B is the blocking factor. Under such a scheduling approach, the latency for a single graph iteration may increase. However, the latency for a block of B successive graph iterations may be reduced significantly, which leads to an increase in throughput (in terms of executed graph iterations per unit time). Such a trade-off is favorable in many throughput-critical systems or in applications where the increased latency does not exceed the given latency constraint.

In our workflow, we set the level of global vectorization before the mapping step to properly inform the multiprocessor scheduler of the vectorized running time of the actors in the application for each processor type. By doing so, we efficiently utilize the SIMD cores by simultaneously firing multiple graph iterations. Therefore the basic multiprocessor scheduler objective is set to minimize the overall latency L_B of B graph iterations, which provides an optimized graph execution throughput of N/L_B graph iterations per unit time. Here B is a parameter than can be changed flexibly in our framework to help explore the scheduling design space.

The BPDAG is sent to the core of our multiprocessor scheduling engine to perform task mapping (assignment of tasks to cores) and ordering (ordering of tasks assigned to the same core). BPDAG tasks are annotated with their running times, which generally

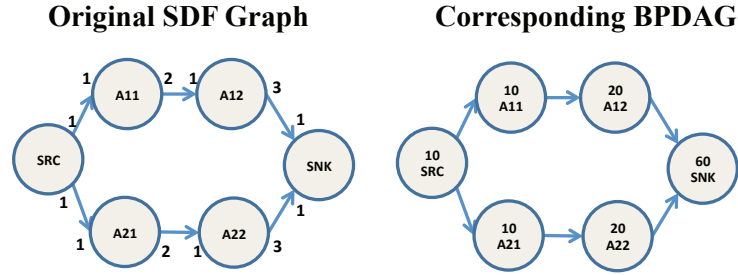


Figure 3.1: Example of an SDF graph for the mp-sched benchmark and its corresponding BPDAG.

are functions of the vectorization factors. Figure 3.1 shows an example of transforming an SDR benchmark, called mp-sched and explained in Section 4.4, to its corresponding BPDAG for a blocking factor of 10. This blocking factor generates the vectorization factor and is used to derive each task in the BPDAG.

3.2.2 Architecture Model

Many platforms have been proposed to run SDR applications. A typical solution consists of an RF front end, units that can perform digital signal processing (processors, IP cores, etc.), and interconnection media. The system input is the digitized signal produced by the analog to digital converter. In typical SDR platforms, the processing is executed by multiple heterogeneous processors that are suitable for different actor operations. Actors that perform control functions require complex pipelines and branch prediction units. GPPs (e.g., Intel quad cores) are usually suitable for these actors. Another relevant type of processor is the Single Instruction Multiple Thread (SIMT) type (e.g., NVIDIA GPUs). These processors have less sophisticated cores that are able to process individual functions on different data sets (e.g., symbol mapping and coding). Many physical layer actors require this kind of data parallelism, and GPUs often exhibit good performance for such

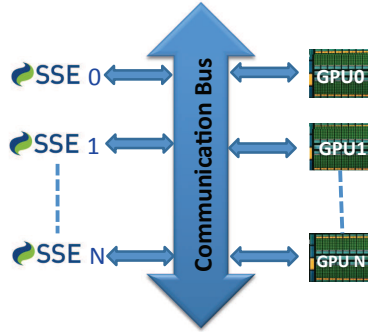


Figure 3.2: Example of typical off-the-shelf platform consisting of GPPs and GPUs using the PCI bus as communication medium.

actors. Another possible operators are IP cores that are designed to efficiently execute some of the algorithm operation as Fast Fourier Transform and Turbo Decoding. Usually these core are implemented either on ASIC or Field Programmable Gate Arrays.

The target platform that we consider consists of a multicore GPP, possibly with one or more SIMD accelerators (e.g., SSE extensions in Intel cores) accompanied with one or more GPUs. All of the processors are assumed to be connected with an all-to-all communication medium. Figure 3.2 shows an example for a typical platform. If two dependent actors are allocated on the same processor, data movement will take place using shared memory at zero cost; otherwise, communication occurs across a contention-based communication medium (e.g., PCI bus).

3.2.3 Problem Formulation

The input to our multiprocessor scheduler consists of the task graph, a description of the target platform; and profile data (for execution time estimation) for each task. The objectives of the scheduler are to perform task assignment and ordering in order to meet a given objective function under a given set of constraints. For assignment, the scheduler

is responsible for mapping tasks to processors and edges to communication media. Non-preemptive operation is assumed for both tasks and edges (communication). The ordering aspect of scheduling is required if multiple tasks (edges) are assigned to the same processor (medium). Execution times for tasks are the associated BPDAG vertexes processing times (as determined by the actor profiles together with the associated vectorization factors), while execution times for edges are estimated as the data communication times. The input to the multiprocessor scheduler consists of the following items.

a — Architecture description: The platform is described by a set P of processors and a set β of communication buses.

b — Application description: The application model (input BPDAG) consists of a set T of tasks, and edges E .

c — Dependency descriptions: Dataflow dependencies are defined by the *src* and *snk* functions described in Section 2.1.

d — Task and edge profiles: The task and edge execution times are obtained by simulating the tasks (edges) on different processors (communication media). These profiles are described by two functions: $\text{RTP}(t \in T, p \in P) \rightarrow R$ defines the execution time of task t on processor p , and $\text{REB}(e \in E, b \in \beta) \rightarrow R$ defines the execution time of edge e on bus b . Here, R is the set of positive real numbers.

e — Dependency analysis: Task t_1 is said to be *dependent* on task t_2 if there is a path that starts at t_1 and ends at t_2 . If no such path exists between t_1 and t_2 , then they are called *parallel* tasks. A similar concept can be applied to edges.

The actor dependency function $AD : A \times A \rightarrow \{0, 1\}$ is defined by the equation:

$$AD(a_1, a_2) = \begin{cases} 1 & \text{if } a_1 \text{ and } a_2 \text{ are dependent} \\ 0 & \text{if } a_1 \text{ and } a_2 \text{ are parallel} \end{cases}$$

The objectives of the scheduler are to perform task assignment and ordering in order to meet a given objective function under a given set of constraints. For assignment, the scheduler is responsible for mapping tasks to processors and edges to communication media. Non-preemptive operation is assumed for both tasks and edges (communication). The ordering aspect of scheduling is required if multiple tasks (edges) are assigned to the same processor (medium). The input summarized in items *a-e* above is sent to the multiprocessor scheduler in order to perform the operations of mapping and ordering.

3.3 Multiprocessor MLP Scheduler

The problem description in Section 6.2.1 can be solved using available heuristics and optimal schedulers. As offline analysis is suggested to schedule static applications, a mixed linear programming (MLP) heterogeneous multiprocessor scheduler is proposed in order to find efficient solutions. The MLP scheduler consists of a set of equalities and inequalities that describe the application and architecture graphs, solution variables, constraints and objective.

3.3.1 Basic Variables

The basic MLP variables in our formulation are as follows.

- *Mapping variables:* $\forall t \in T$ and $\forall p \in P$, $XT[t, p] = 1$ if task t is assigned to processor p , and $XT[t, p] = 0$ otherwise. Similarly, $\forall e \in E$ and $b \in \beta$, $XE[e, b] = 1$ if edge e is assigned to bus b , and $XE[e, b] = 0$ otherwise.
- *Ordering variables:* \forall parallel tasks t_1 and t_2 that are assigned to the same processor, $YT[t_1, t_2] = 1$ if t_1 is scheduled to run before t_2 , and $YT[t_1, t_2] = 0$ if t_1 is scheduled to run after t_2 . A similar formulation is applied for parallel edges.
- *Actual running time:* $\forall t \in T$, $RT[t]$ is the actual (platform-dependent) execution time of the task t depending on its mapping. Similarly, $\forall e \in E$, $RE[e]$ is the actual token transfer time for the edge e .
- *Start time:* $\forall t \in T$, $ST[t]$ is the start time for execution of task t . $\forall e \in E$, $SE[e]$ is start time of data transfer across edge e . These variables will be controlled by the dependencies expressed in the BPDAG and the ordering variables.

In this formulation, the basic variables (defined above) are used to derive a number of other variables. These derivations are carried out so that we can use linear equations to “detect” pairs of tasks that are assigned to the same processor. First we define the variables $ZTP[t_1, t_2, p]$, where $t_1 \in T$, $t_2 \in T$, $p \in P$, and $t_1 \neq t_2$. $ZTP[t_1, t_2, p]$ equals one if t_1 and t_2 are both assigned to p , and equals zero otherwise. Clearly, this variable depends on $XT[t_1, p]$ and $XT[t_2, p]$. This dependency can be linearized according to the following constraints:

- $ZTP[t_1, t_2, p] \geq XT[t_1, p] + XT[t_2, p] - 1$
- $ZTP[t_1, t_2, p] \leq XT[t_1, p]$

- $ZTP[t_1, t_2, p] \leq XT[t_2, p]$

The first equation handles tasks are assigned to the same processor, while the other two handle the three other cases. It can be shown that these equations dominate the problem size, and as a result, they contribute significantly to the time required by the applied solver.

Next, we define another set of variables $ZT[t_1, t_2]$, where $t_1 \in T, t_2 \in T, t_1 \neq t_2$. $ZT[t_1, t_2]$ equals one if the two tasks t_1 and t_2 are collocated. These variables can be easily derived by the following inequality:

$$ZT[t_1, t_2] \geq \sum_{p \in P} ZTP[t_1, t_2, p].$$

The derived variables ZT will be used in two cases. First, for collocated parallel tasks, these variables help to adjust the start times of tasks based on their ordering. Second, if a pair of tasks is connected by an edge, then these variables serve to make the corresponding edge transfer time equal to zero, which is appropriate since the communication occurs through processor shared memory.

3.3.2 Constraints

We use the following inequalities to formulate our targeted heterogeneous scheduling problem:

- *Assignment:* Every task (edge) is assigned to only one processor (communication

medium):

$$\forall t \in T, \sum_{p \in P} XT[t, p] = 1 \text{ and } \forall e \in E, \sum_{b \in \beta} XE[e, b] = 1$$

- *Task running time:* $\forall t \in T, p \in P$

$$RT[t] \geq XT[t, p] \times RTP[t, p]$$

- *Edge running time:* $\forall e \in E, b \in \beta$

$$RE[e] \geq XE[e, b] \times REB[e, b] - K \times ZT[src(e), snk(e)]$$

where K is a very large number. The second term in this inequality models the “edge zeroing process” (i.e., the process of setting an edge’s token transfer time to zero) if the source and the sink tasks of the edge are assigned to the same processor.

- *Starting times for dependent tasks:* $\forall e \in E$

$$SE[e] \geq ST[src(e)] + RT[snk(e)]$$

$$ST[snk(e)] \geq SE[e] + RE(e)$$

These two equations guarantee the proper execution order of dependent tasks by also taking into consideration relevant edge execution times.

- *Starting times for parallel tasks:* Orderings for parallel tasks can be achieved using an adaptation of an equality from [30]: \forall parallel tasks $t_1 \in T$, and $t_2 \in T$, $t_1 \neq t_2$:

$$ST[t_1] \geq ST[t_2] + RT[t_2] - K(1 - YT[t_1, t_2]) - K \times ZT[t_1, t_2]$$

$$ST[t_2] \geq ST[t_1] + RT[t_1] - K \times YT[t_1, t_2] - K \times ZT[t_1, t_2]$$

Note that the last term effectively disables these inequalities if the two tasks are not collocated

3.3.3 Objective

Finally, the objective function minimized is the total graph latency (makespan) M , which can be specified by:

$$\forall t \in T, M \geq ST[t] + RT[t]$$

The solution of the formulated MLP problem is then sent to the workflow back-end to generate a running system. In the next chapter, implementation of different parts of the scheduling workflow is described for SDR systems used in the GNU Radio environment, while targeting platforms consisting of off-the-shelf GPPs and GPUs.

3.4 Related Work and Contribution

Many previous research efforts have considered facilitation of DSP system implementation on new computing platforms. In this section, related work that considers application and architecture modeling and multiprocessor scheduling is surveyed.

Various heuristics and mixed linear programming models have been suggested for scheduling task graphs on homogeneous and heterogeneous processors (e.g., see [31]). In these works, the problem formulations are developed to address different objective functions and target platforms for implementing the input application graphs.

In [21], a dynamic multiprocessor scheduler for SDR applications is described. The basic platform consists of a Universal Software Radio Peripheral (USRP), and cluster of General Purpose Processors. A flexible framework for dynamic mapping of SDR components onto heterogeneous multiprocessor platforms is described in [20].

In [32], the authors present a multicore scheduler that maps SDF graphs to a tile based architecture. The mapping process is streamlined to avoid the derivation of equivalent HSDF graphs which can involve significant time and space overhead.

Vectorization for single-processor implementation of SDF graphs has been studied previously (e.g., see [29] and [33]). In [34] automatic “SIMDization” (conversion to a form that utilizes SIMD acceleration on the target processor) of streaming programs from a general purpose programming approach is proposed. A combination of SIMDization techniques with homogeneous multiprocessor scheduling is also discussed.

In addition to the previous work, we target platforms that consist of multiple GP-P and GPU components, and systematically integrate SDF vectorization and inter-actor

(task-level) parallel scheduling to optimize application throughput and latency on the targeted class of heterogeneous multiprocessor platforms.

In the current GNU Radio engine, a strictly runtime multiprocessor scheduler is used to run applications through dynamic scheduling. However, for a wide range of SDR systems, offline profiling and analysis is possible, and more efficient scheduling solutions can be computed statically. To exploit such static scheduling opportunities, we provide an Mixed Linear Programming (MLP) formulation for the targeted multiprocessor scheduling problem.

Our new scheduling technique shows how to make use of three levels of parallelism in order to increase the system throughput. Our approach is restricted to acyclic SDF graphs, which can be used to represent a broad class of practical SDR applications and subsystems. Generalization of our techniques to graphs that contain cycles is a useful direction for future work.

The primary contribution of this chapter is a novel workflow for scheduling SDF graphs while taking into account actor execution times, efficient vectorization, and heterogeneous multiprocessor execution. This scheduling workflow is targeted carefully towards heterogeneous platforms that consist of on the shelf GPPs and GPUs and applications described in a domain specific optimized language. Moreover, as offline analysis can be used to generate efficient solutions, we present a novel mixed linear programming multiprocessor scheduler.

3.5 Summary

In this chapter, we described the main steps of porting DSP applications to heterogeneous platforms consisting of GPPs and GPUs. We showed how careful actor implementation should be considered while targeting such processors, where data level parallelism can be exploited efficiently through SIMT and vector processing units. In order to fully utilize such units, vectorization should be considered in order to increase the system throughput. Therefore, we utilize different forms of parallelism in two steps. First, we construct a model called the block processing DAG (BPDAG), where different configurations for vectorization and pipeline parallelism can be represented. Then we send the BPDAG to a scheduler to perform the mapping and ordering of tasks onto heterogeneous processors.

We precisely defined the various input parameters and variables that are associated with our proposed multiprocessor scheduler. In addition to the profiling of application graph actors for different processor types, we also described our architecture model, which can accommodate heterogeneous mixes of processors. We developed a mixed linear programming (MLP) formulation for our targeted multiprocessor scheduling problem. This MLP formulation has the objective of minimizing the total latency of multiple graph iterations, where the number iterations to consider is taken as a parameter of the formulation.

In the next chapter, we show in detail how our proposed workflow is integrated into GNU Radio, and we conduct experiments to profile solutions obtained using our MLP scheduler.

Chapter 4

Integration of the Heterogeneous Platform Scheduling Workflow with GNU Radio

In this chapter, we show how to use the scheduling techniques explained in Chapter 3 by implementing the design flow proposed in Figure 4.1. In this workflow, we use GNU Radio as the runtime environment for SDR applications; and the dataflow interchange format (DIF) for dataflow graph representation, analysis, and optimization. Our targeted heterogeneous platform consists of CUDA-enabled NVIDIA GPU and Intel Xeon GPP devices. Preliminary versions of this work were presented in [35] and [36].

We start by an example of porting DSP kernels to new processing platforms as explained in Section 4.1. Then in Section 4.2, we show how to automatically import a GNU Radio flowgraph to a formal description in DIF. Augmenting the GNU Radio flowgraph with APIs that facilitate offloading actors assigned to run on GPUs is explained in Section 4.3. Finally, we demonstrate the actual system realization and trade-offs between different objectives in Section 4.4.

4.1 FIR Filter Mapping to GPU Architecture

Finite Impulse Response (FIR) filters are common actors used in many DSP applications. In an FIR filter, an output sample is derived as a sum of products of input samples by the filter's coefficients. In this implementation, we take advantage of pipeline

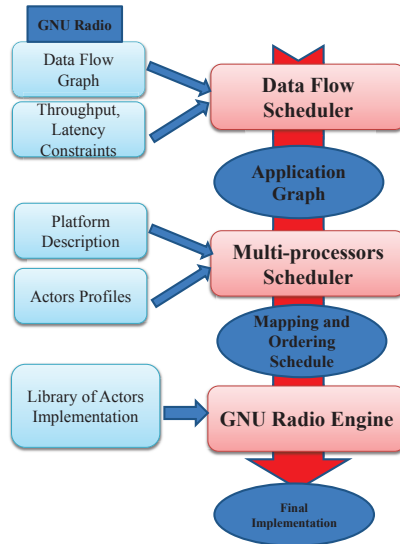


Figure 4.1: Implemented workflow for SDR applications described in GNU Radio.

parallelism across the filter output samples by considering the production of every output token as a separate dataflow graph iteration. For relatively large chunks of samples, the CUDA kernel is configured such that the number of blocks is equal to double the number of available streaming multiprocessors. By using this configuration, pipeline parallelism can be achieved efficiently if each CUDA block is responsible for calculating a different set of output samples. In other words, the required output samples are evenly distributed across the employed CUDA blocks.

To overcome the inherent stateful property of FIR filtering (i.e., consecutive output samples depend on some shared input samples), the input of every block contains an extra set of delayed input samples equal to the number of filter taps. To reduce the number of device memory accesses in our implementation, initially each of the threads performs a load of a coalesced chunk of input elements to the shared memory of its associated multiprocessor. Then each thread is responsible for calculating a single output sample. After processing a set of inputs, the threads perform a block store of the calculated results

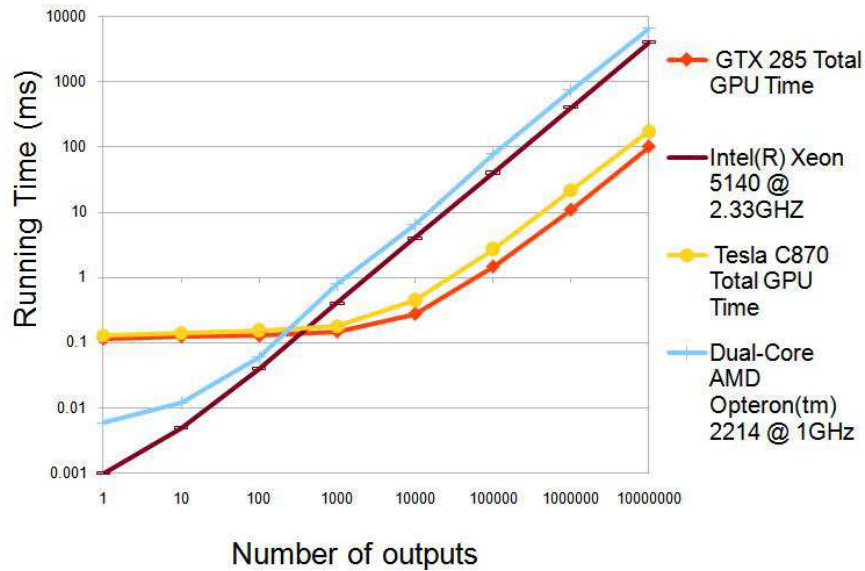


Figure 4.2: Comparison of total running time versus number of outputs for FIR filter implementation on different processors.

to the GPU device memory.

Figure 4.2 compares FIR filter performance in terms of the execution times required for GPU and GPP implementations. Even though the GPU can compute more floating point operations per second than the GPP, using it becomes beneficial only if the amount of output required is large enough to compensate for the communication overhead between the GPP DRAM and the GPU device memory. This number (i.e., the number of processed outputs) can be added as an application attribute to help in deciding on actor-to-processing-unit assignment. The execution time per input token is another important attribute that not only takes into consideration the number of processing units but also the behavior of relevant implementation details.

4.2 DIF Importer from GNU Radio

It is useful to express the dataflow structure of a DSP application using a language that emphasizes the basic dataflow behavior (i.e. the DIF language). In this section, steps towards building this model from GNU Radio is implemented.

In GNU Radio, all primitive blocks take one of the following two forms, which becomes an important consideration for scheduling.

- Synchronous Blocks — Primitive blocks inheriting from a *gr-sync-block* always maintain a fixed ratio of input items consumed to output items produced. This may be 1-to-1 or N -to- M , but it is a fixed value and this is enforced by the methods available to each block implementation.
- Non-Synchronous Blocks — Primitive blocks inheriting from a *gr-block* do not need to maintain a fixed input to output item ratio. Instead, the work function (core block functionality) determines during each execution how many items will be consumed from the input buffer and how many will be produced in the output buffer.

In this work, the goal is to develop new capabilities in DIF for synthesizing efficient SDR implementations. Incorporating dataflow scheduling by generating a DIF file through a new module in GNU Radio will allow us to perform more design space exploration.

SDR blocks that have fixed production and consumption rates can be easily mapped to SDF models. In this case, multiple optimization techniques can be applied at compile-

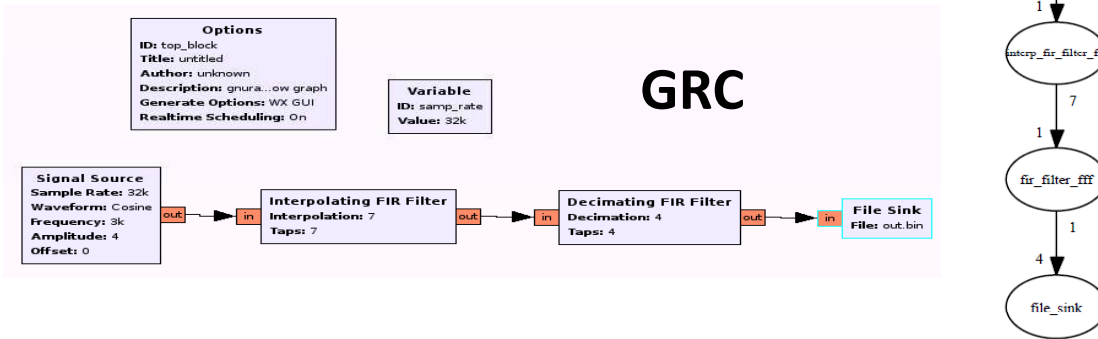


Figure 4.3: Multirate application exported to DIF.

time to meet specific platform constraints (e.g., see [9]). On the other hand, variable rate actors can be mapped to the more flexible *core functional dataflow* model of computation [17]. In this model, every actor has a set of *modes* such that the production and consumption rates are fixed in each mode, but can vary across different modes. Core functional dataflow representations are amenable to *quasi-static scheduling*. This form of scheduling permits dynamic changes in dataflow behavior while fixing significant portions of schedule structure at compile time, which generally increases efficiency and predictability compared to conventional dynamic scheduling approaches. Quasi-static scheduling for core functional dataflow graphs is addressed in [24].

An TDL importer was written to transform system descriptions from python scripts in GNU Radio to equivalent TDL representations. Figure 4.3 shows a snapshot of a system generated by the basic GNU radio graphical user interface and its corresponding graph as generated by the DIF package.

The importer starts by flattening the application graph, and computing a topological sort. Then for every block in the GNU Radio model, a DIF actor and its incident dataflow

edge connections are created and stored temporarily in linked lists. After traversing all of the application graph blocks, the lists of actors and edges are exported to a TDL file using the appropriate TDL syntax.

One difference between the GNU Radio representation format and that of DIF is in the expression of dataflow production and consumption rates. In DIF, these rates are represented as integer numbers of tokens produced and consumed for every actor invocation. On the other hand, in GNU Radio, the rates are managed in terms of individual floating point numbers that represent ratios between corresponding production and consumption rates. Thus, part of the process of converting a GNU Radio representation to TDL involves appropriate conversion of data associated with production and consumption rates.

4.3 GRGPU: GPU Acceleration in GNU Radio

We augment GNU Radio, with a stand-alone GPU accelerated library. The focus of this API is to construct a back-end capable of integrating specialized multicore solutions into a domain specific prototyping environment. This should facilitate the previously described dataflow based design flow, but should also enable these other works to be applied in the field of SDR. Any solution targeting a complex multicore system is unlikely to produce the optimal solution with its first implementation. The ability to quickly generate and evaluate many solutions on a multicore platform should improve the efficacy the approach and ultimately the quality of the final solution. Therefore, this developed approach gives an SDR developer the ability to prototype a GPU accelerated application

and explore its design space fast and effectively.

We have developed a set of GPU accelerated, GNU Radio actors in a separate, stand-alone library called *GRGPU*. GRGPU extends GNU Radio's build and install framework to link against libraries in CUDA. The resulting actors may be instantiated alongside traditional GNU Radio actors, meaning that designers may swap out existing actors for GRGPU actors to bring GPU acceleration to existing SDR applications. The traditional GNU Radio actors run unaffected on the host GPP, while the GRGPU actors utilize the GPU.

When writing a new GRGPU actor, application developers start by writing a normal GNU Radio actor including a C++ wrapper that describes the interface to the actor. The GPU kernels are written in CUDA in a separate file and tied back to the C++ wrapper via C functions such as *device_work()*. Additional configuration information may be sent in through the same mechanism. For example, the taps of a FIR filter typically need to be updated only once or rarely during the execution, so instead of passing the tap coefficients during each firing of the actor (taps sent from *work()* to *device_work()* to the kernel call), they could be loaded into device memory when the taps are updated in GNU Radio. The CUDA compiler, NVCC, is invoked to synthesize C++ code which contains binaries of the code destined for the GPU, but glue code formatted for C++. By generating the C++ instead of an object file directly, we are able to make use of the standard GNU build process using libtool. Even though the original application description was in a different language, the code is wrapped and built in the GNU standard way giving it compatibility with previous and future versions of GNU and GNU Radio.

When a GNU Radio actor is instantiated, a new C++ object is created which stores

and manages the state of the actor. However, state in the CUDA file is not automatically replicated, creating a conflict when more than one GRGPU actor of the same type is instantiated. To work around this issue, we save CUDA (both host and GPU) state inside the C++ actor, which includes GPU memory pointers of data already loaded to the GPU. The state from the GPU itself is not saved inside the C++ object, but rather the pointers to the device memory are. Data residing in the GPUs memory space is explicitly managed on the host, so saving GPU pointers is sufficient for keeping the state of the CUDA portion of an actor.

To minimize the number of host-to-GPU and GPU-to-host transfers, we introduce two actors, *H2D* and *D2H*, to explicitly move data to and from the device in the flow graph. This allows other GRGPU actors to contain only kernels that produce and consume data in the GPU memory. If multiple GPU operations are chained together, data is processed locally, reducing redundant I/O between GPU and host as shown in Figure 4.4. In GNU Radio, the host side buffers still exist which connect links between the C++ objects that wrap the CUDA kernels. Instead of carrying data, these buffers now carry pointers to data in GPU memory. From a host perspective, *H2D* and *D2H* transform host data to and from GPU pointers, respectively.

While having both a host buffer and a GPU buffer introduces some redundancy, it has a number of benefits which make this an attractive solution. First, there is no change to the GNU Radio engine. The GNU Radio engine still manages data being produced and consumed by each actor, so decisions on chunk size or invocation order do not need to be changed with the use of GRGPU actors. Second, GPU buffers may be safely managed by the GRGPU actors. With GPU pointers being sent through host

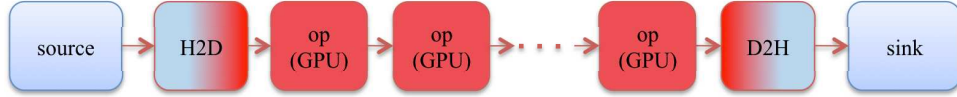


Figure 4.4: GRGPU actors within H2D and D2H communicate data using the GPU memory, avoiding unnecessary host/GPU transfers.

buffers, actors need only concern themselves with maintaining their own input and output buffers. This provides dynamic flexibility (actors can choose to create and free memory for data as needed) or static performance tuning (actors can maintain circular buffers which they read and write a fixed amount of data to and from). Such schemes require coordination between GRGPU actors and potentially information regarding buffer sizing, but the designer does have the power to manage these performance critical actions without redesigning or changing GRGPU. Future versions of GRGPU could provide a designers with a few options regarding these schemes and even make use of the dataflow schedule or other analysis to make quality design decisions. Finally, no extraneous transfers between GPU and host occur. While the host and GPU buffers mirror each other, no transfers occur between them, which avoids I/O latencies that can be the cause of application bottlenecks.

4.4 Empirical Results

We have implemented the proposed workflow, multiprocessor scheduler, and GNU Radio integration. We have experimented with the framework using the mp-sched benchmark [13] shown in Figure 4.5. This benchmark is a synthetic benchmark with a parameterized structure that is representative of a broad class of practical signal flowgraph structures. This benchmark describes a flow graph that consists of a rectangular grid of FIR filters. The dimensions of this grid are parameterized by the *number of stages*

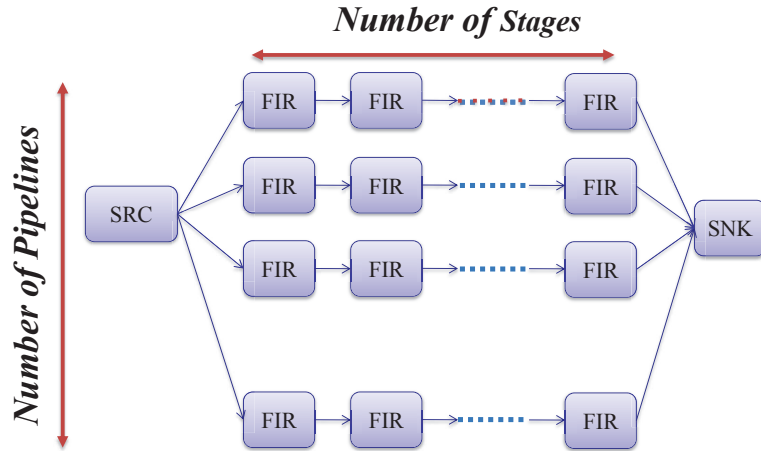


Figure 4.5: MP-sched SDR benchmark.

(*STAGES*) and *number of pipelines (PIPES)*. The total number of FIR filters is thus equal to $STAGES \times PIPES$. In this evaluation, the number of filter taps equals 60. This benchmark represents a non-trivial problem for the multiprocessor scheduler as all actors in different pipes can be executed in parallel. In practical GNU Radio system design, designer input typically eliminates a significant part of the solution space by restricting the allocation of some actors to specific processors.

4.4.1 GRGPU Profile

While GRGPU facilitates a fast path to implementation and consequently the design exploration of how to offload functionality to acceleration platforms, it does contain overheads. To benchmark the overheads, we consider two application types: a lightly loaded application graph and a heavily loaded application graph. Intuitively, the lightly loaded graph isolates the minimum overheads associated with using GRGPU, while the

heavily loaded application indicates the overheads associated with compute intensive kernels. We benchmark these applications on an NVIDIA GTX 260 in a box with a dual core Xeon running at 3.0 GHz.

The structure of both the lightly and heavily loaded application graphs is the same as that shown in Figure 4.4. A single source of samples is introduced into a chain of operations, which for these tests are either all processed by the GPP or all accelerated with the kernel. Samples are chunked into groups of 32k to be processed by the operations to ensure some baseline level of vectorization available to the actor. In the lightly loaded case, we use an operation that has typical CUDA vectorization capabilities, but has a small compute load: constant add. In the heavily loaded case we replace this operation with an operation well optimized for the GPU: a 32K point Fast Fourier Transform (FFT). By cascading the FFTs, we are able to simulate significant compute loads with a single actor type. The GPP implementation of the FFT is the SSE accelerated implementation from the GNU Radio library, while the GRGPU implementation is based on the CUFFT library released by NVIDIA.

The results for the two benchmarks are shown in Figure 4.6. In the lightly loaded case, the GPP implementation outperforms the GPU accelerated case regardless of how many samples are to be processed. Because there is negligible speedup with the constant add kernel, the GPU implementation does not overtake the performance of the GPP. Instead, there is a fixed latency penalty of 200ms. This is incurred from a variety of sources including transferring samples to the device, launching the kernel, and collecting the results. There is also time spent in the GRGPU control logic that coordinates the host thread queues with the device queues. While these penalties appear negligible

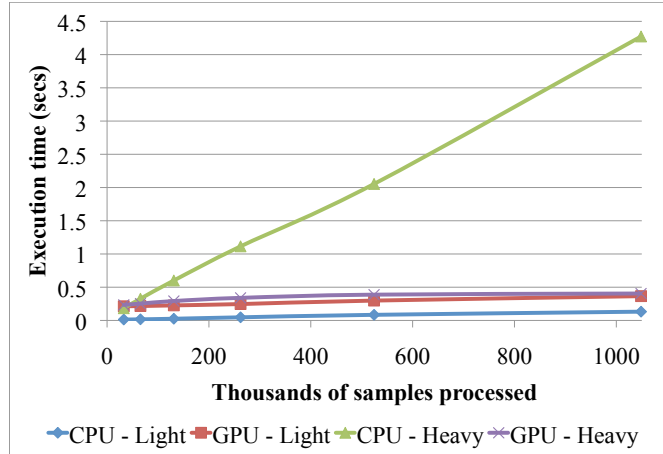


Figure 4.6: GRGPU overhead for various benchmarks.

from a throughput standpoint, the latency penalty is currently high. In the heavily loaded benchmark, the acceleration from the CUFFT library almost immediately makes up for the latency overhead. The GPU approaches 40x acceleration over the SSE accelerated GPP implementation.

4.4.2 Scheduler Empirical Results

4.4.2.1 Test Setup

The input files for the workflow shown in Figure 4.1 consist of the following.

- The SDF application graph described using the dataflow interchange format language.
- Constraints on the blocking factor B , which are intended to be derived from constraints on memory requirements and overall application latency.
- A platform description that includes the available processors types and the number of processors for each type.

- Profile information for every actor (edge) on the available processor types (communication media).

The first stage of the workflow consists of the SDF scheduler, which reads the application SDF graph; calculates the repetition count $q(v)$ for every actor v ; reads the global blocking factor B ; and generates the corresponding BPDAG. Dependency analysis will be performed to set the required values associated with task dependencies. These operations are implemented using the DIF package.

In the second multiprocessor scheduler stage, the scheduler input is generated by setting the run-time of every actor and edge. This run-time will depend on the number of generated tokens per task invocation. The profiles of every actor for a given processor type are stored as tables that are indexed by the number of produced tokens. In this way, the tables are consistent with GNU Radio synchronous block descriptions. In this evaluation, this step is repeated for different blocking factors (B values). The MLP formulation is implemented using the GNU MathProg language [37]. This implementation consists of two parts: the problem description and data section. The problem description specifies the equalities and inequalities mentioned in sec 3.3 in a parameterized format. For every platform and application graph, the data part described in 6.2.1 changes. The MLP problem is solved using the IBM ILOG CPLEX optimizer.

4.4.2.2 Design Space Exploration

To evaluate our approach empirically, we selected a solution to implement within GNU Radio. Figure 4.7 shows the mapping and ordering solution for a 2x5 mp-sched

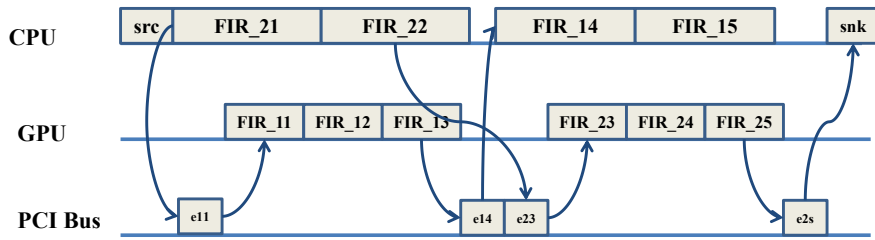


Figure 4.7: Gantt chart for 2x5 mp-sched graph on 1 GPP and 1 GPU.

Table 4.1: Solver results for different mp-sched graphs.

Graph Size		Platform Description		Improvement	Solver
PIPES	STAGES	GPPs	GPUs	%	hours
2	5	1	1	55%	0.01
4	4	2	2	400%	0.49
6	6	3	3	494%	3.94
8	8	4	4	398%	19.6

graph running on a typical modern platform that consists of 1 GPP (Intel Xeon CPU 3GHz), 1 GPU (a NVidia GTX 260), and a PCI bus for a blocking factor $B = 2048$. According to the model, this is a 55% performance improvement over an all-GPP implementation and a 19% improvement over an all-GPU implementation. To validate the model result, we implemented this design within GNU Radio using profiling for latency per token and ensuring accuracy within 6 decimal places to the existing solution. With GRGPU, our solution provided a 39% performance improvement over our empirical results for an all-GPP solution, and a 21% improvement over an all-GPU solution. For this level of vectorization ($B = 2048$), using both a GPU and GPP in the implementation provides the best results, as the model indicates.

Figure 4.8 shows a graph of latency per iteration for different vectorization levels. From the characteristic curves of the GPP and GPU implementations of the FIR actor, the GPU is selectively used when I/O latency bound, but more heavily used when sufficient

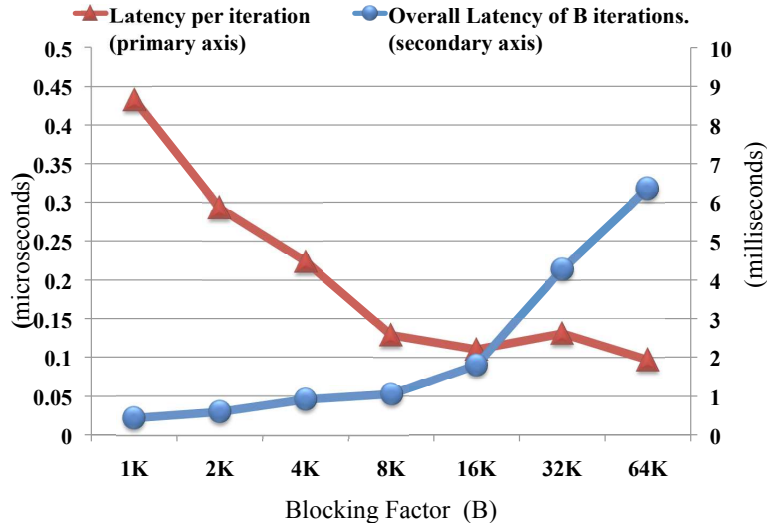


Figure 4.8: Design space for a 2x5 mp-sched graph on 1 GPP and 1 GPU for different blocking factors.

vectorization makes the problem compute bound for the GPU. Using this design space graph, the designer can start by choosing the maximum allowable latency of the DSP application, and then pick the design point that provides the maximum throughput that can be supported for this latency.

Table 4.1 shows the solver running time for different mp-sched graphs on various platforms. As GRGPU backend is the first attempt to implement SDR systems on heterogeneous platforms, we have added the amount of improvement of the scheduled mp-sched graphs over the existing homogeneous GPP implementation in GNU Radio. For the reported solver time, the *solution gap* ranges from 17% for the 4x4 graph to 67% for the 8x8 graph. By the solution gap, we mean the difference between the generally non-realizable results obtained from the real-valued solutions produced by the solver, and the practical results obtained from the corresponding integer valued solutions (derived by rounding the solver solutions). In these experiments, the MLP solver was executed on an Intel Core 2 Duo processor at 3 GHz. It can be shown that the developed MLP problem formulation

Table 4.2: Evaluation of our MLP formulation on multicore processors.

Experiment	1 thread		4 threads		8 threads	
	Imp.	Gap.	Imp.	Gap.	Imp.	Gap.
mp-4-4-2-2	400%	5.34%	400%	0%	400%	5.5%
mp-6-6-3-3	535%	37.07%	521%	38.32%	546%	35.12%
mp-8-8-4-4	612%	50.07%	574%	52.28%	588%	51.11%

can solve problems with size less than 64 nodes. For larger graphs, different scheduling heuristics can be incorporated with our workflow to find efficient solutions [38].

We also evaluate the performance of our MLP formulation using a parallel version of the IBM ILOG CPLEX solver running on an Intel i7-2600K CPU with a 3.4GHz clock speed. Table 4.2 shows the results of these experiments. The digits in the column labeled “Experiment” signify the numbers of PIPES, STAGES, GPPs, and GPUs, respectively. The results show the best mapping solution obtained in a one day period (i.e., with one day allotted on each run to execute the optimization process). We report the amount of improvement (Imp.) over a baseline GPP implementation, and the solution gap. It is important to note that in these experiments, the improvements in the solutions mainly came from using higher-powered processors, while increasing the number of threads did not provide significant gains for these experiments.

4.5 Related Work and Contribution

Actor implementation on GPUs is discussed in [27]. A GPU compiler is described in order to take a naive actor implementation written in CUDA, and generate an efficient kernel configuration that enhances the load balance on the available GPU cores, hides memory latency, and coalesces data movement. This work can be used in our proposed

framework to enhance the implementation of individual software radio actors on a GPU.

Previous research to reduce the gap between application and multicore processor modeling is reported in [39]. In this work, the authors develop a new programming language that is able to describe SDR systems and their implementation on Single Instruction Multiple Data platforms.

Many models of computation have been suggested to describe DSP systems. In [40], the advantages and drawbacks of various models to describe SDR applications are investigated. Also different dataflow models that can be applied to various actors of an LTE receiver are demonstrated. In [19], a hierarchical dataflow programming approach is suggested to specify SDR graphs, and the Satisfiability Modulo Theory is used to formulate the scheduling problem in order to increase system throughput subject to platform memory constraints.

In contrast to prior work, we begin with applications described in a domain specific environment (e.g., GNU Radio), and allow designers to use their existing optimized libraries alongside GPU-accelerated library elements instead of requiring migration to a new programming model.

4.6 Summary

In this chapter, we have empirically evaluated the workflow proposed in Chapter 3. We started by discussing the advantages and drawbacks of implementing actors on GPUs. For the FIR filtering example, we showed that the benefits of using a GPU come only after a certain amount of vectorization is used. Also we noted that while increasing the

amount of vectorization, actor runtime on the GPU remains constant until all the SP are utilized. Then we demonstrated an automated importer that we have developed in the DIF framework. This importer translates applications written in a popular SDR system design environment (i.e., GNU Radio) to SDF graphs in the DIF tool. This translation allows the application designer to use all of the SDF analysis tools provided in DIF for a translated GNU Radio specification.

We have chosen GNU Radio as our implementation environment in these experiments as it is an open source environment that is widely supported by a large community. We have applied GRGPU as the first heterogeneous back-end that can be integrated efficiently with GNU Radio to support GPUs. We showed how GRGPU is implemented to allow designers to easily swap their actors between GPP and GPU implementations without changing the GNU Radio build structure. We minimized data movement between GPUs and GPPs by saving GPU pointers instead of moving data back and forth between the two kinds of processors.

We profiled the overhead of GRGPU and evaluated the effectiveness of our scheduling workflow using relevant SDR benchmarks. We showed how a system designer can automatically choose an appropriate blocking factor B for an application to satisfy given latency and throughput requirements. Finally, we profiled the solution quality produced by single and multi-threaded IBM ILOG CPLEX solvers on different processors. When the MLP formulation is chosen, paying the price of spending more time to obtain the solution (less than one day for typical GNU Radio applications) is rewarded by getting an optimal solution for an embedded signal processing system that may run for several years or more.

Chapter 5

Partial Expansion Graphs

5.1 Introduction

Many DSP systems, such as LTE communication systems, codecs for video and audio players, and Network Intrusion Detection systems (NIDs), require dynamic adaptation and online reconfiguration for the implemented systems at runtime. In addition to GPUs and GPPs, programmable digital signal processors (PDSPs) have demonstrated major benefits for implementing such real time streaming applications due to their high processing performance and low power consumption. PDSPs are widely used in mobile applications, network trans-coding systems, and base station modems. PDSPs are often shipped with large bodies of optimized libraries and legacy code that carefully utilize the processor pipelines and different hardware accelerators for common signal processing functions. In this chapter, we develop novel model based design and implementation methods for PDSP-based realization of signal processing systems. A preliminary version of this work was presented in [41].

We introduce in this chapter a novel implementation model called Partial Expansion Graphs (PEG). We develop efficient scheduling techniques to efficiently map PEG-based system representations into optimized implementations on PDSP platforms. These techniques are designed to systematically explore and utilize different types of parallelism in DSP applications.

Fully exploring data parallelism in DSP applications can be facilitated by transforming SDF graphs to their equivalent HSDF representations [11]. However, this transformation can lead to an exponential expansion in graph size [42]. Using the PEG strategy, we can avoid generating fully expanded HSDF representations while exposing an amount of parallelism that is carefully tuned with respect to the capabilities of the target platform. Also, within the PEG representation, different types of parallelism — including data, pipeline and task parallelism — can be naturally expressed and seamlessly integrated to facilitate derivation of more efficient solutions.

For an important class of DSP applications, kernel execution times can exhibit significant dependency, and are not known at compile time. This is the case, for example, in communication systems such as LTE, where the modulation schemes are dependent on the channel status. Also, different underlying platform issues such as caching, pipelining and multithreading make it hard to predict execution times accurately. For such systems, a dynamic scheduler that can postpone some scheduling decisions to runtime is useful to enhance efficiency and robustness.

Using our proposed PEG strategy for load distribution helps in implementing dynamic scheduling algorithms that effectively distribute workloads associated with SDF graphs. Furthermore, integration of such algorithms with Real Time Operating Systems (RTOSs) can simplify the realization of working systems and facilitate reuse of previously optimized DSP libraries and legacy code.

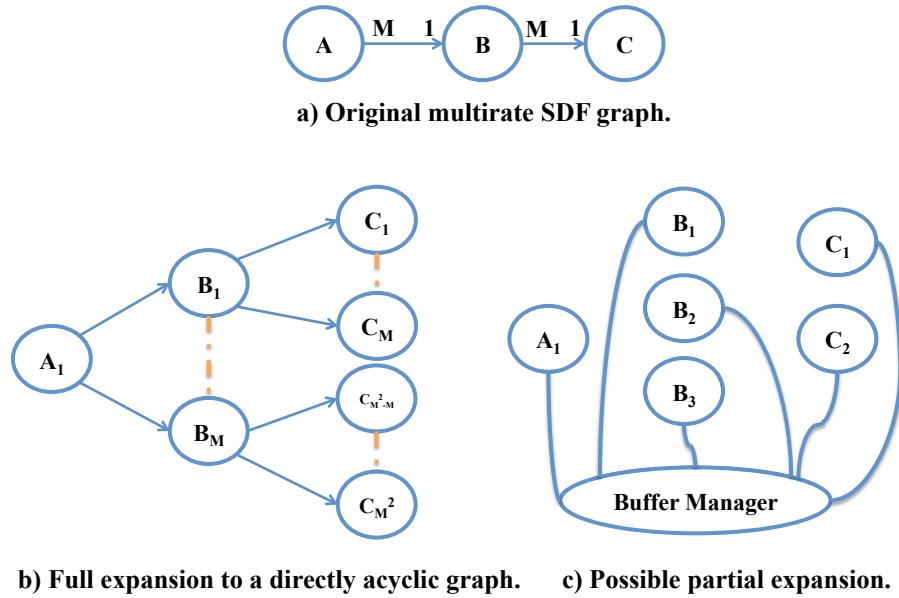


Figure 5.1: Expansion of a multirate SDF Graph.

5.2 Partial Expansion of Dataflow Graphs

Classical multiprocessor scheduling of SDF graphs consists of transforming the input SDF graph to its equivalent HSDF graph or Directed Acyclic Graph (DAG), as shown in Figure 5.1-b. The DAG representation is derived from the HSDF representation by simply removing all edges that have delays (initial tokens) on them. The DAG representation can then be passed to a conventional task graph driven multiprocessor scheduler to generate a schedule for the given SDF graph onto the available processors [43]. Considerations can be incorporated so that the interprocessor communication costs associated with the removed edges (edges with delay) are taken into account in construction of the generated schedule (e.g., see [44, 45]). This technique suffers from two major disadvantages.

First, for a graph that contains multirate edges, the number of generated vertices and edges in the HSDF graph can grow exponentially as shown in Figure 5.1-b, which

has $M^2 + M + 1$ vertices. If this construction is extended to $n \geq 3$ actors (i.e., as a chain of actors connected by edges having production and consumption rates of M and 1, respectively), then the number of HSDF graph vertices exceeds M^n . Considering the typical number of processors in contemporary platforms, such excessive expansion of the graph representation complicates the job of the multiprocessor scheduler, and may not always lead to better solutions. Second, for applications in which schedules must be recomputed or adapted at run-time, it is difficult to dynamically manage an HSDF representation in terms of memory space and execution time overhead. Such dynamic schedule management is important, for example, when actor execution times exhibit significant run-time variation. However, the overhead of performing such management on an HSDF representation can be prohibitive, thereby forcing use of inflexible static schedules that are not matched to execution time dynamics, and perform poorly.

5.2.1 Formal Definition

Intuitively, a *partial expansion graph (PEG)* $G_p = (V_p, E_p)$ is an undirected graph that is used for scalable mapping of an SDF graph G onto a programmable platform. The vertex set $V_p = I_p \cup \{BM\}$, where I_p corresponds to a set of *instances*, which execute groups of successive actor firings, and BM is a special inter-vertex coordination actor called the *buffer manager* of the PEG. The edge set of the PEG is defined as $E_p = \{\{BM, i\} \mid i \in I_p\}$ — in other words, an (undirected) edge is connected between the buffer manager and every other vertex in the PEG.

Each actor v in G corresponds to a set I_v of N_v instances in G_p ($I_v \subset I_p$), where the

mapping $f_v : V \rightarrow N_v$ can be viewed as a design parameter of the PEG. Given a PEG, and a vertex v in the corresponding SDF graph G , N_v is referred to as the *instantiation factor* of v . In general, N_v is positive-integer-valued. However, if v has *state* (internal actor variables whose values persist across firings), then N_v is always equal to 1.

Figure 5.1-c shows one possible partial expansion graph for the SDF graph of Figure 5.1-a. In this example, the original actors A , B and C , which satisfy $q(A) = 1$, $q(B) = M$, and $q(C) = M^2$, are expanded in G_p to $N_A = 1$, $N_B = 3$ and $N_C = 2$ different groups of instances. Such expansion (or consolidation, when viewed in terms of the underlying HSDF graph) could be done, for example, for reasons related to load balancing, as discussed later in this chapter.

The partial expansion graph naturally expresses three fundamental forms of parallelism in DSP applications — data, task and pipeline parallelism — in a scalable manner. Data parallelism is represented when $N_v > 1$; task parallelism is shown when an actor has more than one output edge; and pipeline parallelism is realized by inserting delays between the source and the sink of the application, as illustrated with edge e_{10} of Figure 5.4-b. In the latter case, multiple iterations of the graph can be executed concurrently.

In contrast to the original SDF graph G , the PEG representation relates the actors and edges in G to a particular multiprocessor implementation. The set of instances that correspond to a given actor in the PEG can execute concurrently on the available processors. Therefore, in a platform with C cores where every instance is mapped to a different core, we will have $N_v \leq C$. Associated with every PEG instance, there is a unique kernel (software code block) that executes the code provided with the corresponding SDF actor v . A call to this kernel is called an *activation* of the associated PEG instance. An

activation can be viewed as a vectorized firing of v through the kernel associated with v . Such an activation, which we denote by $vec(v)$, consumes and produces an amount of data equal to an integer multiple of the consumption and production rates of the input and output edges, respectively, for v . This integer multiple is called the *vectorization factor* of the associated activation. Thus, an activation executes a number of successive actor firings that is equal to its vectorization factor. Use of vectorized firings for SDF graphs has been studied extensively, starting with the foundational work of Ritz, Pankert and Meyr [46]. Our PEG formulation provides a novel framework in which vectorization can be integrated efficiently into multiprocessor scheduling contexts.

In the next section, we discuss the special buffer manager vertex BM , which is a critical component in the definition and use of the PEG model.

5.3 Buffer Manager

In Section 5.2 we showed that dataflow along an edge e in the original graph will be implemented and controlled using a state in the buffers manager process. In this section, this functionality of the buffer managers using a Finite State Machine (FSM) implementation is explained. The FSM shows how to implement the peek attribute of an edge in a shared memory system and to capture important platform attributes such as cache line size in a given platform.

When implementing a PEG, the buffer manager vertex BM is mapped into a software process that coordinates the sharing of state across instances that share the same SDF graph actors, and coordinates data transfer between communicating instances that

are mapped to different processors. Within this process, there is a local state that stores information corresponding to the edges in the enclosing PEG G_p , and special data packets, called *PEG messages*, that are associated with instances in G_p . A PEG message for a PEG instance i encapsulates one more pointers to memory blocks that implement the SDF graph edges incident to $A(i)$, where $A(i)$ denotes the SDF graph actor that corresponds to i .

An instance i is activated when it receives a PEG message that contains a number of pointers equal to the of number edges that $A(i)$ is connected to. These pointers have along with them the required information for the positions and amounts of data to be consumed from the referenced input buffer(s), and the data to be produced onto the referenced output buffer(s). A PEG message is sent to the processor that contains an instance to schedule its execution once all buffers associated with the instance are ready. Once the instance finishes execution, it acknowledges BM by sending back the same PEG message. For interprocessor communication across instances, we assume that buffers in shared memory are used.

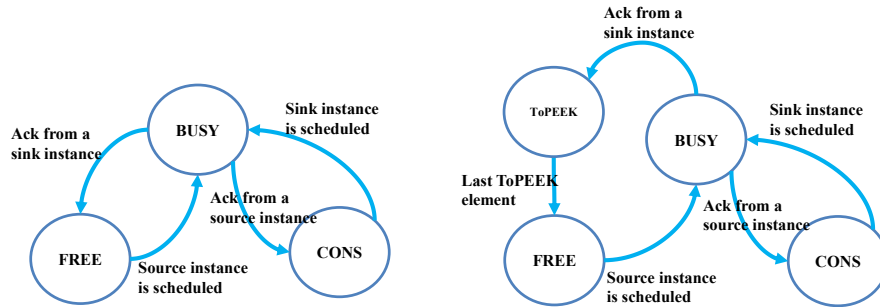
In order to guarantee the right ordering for producing and consuming tokens by different PEG instances, it is assumed that the buffer manager controls a contiguous block of memory, where the size of the block is given by the associated buffer length. This length is proportional to the number of instances N_v , which can be viewed as a parameter that controls the classical trade-off between throughput and buffer space. Instances are designed to run concurrently, therefore the buffer size should be large enough to provide the required space so that all instances can read and write their tokens simultaneously. These buffer lengths can be determined statically based on classical SDF analysis techniques

(e.g., see [44, 45]).

5.3.1 Slot States

In our model of PEG-based implementation, buffers are decomposed into *slots*, where each slot is large enough (in terms of number of bytes) to store some number of tokens that is less than or equal to the corresponding buffer size (i.e., the maximum token capacity of the buffer). Associated with every slot s , there is a *slot status* variable $\nu(s)$, which helps to select among the possible actions that can be performed on the slot. In our implementation, $\nu(s)$ can have four possible values, which are summarized as follows.

- **FREE state:** The bytes that correspond to the slot are free to be overwritten by an instance that produces tokens onto this buffer.
- **CONS state:** The bytes that correspond to the slot are ready to be read by an instance that consumes tokens from this buffer.
- **BUSY state:** The underlying bytes are being either produced onto or consumed from by an instance. This acts like a semaphore on the slot to help avoid race conditions.
- **ToPEEK state:** If the value of the peek attribute (see Section 2.1) of an edge is greater than zero, this status is given to a slot after tokens from it have been consumed. This allows subsequent activations to non-destructively read the corresponding tokens if they fall within the range to be peeked during an activation.



a) Buffer manager FSM for zero peek edges.

b) FSM to support the peek attribute in shared memory architectures.

Figure 5.2: Finite state machines for buffer slot states.

Figure 5.2-a illustrates the finite state machine (FSM) that the buffer manager implements to change the states of slots for edges with zero-valued peek attributes. For a delayless edge, all slots are originally in the FREE state. The buffer manager starts assigning buffer space to source instances and marks the corresponding slots as BUSY. As the buffer is initially empty, the sink actors will be inactive. Upon reception of acknowledge messages that confirm the completion of source activations, the buffer manager sets the produced slots to the CONS state. If the number of slots that have the CONS status is enough to trigger sink instances, the buffer manager sets them to BUSY and activates the sinks. Finally, upon the completion of the sinks, the consumed slots are returned back to the FREE state.

For buffers that have initial tokens (delays), and non-zero peeking values, the delay tokens (i.e., the initial tokens that are placed in the buffer) represent initial values that are consumed or peeked during the first activations of the sink instances. In this case, the buffer manager executes the extended version of the FSM shown in Figure 5.2-b. In this FSM, the buffer manager follows Algorithm 1 to change the status of a slot either to

the ToPEEK or FREE state. A slot is cleared to the FREE state only after guaranteeing that it will not be read in any other subsequent activation. The algorithm also takes into consideration that different sink instances have varying processing times and can finish execution in a different order from that which they were assigned. This algorithm is executed for every buffer state b that corresponds to an edge in order to update a pointer called the *peek_pointer*. This pointer indicates the last token position in the buffer space to be peeked.

Algorithm 1: Implementation of peeking attribute on a shared memory architecture.

```

1: {Initialization}
2: for all local state  $b \in BM$  do
3:    $b.peek\_pointer \leftarrow last\_token\_position;$ 
4: end for
5: {Upon Reception of a PEG message  $msg$  form an instance  $i$ }
6: for all input local state  $b$  of  $i$  do
7:   if  $msg.peeked\_position == b.peek\_pointer$  then
8:      $b.peek\_pointer += msg.n\_consumed\_slots;$ 
9:      $state\_of\_consumed\_slots \leftarrow FREE;$ 
10:    {If previous activations have finished execution}
11:    while  $b.state[peek\_pointer + 1] == ToPEEK$  do
12:       $b.state[peek\_pointer] \leftarrow FREE;$ 
13:       $b.peek\_pointer++;$ 
14:    end while
15:  else
16:     $state\_of\_consumed\_slots \leftarrow ToPEEK;$ 
17:  end if
18: end for

```

5.3.2 Slot Size Selection

The slot size L interacts with key performance factors including run-time buffer manager overhead, cache performance, and allowable vectorization factors. Simulation of alternative slot sizes within the structured framework of PEG-based implementation

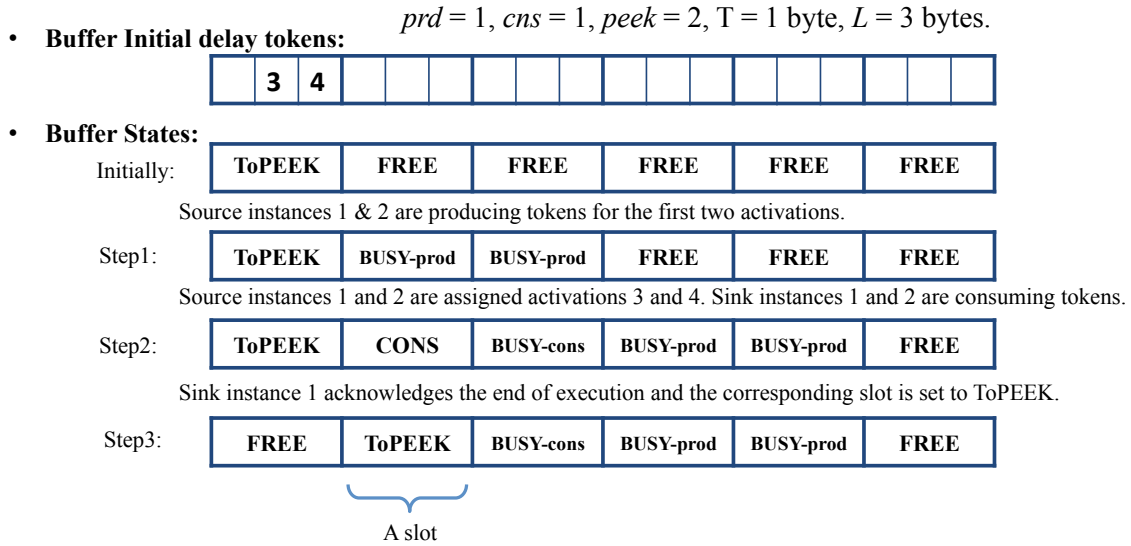


Figure 5.3: Updates of buffer states during graph execution.

can help to determine an efficient slot size for a given application/platform combination. Figure 5.3 illustrates a simulation for executing the FSM in Figure 5.2-b for a slot size of $L = 3T$, where T is the token size (in bytes). Important to notice is that the *peek_pointer* p may not always be aligned with the beginning or end of a slot. In this case, the entire slot will be in the ToPEEK status while p is pointing into the slot, and the slot will be cleared to the FREE state only after p falls outside the range of tokens covered by the slot.

One straightforward way to set the slot size L for a buffer is to set it equal to the minimal possible size, which is the token size (i.e., the number of bytes in a single token). However, this fine granularity choice leads to relatively high buffer manager overhead since frequent changing between slot states is required.

The slot size can be set more efficiently in conjunction with highly vectorized PEG instances — i.e., PEG instances that have high vectorization factors. Applying large vectorization factors ($vec(i) \gg 1$) helps to amortize the overhead of Inter Processor

Communication (IPC), and function calls. Reducing this overhead can be accomplished when s represents the status of multiple tokens. To optimize the setting of L , we set L to be large enough so that the produced tokens from an activation can be consumed as soon as they are ready. Equation 5.1 satisfies this condition:

$$L = \gcd(\text{prd}(e) \times \text{vec}(\text{src}(e)), \text{cns}(e) \times \text{vec}(\text{snk}(e))). \quad (5.1)$$

The slot size L can be useful also in the modeling of application/platform interactions that involve caches. In cache enabled processors, such as the Texas Instruments C64X+ family of PDSPs, operations to load, invalidate and write back memory segments can only take place in terms of cache line sizes (e.g., 128 bytes). This means that the memory that corresponds to a cache line cannot be simultaneously accessed by two processors, otherwise a race condition can occur. This problem can be solved if L is chosen to be a multiple of the cache line size. To guide efficient and correct buffer manager implementation, this cache-based condition, along with Equation 5.1, guides the selection of the slot size parameter to achieve correct and efficient implementation of the buffer manager.

5.4 Dynamic Scheduling

The buffer manager BM schedules an activation only if the corresponding input and output dataflow resources are available. Initially, all of the buffer states are checked and if all ports connected to an instance i are ready, then BM sends a PEG message to schedule the activation of i on its local processor. Once the activation completes execution, the

instance i sends an acknowledgment message to BM . The buffer manager then updates the states of all of the buffers connected to i , and triggers subsequent instances for which all input and output buffers are ready.

From the discussion in Section 5.2, instances can be viewed as threads that execute activations. In this context, we distinguish between two scheduling sub-tasks: *mapping* an instance i to be executed on a processing node p , and *assigning* an activation a to be run by an instance i . These decisions highly affect the final system performance in terms of latency and throughput, and can take place either at compile time or run-time. Also, instances assigned to the same processor can interrupt one another depending on their priorities. Intuitively, the mapping functions determine the maximum load on every processor as the application executes. The separation of mapping and assignment decisions gives the system designer the flexibility to either statically control the system behavior, or postpone some scheduling decisions to run-time. In the experiments reported on in this chapter, we fix the mapping decisions at compile time, and we compare between a) taking the assignment decisions at compile time using classical methods, and b) taking the assignment decisions dynamically at run-time.

Low priority instances can be preempted by higher priority ones if the resources required by the higher priority instances become available. Since the buffer manager always checks for available resources before activation, deadlock is systematically avoided. As execution of a PEG-driven SDF graph implementation proceeds, both data and task parallelism can be exploited, depending, respectively, on whether instances of the same actor or instances of different actors execute at the same time. These two forms of parallelism in general reduce the total latency of the application; however, they may not be sufficien-

t to utilize all of the processors, and some amount of pipeline parallelism (parallelism across distinct graph iterations) may also need to be exploited.

We have developed Algorithm 2 to dynamically assign activations to instances of a given actor. Upon receipt of a PEG message, it will schedule the next activation a to execute if the local processor is idle or if a has higher priority compared to the currently executing thread. The proposed heuristic uses a First Acknowledge First Assign (FAFA) mechanism for the buffer manager BM to assign activations to instances. This mechanism is based on the assumption that the instance that finishes its execution first is ready to receive a new activation.

In Algorithm 2, the “instances allocation variable” i_alloc stores the last activation value that is assigned to every instance. The buffer manager later checks this value to construct the PEG message to be sent when an activation is ready. The utility of this scheduling approach is demonstrated in our experiments presented in Section 5.6.

5.5 Code Generation

Our proposed PEG-based implementation methodology can be realized on a platform consisting of multiple processors that have an RTOS running on them. The basic implementation model assumes that actor instances and the buffer manager will run within threads. The operating system must also provide mechanisms to allocate shared memory space, perform IPC to send and receive PEG messages, and schedule threads on individual cores.

Input to the code generator consists of a library of algorithm kernels that represent

Algorithm 2: FAFA dynamic scheduling heuristic

```
1: {Initialization: Distribute the activations of an actor across the available instances}
2: for all actor  $v \in V$  do
3:   for all instance  $i$  in  $N_v$  do
4:      $i\_alloc[v][i] \leftarrow i$ ;
5:   end for
6:    $last\_act[v] \leftarrow N_v$ ;
7: end for
8: {Upon reception of a PEG msg}
9: for all buffer states  $b$  connected to that actor do
10:  if  $b$  is input to  $i$  then
11:    if  $msg.act == i\_alloc[b.src\_actor][msg.i\_id]$  then
12:       $i\_alloc[b.src\_actor][msg.i\_id] \leftarrow ++last\_act[b.src\_actor]$ ;
13:    end if
14:  end if
15:  if  $b$  is output to  $i$  then
16:    if  $msg.act == i\_alloc[b.snk\_actor][msg.i\_id]$  then
17:       $i\_alloc[b.snk\_actor][msg.i\_id] \leftarrow ++last\_act[b.snk\_actor]$ ;
18:    end if
19:  end if
20: end for
```

actors, a scheduling solution in terms of buffer lengths, amounts of expansion (numbers of PEG instances) for the actors, instance-to-processor mappings, instance priorities, and required implementation attributes (e.g., filter coefficients and values for initial tokens). The output of the code generator is then a complete multi-core software realization of the given SDF graph using the available Application Programming Interfaces (APIs) associated with the targeted RTOS and processing platform. In this dissertation, we implemented our PEG realization using the multithreading and IPC APIs provided by the Texas Instruments DSP/BIOS RTOS.

DSP/BIOS is designed for multiple DSP platforms and is used by many developers to implement sophisticated real time systems. Besides the availability to create threads and execute IPC required for the PEG implementation, it provides mechanisms

to implement different hardware interrupts, periodic functions, general I/O and memory managers needed to run secondary system functions besides its main application. This gives the designers the flexibility of adding additional event triggered tasks to the final implementation as well as ease of migration of generated solution to new platforms.

Different algorithm and architecture attributes can be represented in the application and platform models, and applied by the scheduler, code generator, and other relevant tool components. Some of these attributes are related to dataflow graph or architectural models while others are necessary for directing scheduling-related settings (e.g., buffer lengths) and system functionality (e.g., filter coefficients). We refer to these attributes as “individual element attributes” since they are associated with individual components in an overall system design (e.g., actors, processors, or memory units).

Some of these attributes are originally defined in the application and architecture graphs while others are related to implementation decisions that are derived by the scheduler. In many DSP systems, there is also a need to express *cross attributes*. This type of attribute is in the form of (T, v) , where T is some sort of tuple that can be defined based on any number of elements in the system design, and v represents a value (of some attribute-specific data type) that is associated with the corresponding tuple T .

An example of a cross attribute is an attribute that specifies the estimated running time of an actor on a specific type of target processor. Such a cross attribute could naturally be formulated as $((\alpha, P), t)$ (a two-element tuple (α, P) together with a value t), where α specifies an actor, P specifies a processor type, and t gives the estimated execution time (e.g., in terms of cycle count) of actor α when it runs on a processor of type P .

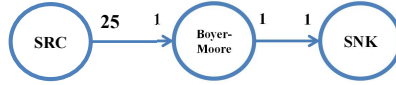
The output of our developed code generator is in the form of embedded C code,

which can be compiled using the Texas Instruments C6000 compiler and linker. For every actor in the system, a header file is generated, which contains information about the number and types of edges that are connected to the actor in the form of integer identifiers.

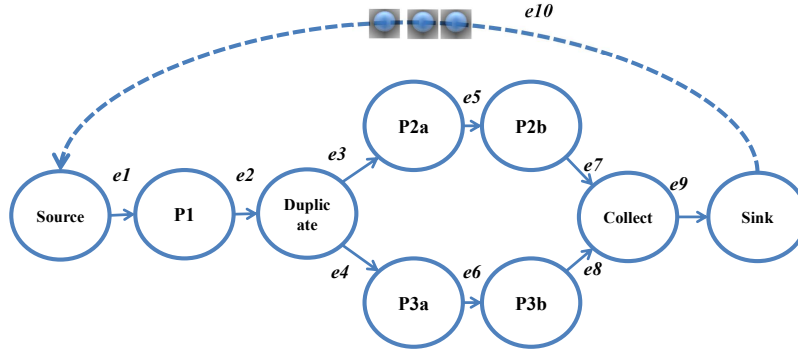
An initialization function and an execution function are generated for every instance in the system. The initialization function for an instance i runs at system start-up to set up a “mailbox” (e.g., MessageQ in DSP/BIOS) for every other instance from which i can receive PEG messages. Upon reception of a PEG message, the thread that wraps an activation is scheduled. This wrapper is generated as the execution function for the associated instance. Within this function, the PEG message is read from the MessageQ, the actor kernel is called, and upon completion of the actor kernel, the PEG message is acknowledged back to the buffer manager.

Similar to the instances, the buffer manager has an initialization function and an execution function. The execution function is a pre-compiled library module that executes the PEG scheduling model in terms of iteratively sending and receiving PEG messages, updating buffer status values, and assigning activations to instances. The initialization function is generated for every scheduler solution to allocate PEG messages, open remote MessageQs, allocate the required buffers space, and allocate the buffer status array. It also sets the dataflow attributes for every buffer state and initializes the associated pointers.

After parsing the given PEG-based scheduling solution, the code generator constructs a folder that contains the required .c, .h, and configuration files for every core in the target platform. The generated multi-core implementation can then be compiled and run for profiling or validation.



a) Intrusion detection application using the Boyer-Moore actor.



b) Pseudo-communication receiver application.

Figure 5.4: PEG model evaluation benchmarks.

5.6 Evaluation

We have implemented and experimented with the PEG strategy by using various SDF-based benchmarks as DSP applications and DIF for dataflow graph specification. The benchmarks were constructed to independently profile different PEG aspects, such as speedups, dynamic scheduling characteristics, and buffer manager overhead. We executed all of our experiments on a Texas Instruments TMS320C6472 six-core PDSP device with 769K-Byte shared memory and 608 K-Bytes of private, configurable L2 memory or cache on each core. In these experiments, the buffer manager is implemented on a separate core, while the other five cores implement the other PEG graph vertices (actor instances).

We carried out three experiments. For the first experiment, we implemented the “pseudo-communication graph” shown in Figure 5.4-b. This application is representative of the front end processing structure for a digital communication receiver. Input data is

received by the actor *Source*. Actor P_1 represents a filter and has a positive-valued peek attribute. The two parallel branches represent possible processing on the I-Q channels and the last two actors, *Collect* and *Sink*, represent the application back-end. The focus of this experiment is to measure the speedups for different sources of parallelism using the PEG strategy — therefore, the actor loads (execution times), and the production and consumption rates of different edges are manually adjusted to generate different graphs.

Figure 5.5 shows possible speedups for different application graph configurations that are derived from the pseudo-communication synthetic “benchmark template” shown in Figure 5.4-b. In the *Task graph*, the execution time loads of the two parallel branches represent 44% of the total execution time load. However, using only task parallelism is not sufficient to achieve a reasonable speedup, while mapping different instances of the pipeline to different cores and combining both task and pipeline parallelism gives a significantly better speedup of 4.15x. The *Data graph* illustrates use of the PEG strategy to avoid full expansion. In this application model, the production and consumption rates in the graph are adjusted such that actor P_1 has a repetition count of 10. However, as there are only 5 cores in the platform, P_1 is partially expanded 5 times and its load constitutes 81% of the total graph load. In this example, a speedup of 4.2x over single core implementation is achieved by using the three different levels of parallelism. Finally, in the *Pipeline graph*, all actors have the same load and good speedups can be achieved only using pipeline parallelism. As there are 9 instances in total, the maximum theoretical speedup is 4.5x and our implementation achieves a speedup of 3.84x.

Our second experiment is summarized by Figure 5.6, which quantifies how the PEG strategy can be useful to achieve speedups for different computation-to-scheduling ratios.

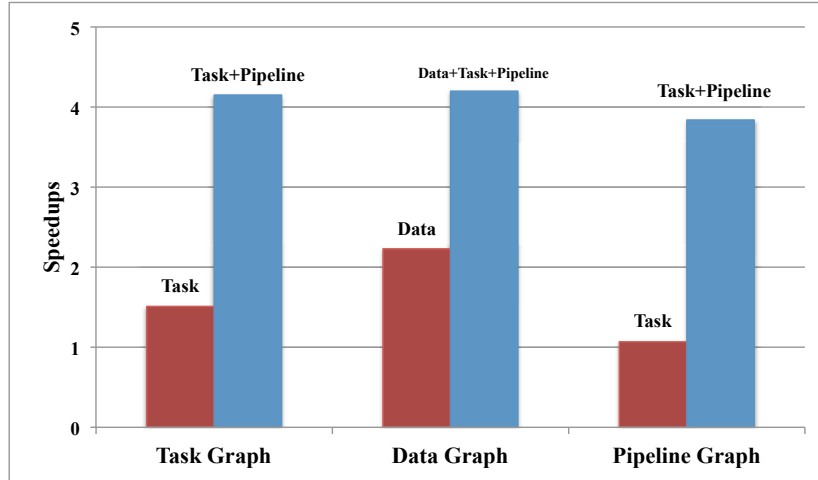


Figure 5.5: Speedups for different sources of parallelism using the PEG strategy.

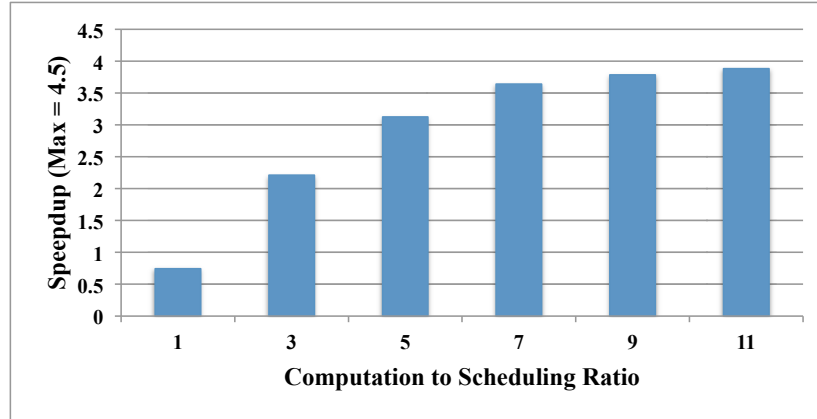


Figure 5.6: Efficiency of the PEG strategy for different computation to scheduling ratios.

This experiment is carried out on the *Pipeline Graph* described above. In this experiment, all of the application actors have the same execution time load λ , which is assumed to be a multiple of the scheduling load S . Here, by the “scheduling load”, we mean the average number of cycles spent in the buffer manager to schedule one activation. The horizontal axis represents the ratio of λ to S for different scenarios. For this experiment, we see that for a ratio of 7:1 (or greater), reasonable speedups can be achieved. For lower ratios, the granularity of computation is too small to adequately amortize the scheduling overhead.

Our third experiment, summarized in Figure 5.7, shows a comparison between the

FABA algorithm for PEG-based dynamic scheduling and the conventional round robin distribution of activations to instances. In this graph, we applied the Boyer-Moore string matching actor shown in Figure 5.4-a [47]. This actor can be used in many important DSP applications — e.g., in network intrusion detection, where incoming packets are searched for sets of malicious strings.

The runtime of the Boyer-Moore actor can exhibit significant variation across different packets — e.g., depending on the target string position when a match is found. The maximum standard deviation in runtime is achieved when the matching string can be located in any position within the packet, and zero standard deviation means that all matching strings are located in the middle of the packet. The horizontal axis shows the normalized standard deviation for a single activation’s runtime compared to the maximum standard deviation. Within the enclosing SDF graph, the Boyer-Moore actor has a repetition count of 25, but it is partially expanded on only 5 cores due to platform limitations. The empirical results for this “embarrassingly parallel” but “unpredictable-load” application show that both scheduling algorithms achieve the same speedup when there is no variability in runtime. However, the PEG-based dynamic scheduler is 35% more efficient than conventional methods when execution times exhibit high variability.

Since the results presented in Figure 5.7 show only the average speedups between the dynamic scheduler and the round robin technique, a paired-t comparison is also conducted to provide a more rigorous statistical analysis about the difference between the two methods. To conduct this experiment, the same packet is used as input for both systems, and the difference between the two methods is calculated. Table 5.1 shows the normalized standard deviation, the difference in the mean speedup over 10 experiments, and the the

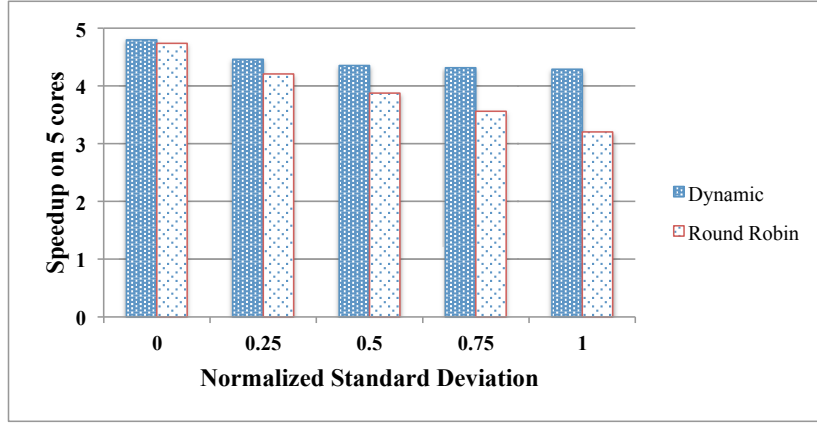


Figure 5.7: Comparison between round robin and dynamic scheduling of activations.

Table 5.1: Average mean and half confidence interval between the dynamic and round robin scheduling techniques.

Normalized Standard Deviation	0.25	0.5	0.75	1
Difference in Mean Speedups	0.26	0.54	0.87	1.08
Half Confidence Interval	0.17	0.09	0.14	0.23

95% half confidence interval using a T distribution. With 95% confidence, our proposed dynamic scheduling method shows superiority over the round robin method.

5.7 Related Work and Contribution

In [48], architecture and application models are presented to support the Algorithm Architecture Adequation (AAA) methodology. In this methodology, a medium-grain architecture description is applied to capture possible parallelism in the underlying platform. An algorithm description is accepted as a DAG annotated with worst case execution times. Scheduling of the algorithm is performed to satisfy real time constraints without usage of a real-time operating system (RTOS).

In [49], the StreamIt compiler is described to utilize different sources of parallelism

on coarse-grained multicore architectures. The input application is re-written using the StreamIt language and describes an SDF graph using a subset of SDF modeling semantics. Different graph transformation techniques are explored to amortize the computation to communication ratio for different applications. Data parallelism is achieved by using a static round-robin actor that distributes the loads on the given processors.

In [50], a comparison is provided between static and dynamic scheduling. The objective in this comparison is to profile and identify the sources of overhead in a dynamic scheduler that is implemented for a certain model-based programming approach. In contrast, in this dissertation we develop an implementation framework for dynamic runtime applications that builds on DSP RTOS technology. Potentially, insights from the comparison in [50] can be applied to further improve the performance of our implementation framework.

An adaptive compilation framework that can track dynamic changes in the architecture is presented in [51]. Low overhead runtime compilation is introduced to accommodate possible changes in resources in terms of the number of processors at runtime. The system is originally compiled on a virtual platform using a higher number of cores and an adaptation heuristic is implemented to remap the system onto fewer cores.

Online nested task parallelization depending on data set loads is discussed in [52] and [53]. In these works, algorithms are presented to dynamically split an actor and explore more parallelism depending on the actual actor load at runtime.

Scheduling adaptive DSP applications such as LTE is presented in [54]. In this work, real time calculation of an efficient schedule is performed at runtime. A full expansion of the dynamic subgraph of the algorithm is suggested. Integration of dataflow

semantics with an RTOS is presented in [55].

Perhaps the closest prior work in relation to our PEG approach is the work presented in [56]. Here, Baudisch, Brandt and Schneider suggest mechanisms to provide out of order execution for dynamic dataflow graphs. A *Central Buffer Station (CBS)* is used to handle instance activations. Algorithms in the CBS are inspired by the reservation stations of the Tomasulo algorithm [57]. Operation of the CBS requires special operating system and architecture support, such as certain atomic instructions and mechanisms for ensuring sequential memory consistency. Our work differs from its approach in our emphasis on model based foundations, including our novel PEG modeling formulation, and the flexibility of applying our approach to arbitrary embedded signal processing platforms without the need for any specialized architectural support.

In addition to the previous work, we propose partial expansion graphs (PEGs) as a novel dataflow intermediate representation, and foundation to efficiently and effectively schedule static dataflow applications on multiprocessor platforms without fully expanding the SDF graph (into an equivalent HSDF or DAG representation). The proposed strategy makes use of task, data, and pipeline parallelism in the application graph, and allows the designer to easily port legacy code and optimized libraries to RTOS-based parallel implementations. Our strategy allows different scheduling techniques to be implemented to dynamically adapt scheduling decisions when execution times exhibit significant run-time variations. Also, by integrating PEG-driven scheduling with RTOS-based implementation, we allow seamless coexistence of the main DSP application with any secondary tasks that may be developed without use of dataflow models.

5.8 Summary

In this chapter, we presented the concept of partial expansion graphs (PEGs), which is an implementation model to realize dynamic DSP systems on PDSPs. We showed how this model helps to address two classical problem in SDF scheduling on multicore platforms: 1) exponential HSDF graph expansion, and 2) dynamic execution times of actors. In a PEG, a stateless actor can be partially expanded to a number of instances, where coordination across instances is performed by a special process called the buffer manager.

We showed in depth how the buffer manager maintains the state of each edge in the application graph in terms of units called slots. We presented two platform-aware algorithms to control slot states, and select slot sizes. Depending on the edge type, we formulated two FSMs to control data access in buffers and support “peeking” capabilities in shared memory architectures. We developed methods for dynamic scheduling in the context of the PEG. These scheduling methods apply the first acknowledge, first assign algorithm, and reduce runtimes over multiple activations by balancing the load over different instances.

We discussed how to implement a code generator for the PEG, and how to integrate it with the Texas Instruments DSP/BIOS RTOS. Our code generator accepts as input the application dataflow graph, architecture graph, and representations of designer-specified PEG expansion and mapping solutions. Based on these input components, the code generator automatically produces a running application that can be profiled using the platform-specific simulator, and experimented with or deployed on the targeted PDSP

platform.

In the next chapter, we show to how automatically generate the expansion and mapping solutions for PEGs through the development of novel PEG-driven scheduling techniques.

Chapter 6

Scheduling and Mapping of Partial Expansion Graphs

In this chapter, we present a novel scheduling technique for PEG-based realization of DSP systems. In this technique, the scheduling process is divided into two stages, called *expansion* and *mapping*. We also introduce a method for embedding our proposed scheduling engine in the design loop, and within the targeted embedded platform. Such *embedded schedule evaluation* allows designers to evaluate and search the design space at runtime with high efficiency and accuracy.

6.1 PEG Scheduling

Searching the PEG design space consists of two parts: 1) finding the amount of partial expansion for every data parallel actor, and 2) mapping the instances of the partially expanded graph to the available PDSP cores. We introduce a two-stage scheduler to efficiently carry out such a two-part optimization process. This scheduler is illustrated in Figure 6.1. In the first scheduling stage, the objective is to determine the amount of data parallelism that is to be exploited, taking into account the available cores in the underlying platform, and the costs associated with communication and coordination with the buffer manager. While the first stage is focused on data parallelism, the second stage ensures that the PEG instances are mapped to cores in a manner that efficiently integrates opportunities for exploiting task and pipeline parallelism as well.

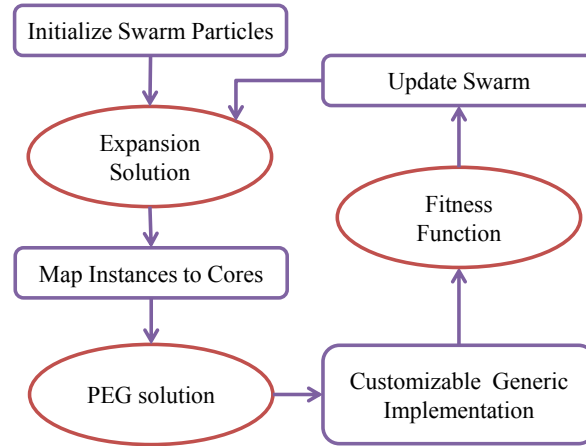


Figure 6.1: PEG-based scheduling workflow.

An important feature of the PEG model is its utility in targeting dynamic applications in which instance execution times are not completely predictable. In such dynamic situations, the overall fitness is a function of many varying and inter-related parameters that are difficult to model. These include the quality of the mapping heuristic, and unpredictable aspects of system behavior (e.g., the system load, operating system, communication overhead, etc.). Therefore, the realization of only one solution can be highly inefficient in some operational situations, and the design space should be explored using different partial expansion solutions. In our two stage scheduler, the amount of partial expansion is explored using a probabilistic search algorithm, and the mapping part is implemented using a heuristic that leverages basic graph properties to explore the available task and pipeline parallelism. Many kinds of probabilistic search techniques can be used to search the partial expansion space for the set of actors in a dataflow graph. In this work, we have formulated and implemented a particle swarm optimization (PSO) technique to perform this search process. Further details on our PSO-based PEG optimization technique are presented in Section 6.2.

When an actor A is partially expanded, its overall average load over a given dataflow graph iteration (i.e., $q(A)$ firings) is distributed across its associated PEG instances. This distribution (assignment of firings of actor A to PEG instances) is done with an effort to balance the load in terms of the estimated execution time (i.e., to balance the total estimated execution time of each PEG instance). Adjustments to the actor load in the PEG can be realized by changing the amount of vectorization applied to the corresponding actor activations. This step of vectorization can significantly reduce the runtime of the instances that correspond to each actor, through the enhanced throughput provided by vectorization, at the expense of added overhead in the buffer manager due to increased buffer sizes.

An essential part of searching the design space using evolutionary algorithms is the assessment of the generated solutions. This step is traditionally performed offline using models of the application and the underlying architecture. Models can vary depending on the accuracy level [7]. Given the dynamic nature of the targeted PEG applications, effective models of offline use cannot be easily derived. For this reason, we develop a generic implementation of the system and add it to the scheduling loop as shown in Figure 6.1. This approach allows us to evaluate the optimization models at runtime, while the system is executing, so that the models can be calibrated and adapted based on the dynamic application characteristics encountered.

This generic implementation can be customized for each solution, as described in Section 6.4. Therefore, by implementing the evolutionary algorithm and the mapping heuristic on the targeted PDSP platform, the actual fitness of every solution is derived based on actual execution characteristics, without the need for a system model.

6.2 Particle Swarm Optimization

Due to the possible dynamic behavior of a PEG application and as the multicore mapping and scheduling problem is NP complete in the general case, a large design space must be explored to determine the amounts of parallelism to apply to the different actors represented in a PEG. On the single actor scale, the amount of expansion should balance the computation-to-scheduling ratio. In other words, too much expansion of an actor can lead to a degradation in performance due to high buffer manager overhead. At the application level, different partial expansions for the application actors lead to different mapping and scheduling solutions that introduce varying amounts of parallelism and overhead in the final implementation.

Particle swarm optimization (PSO) was introduced as a computational optimization method for non linear functions [12]. Originally inspired by how different swarms behave in nature (e.g., swarms of fish and birds), particles in a PSO swarm collaborate by using their best local fitness values as well as the swarm’s overall best value to update their next “positions”. We select the PSO approach to adjust the amount of actor expansion during iterative search for two reasons. First, it is relatively simple for the PSO technique to be implemented on a PDSP or other embedded platform. Second, it matches naturally with the PEG scheduling problem, as illustrated in Section 6.2.1.

6.2.1 PSO Problem formulation

An execution of a PSO consists of multiple *PSO iterations*. Given a dataflow graph with number of actors $n := |V|$, every particle p in our PSO swarm formulation encapsu-

lates a solution vector s_p , where $|s_p| = n$. Each dimension of s_p corresponds to a distinct actor v , and its value is the amount of partial expansion N_v of that actor.

The PSO swarm consists of a set of particles P . All particles $p \in P$ also hold the best solution vector $best_p$ found by them. The best solution found, through the overall PSO process up to a given point in time and by all particles in the swarm, is denoted by $best_S$.

Algorithm 3: Particle swarm optimization
1: Initialize all particles $p \in P$;
2: for all PSO iterations do
3: for all $p \in P$ do
4: $sol = map_instances(s_p)$;
5: $initialize_system(sol)$;
6: $fitness = run(sol)$;
7: $update_best_p(fitness)$;
8: end for
9: $update_best_S()$;
10: for all $p \in P$ do
11: for all $v \in V$ do
12: $calculate_local_inertia_p[v]$;
13: $calculate_global_inertia_p[v]$;
14: $update_s_p()$;
15: end for
16: end for
17: end for
18: return $best_S$;

Algorithm 3 shows how the PSO technique runs within our proposed PEG scheduling workflow. First, current solutions of all particles are initialized randomly. In every PSO iteration, the vector s_p is passed to the mapping heuristic and the fitness of every particle is evaluated. The vector $best_p$ is calculated for every particle, and $best_S$ is calculated for the swarm. The current solution of every particle s_p is updated using two values: $local_inertia_p$ and $global_inertia_p$. $local_inertia_p$ moves the current solution s_p toward-

s its local best solution $best_p$, while $global_inertia_p$ moves s_p towards the swarm's best solution $best_S$. These “inertia values” are calculated by:

$$local_inertia_p[v] = rand() * (best_p[v] - s_p[v]) \quad (6.1)$$

$$global_inertia_p[v] = rand() * (best_S[v] - s_p[v]) \quad (6.2)$$

$$s_p[v] = s_p[v] + C1 * local_inertia_p[v] + C2 * global_inertia_p[v] \quad (6.3)$$

Here, $C1$ and $C2$ are weighting factors that are adjusted to either explore more of the solution space near the particle's best position or to move more towards the swarm's best solution. In our experiments, we adjust both $C1$ and $C2$ empirically to 0.2. Also, $rand()$ returns a random integer between 0 and 9 to help the particle avoid getting stuck in a local minima.

The integer value of every dimension in the solution space is bounded by the minimum and maximum expansion of every actor, which are denoted by $min_inst(v)$ and $max_inst(v)$, respectively. After updating the current solution using Equation 6.3, the expansion of every actor v is rounded to the nearest integer and reset to $min_inst(v)$ or $max_inst(v)$ in case that $s_p[v]$ is inferior to or exceeds these values, respectively. During the initialization phase, two solutions of interest are introduced in the swarm by setting the expansion of every actor v to $min_inst(v)$ and $max_inst(v)$. These two solutions represent the lower and upper bounds on the expansion for the actors. The PSO solution of every particle s_p is then sent to the mapping heuristic to allocate instances to cores, as shown in the next section.

6.3 PEG Mapping Heuristic

After the amount of expansion for every actor is calculated, the mapping heuristic distributes the PEG instances to the available processing cores while targeting two objectives: to reduce the latency of a single graph iteration, and to increase the throughput of the application. In this section, the mapping heuristic is described to show how the three levels of parallelism are used to achieve these objectives. Data parallelism is enabled by allocating the instances of a partially expanded actor to different cores. Similarly, task parallel actors are identified using the *bottom-level* of every actor [38], and their corresponding instances are mapped to be executed on parallel cores. The bottom-level of an actor v is the length of the longest path from v to an exit actor (i.e., to an actor that is not connected to any output edges). The length of the path is the sum of the actors that exist on this path in terms of execution time. Finally, pipeline parallelism is achieved by balancing the load on the available cores.

Without loss of generality, a dataflow graph in our design flow can be viewed as the flow of tokens between one source node and one sink node (if needed, “dummy” source and sink nodes can easily be inserted to enforce this assumption). In cyclic graphs, back edges can be identified by the application designer and removed during the mapping step. Such removal is valid in this context because we focus at this stage on parallelism within a given graph iteration. In this part of the design flow, tokens are distributed onto multiple data and task parallel instances that can run in parallel within one graph iteration. We denote these sets of instances as Delay Parallel Regions (DPRs). A DPR is a general term that depends on the application and platform. A DPR can be seen as a set of instances that

Algorithm 4: PEG mapping heuristic.

```
1: {Set the load of all cores to zero.}
2: for all  $c \in \text{cores}$  do
3:    $\text{load}[c] = 0$ ;
4: end for
5: {Recalculate the weight of every instance.}
6: for all actor  $v \in V$  do
7:   for all instance  $i \in N_v$  do
8:      $\text{weight}[v][i] = \text{weight}[v]/N_v$ ;
9:      $\text{mapping}[v][i] = \text{NULL}$ ;
10:  end for
11: end for
12: while there are unmapped instances do
13:   {Set all the cores to not used for the current DPR.}
14:   for all  $c \in \text{cores}$  do
15:      $\text{used}[c] = \text{FALSE}$ ;
16:   end for
17:    $\text{end\_DPR} = \text{FALSE}$ ;
18:    $\text{begin} = \text{TRUE}$ ;
19:   {Start a DPR mapping.}
20:   while  $\text{!end\_DPR}$  do
21:     {Identify a DPRL or a close task parallel actor.}
22:     if  $\text{begin}$  then
23:        $v = \text{identify\_DPRL}()$ ;
24:        $\text{begin} = \text{FALSE}$ ;
25:     else
26:        $v = \text{identify\_close\_task\_parallel\_actor}(\text{DPRL})$ ;
27:       if  $v == \text{NULL}$  then
28:          $\text{end\_DPR} = \text{TRUE}$ ;
29:          $\text{break}$ ;
30:       end if
31:     end if
32:      $\text{list } L = I_v$ ;
33:     for all instance  $i \in L$  do
34:        $c = \text{least\_loaded\_unused\_core}()$ ;
35:       if  $c == \text{NULL}$  then
36:          $\text{end\_DPR} = \text{TRUE}$ ;
37:          $c = \text{least\_loaded\_core}()$ ;
38:       end if
39:        $\text{mapping}[v][i] = c$ ;
40:        $\text{used}[c] = \text{TRUE}$ ;
41:        $\text{load}[c] += \text{weight}[v][i]$ ;
42:     end for
43:   end while
44: end while
```

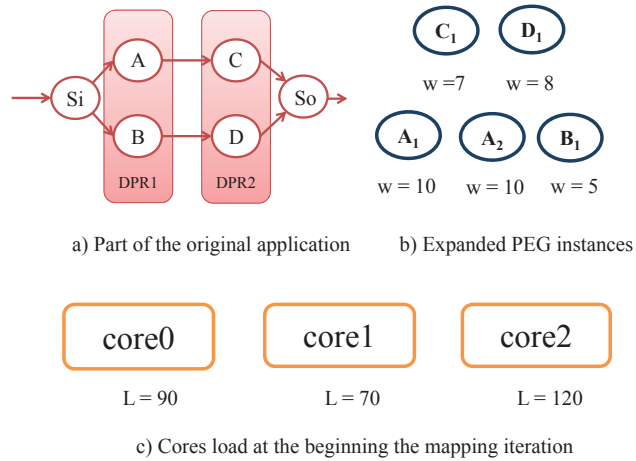


Figure 6.2: Input to the mapping heuristic.

has no data dependency among its corresponding actors. Thus, it is possible to execute the actors in a DPR in parallel if there are enough cores.

Given a platform of three processing cores, Figure 6.2 shows part of a dataflow graph that contains two DPRs. In this example, the average execution time of each actor is represented by the symbol w , and the steady state load on each core at the beginning of this mapping step is represented by the symbol L . The steady state load of a core is the sum of the average execution times of all instances that are assigned to it. The first DPR consists of actors A and B , and the second consists of actors C and D . In this example, A is expanded to two instances A_1 and A_2 .

Algorithm 4 provides a pseudocode specification of our mapping heuristic. Here, scheduling a DPR involves a series of steps, where in each step an instance is chosen to be mapped onto a given processing core.

Each DPR consists of a set of one or more actors M . The associated mapping step allocates the available cores to the sets of all instances that correspond to the actors $I_v \mid v \in M$ in the DPR. The first set of instances corresponds to an actor called the DPR

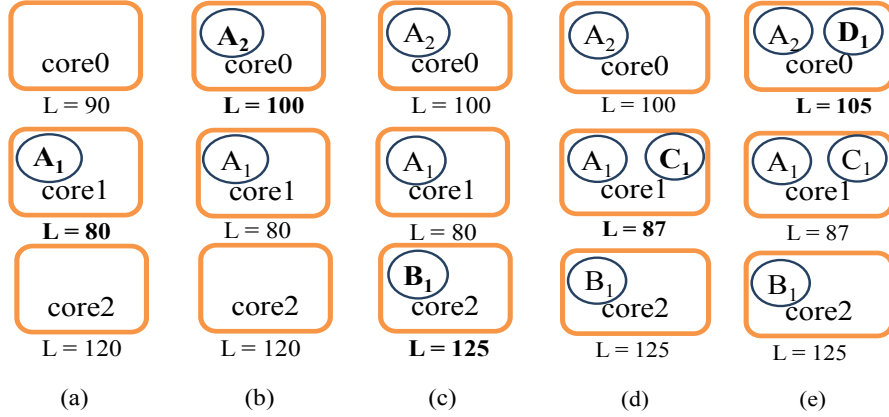


Figure 6.3: Applying the mapping heuristic based on a given graph expansion.

Leader (DPRL), where I_{DPRL} is the set of unmapped instances that have the highest load. After mapping I_{DPRL} , if there is still an unused core, as explained below, a subsequent set of instances is chosen if its corresponding actor is a *close task parallel actor* relative to the DPRL. Identification of a close task parallel actor is achieved by calculating the *bottom-level* of every actor. Two actors are considered as close task parallel if there is no data dependency between them and the difference of their corresponding bottom-levels is minimum over all actor pairs that have no mutual data dependencies.

While mapping every instance, a core is chosen based on two consecutive criteria: first, if none of the current DPR instances is assigned to it, and second, if it has the lowest overall steady state load L of actors. The first criterion guarantees that data and task parallel instances are actually executed in parallel, while the second criterion tries to enhance the throughput of the system in case multiple graph iterations are executed simultaneously.

The process of mapping a DPR terminates depending on the amount of expansion of actors that comprise the DPR and the number of available cores. There are two different

cases that can lead to such termination:

1. Not all cores are used in the given DPR, and there are no more close task parallel actors relative to the DPRL to be mapped.
2. All the cores are marked as used but not all the instances within a set I_v associated with the DPR are mapped. In this case, the DPR mapping process continues until all instances $i \in I_v$ are mapped to cores.

For example, in Figure 6.3, the algorithm starts by identifying A as the DPRL and maps its corresponding set $I_A = \{A_1, A_2\}$ to *core1* and *core0*, respectively. *core1* is chosen first as it is the most lightly loaded core. At this point, as there is still an unmapped core within this DPR, B is identified as the close task parallel actor to A and its set $I_B = \{B_1\}$ will be mapped to *core2*. After every step, the assigned core is marked as “used” in the current DPR. As all cores are used in this DPR, a new DPRL is chosen (e.g., actor C in the shown graph) and the mapping process continues. Therefore C_1 will be assigned to *core1*, and then D_1 will be assigned to *core0*.

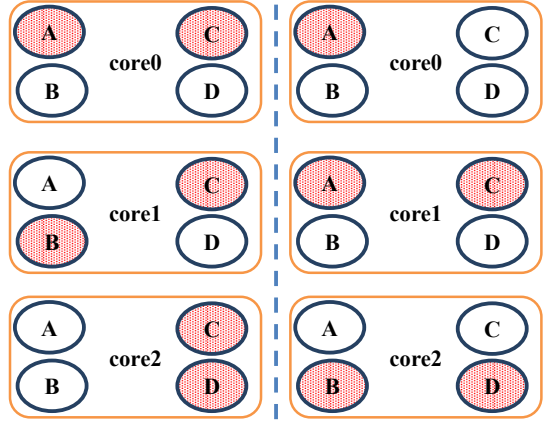
6.4 Generic Implementation

We have implemented the PSO and the mapping heuristic on a state-of-the-art PDSP (see Section 6.5 for specifications on the targeted PDSP device family). The generated PEG scheduling solution is evaluated using a generic implementation that can be customized to cover all possible solutions. The associated code generator, which we have developed, reads the basic mapping attributes from a scenario file. These attributes include whether an actor is expandable or movable. If an actor is expandable, the minimum

and maximum numbers of instances are also given. The “true movable” attribute specifies the set of processing cores that instances of a given actor can be assigned to. The “false movable” attribute indicates that the instances of this actor will not be considered for relocation. These two attributes allow the system designer to use previous experience to limit the design space, prune selected suboptimal solutions, and guide the mapping heuristic by pre-allocating some instances.

Given the scenario file, an actor instance is generated on every core where it can be executed. As actor instances are actually activated upon receipt of a PEG message, the generic solution can be customized by using the messages that correspond to instances used in the solution and discarding the others. In this context, we define the *working PEG messages* as the messages that correspond to used instances in the PEG scheduling solution. For example, suppose that an actor A is movable to any of the cores of a given multi-core platform. Then even if A is not partially expandable, an instance of A will be generated on every core. However, only one PEG message will be used while running the application. For expandable actors, the PEG messages that correspond to working instances are used. By doing so, any generated solution can run and be profiled on the targeted PDSP platform without a need for other implementations. The customizable implementation for the subgraph introduced in Figure 6.2 is depicted in Figure 6.4 with two possible expansions and mappings for actors A and C . The highlighted instances are the ones used in the described implementation.

The evaluation starts by a call to the PSO algorithm to update the partial expansion value for every actor. The PSO solution is passed to the mapping heuristic. Once the solution is calculated, all the application buffers are reset to their initial conditions, and



Solution 1, *A* is expanded in 1 instances, while *C* in 3
 Solution 2, *A* is expanded in 2 instances, while *C* in 1

Figure 6.4: Illustration of a customizable PEG-based implementation.

the working PEG messages are activated. As shown in Figure 6.5(c), the token source actor generates tokens to be consumed in multiple graph iterations. The execution of a given solution starts by activating the token source actor and resetting the instrumentation timers. The buffer manager takes control of running multiple iterations for the application, and then activates the token sink actor and control returns back to the PSO algorithm to collect performance measurements. The PSO stops the design space exploration after a certain criterion is met, such as satisfying a given throughput requirement or reaching a maximum number of PSO iterations.

6.5 Evaluation

6.5.1 Experimental Setup

To test our automatic PEG scheduler, we executed the tasks of the workflow presented in Figure 6.1, and made various measurements to assess the effectiveness of the scheduler. We also validated the correctness of our customizable generic implementation, described in Section 6.4, through different applications and scenarios. In this section, we explain the setup that we have employed in these experiments.

The conducted experiments were run through many steps. The application graphs were written using the DIF language for dataflow graph specification [9], and using the SDF modeling features within DIF. The applications were profiled and the average load of every actor was calculated. In addition to the SDF-based model parameters, all graph elements of the application model were annotated with PEG-related attributes to facilitate our implementation of PEG analysis and optimization techniques using the DIF framework intermediate representation. These PEG-related actor and edge attributes are illustrated in Tables 6.1 and 6.2 respectively. We executed all of our experiments on a Texas Instruments TMS320C6472 six-core PDSP device with 769 K-bytes of shared memory and 608 K-bytes of private, configurable L2 memory or cache on each core.

The input files were passed to the code generator, which generates the generic solution with pointers to the PSO algorithm and the mapping heuristic. The systems were executed in this manner, and performance metrics were collected.

Table 6.1: PEG scheduling attributes for actors.

Parameter name	Description	Type
<i>function_name</i>	The actor call function	string
<i>max_vectorization</i>	Maximum vectorization	integer
<i>movable</i>	Actor can be re-mapped	Boolean
<i>min_inst</i>	Minimum partial expansion	integer
<i>max_inst</i>	Maximum partial expansion	integer
<i>load</i>	Average actor load in cycles	integer
<i>number_of_cores</i>	Cores that can have an instance of that actor	integer

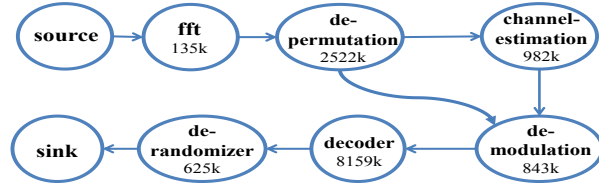
Table 6.2: PEG Scheduling attributes for edges.

Parameter name	Description	Type
<i>buffer_length</i>	Buffer length in tokens	integer
<i>min_state_length</i>	Minimum value for the state in bytes	integer
<i>token_type</i>	The type of tokens that the buffer holds	string
<i>back_edge</i>	The actor is a back edge in cyclic application	Boolean

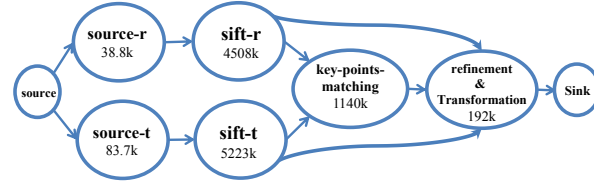
6.5.2 Benchmarks

Several benchmarks are used to evaluate different aspects of our PEG-based automated scheduler. The three versions of the pseudo-communication graph introduced in Chapter 5 are used to validate the ability of the mapping heuristic to capture different forms of parallelism in the application.

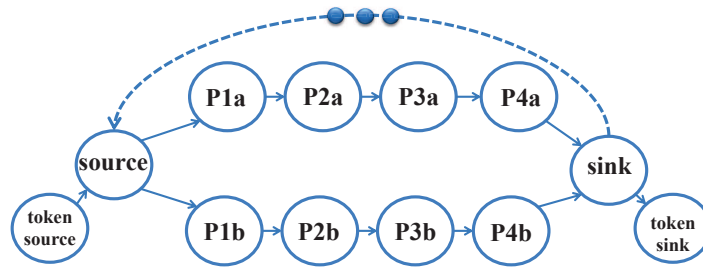
We also constructed two dataflow graph models of practical signal processing applications — a digital receiver benchmark from [10], and an image registration algorithm [58]. For these two applications, the actors’ loads were taken from the papers referenced above. Figure 6.5(a) and Figure 6.5(b) depict the dataflow models that we employed for these two applications. The expandable actors in the image registration application are the `sift-r` and `sift-l` actors. From our profiling results, these two actors consume 87% of the overall application execution time. In the digital receiver, the expandable actor is the `decoder` actor, which consumes 61.5% of the execution time of



(a) Digital receiver benchmark.



(b) Image registration benchmark.



(c) Software defined radio benchmark.

Figure 6.5: PEG scheduling evaluation benchmarks.

each graph iteration.

For all of these applications (the synthetic and practical ones), the solutions generated by our mapping heuristic are used to assign instances to cores. To test the PSO algorithm and its ability to explore the PEG expansion space, we employed the mp-sched 4x2 benchmark, which was introduced in Chapter 4 and is illustrated in Figure 6.5(c). In this graph, each of the actors in the two parallel pipelines can be partially expanded up to 5 instances. This means that the solutions space consists of $5^8 = 390625$ possible expansions.

6.5.3 Results

All of the benchmarks that we experimented with were executed using five cores on the targeted PDSP for the actor firings with the remaining (sixth) core employed by the buffer manager. Therefore, unless otherwise stated, the maximum speedup is 5x compared to the baseline of running the application without any overhead (i.e., buffer manager and multi-core operating system overhead) on a single core.

The same speedups that were obtained manually in Chapter 5 for the three versions of the pseudo-communication benchmark were obtained automatically in our experiments using the mapping heuristic. The mapping heuristic made use of all three types of parallelism (task, pipeline and data parallelism) to reach these levels of performance.

Table 6.3 and Table 6.4 show the best speedups observed in our experiments with their corresponding numbers of instances of the partially expandable actors. The amount of partial expansion in these experiments depends on the number of unrolled iterations of the graph, which is controlled by the number of delay values in the application back edge (i.e., the introduced edge that connects the application's sink to its source, as shown in Figure 6.5(c)). We emphasize that the delays on this back edge are imposed as a synthetic constraint by the PEG-based scheduler to control the amount of inter-iteration parallelism that is exposed.

As the number of partial expansion solutions is relatively small (25 solutions for the image registration application and 5 solutions for the digital receiver), we ran an experiment to generate all possible solutions for each of these benchmarks. We then compared these solutions with the corresponding results derived by our PSO approach. For the im-

Table 6.3: Speedups and expansion of the image registration benchmark

Unrolled Iterations	Runtime (50 iterations)	Speedup	sift-r expansion	sift-t expansion
1	173720	3.20x	5	5
2	133341	4.17x	2	2
3	119976	4.64x	3	4
4	116876	4.76x	3	2
5	118445	4.70x	4	4

Table 6.4: Speedups and expansion for the digital receiver benchmark.

Unrolled Iterations	Runtime(K cycles) (50 iterations)	Speedup	decoder expansion
1	345022	1.92x	5
2	226967	2.92x	4
3	202638	3.27x	3
4	202646	3.27x	3
5	202644	3.27x	4

age registration application, the PSO approach achieved 9% less speedup compared to the best solution found using exhaustive search, while the PSO approach achieved the same speedup as the best solution found using exhaustive search for the digital receiver.

Table 6.5 illustrates the ability of the PSO engine to find effective levels of expansion for different application loads. All of the actors in this synthetic benchmark graph have the same running time and the maximum single iteration speedup is 2.77x. For the given exponential solution space, the optimization engine is able to adjust the expansion of every actor in order to achieve a reasonable speedup without incurring excessive

Table 6.5: Speedups and expansion for the SDR benchmark.

Total load (K cycles)	Runtime(K cycles) (50 iterations)	Speedup	filters expansion
500000	188374	2.65x	All filters→5
50000	29867	1.67x	P1a & P2a→2 All others→3
15000	16663	0.9x	All filters→1

Table 6.6: Mapping heuristic and system-in-the-loop overhead.

System	Expansion	Actors	Instances	Edges	Mapping heuristic	Buffers init.	System init.
Image registration	no exp.	8	8	13	30.7K	40.2K	82.1K
	full exp.	8	16	13	38.8K	44.4K	94.8K
Digital receiver	no exp.	8	8	9	33.4K	21.5K	65.8K
	full exp.	8	12	9	36.7K	21.9K	69.6K
SDR	no exp.	12	12	13	67K	38.4K	116.5K
	full exp.	12	44	13	99.1K	44.3K	155.8K

scheduling overhead. Here, by the “scheduling load”, we mean the average number of cycles spent in the buffer manager to schedule one actor activation. For a high computation-to-scheduling ratio, using the full platform ability to run data and task parallel actors yields the best solutions. On the other hand, for lower computation-to-scheduling ratios, partial or no expansion of actors is required to avoid excessive scheduling overhead. The PSO algorithm can direct its particles to find efficient solutions based on the characteristics of the application and target platform. In this experiment, the PSO swarm consisted of three particles and the best solution was found in 16 PSO iterations.

Table 6.6 reports the following quantities: the runtime of the mapping heuristic; the overhead (time) required for buffer initialization; and the overall system initialization overhead (time) for a single PSO particle. Here, each row represents the runtime for a different benchmark with either no expansion or full expansion (i.e., $max_inst = 1$ or $max_inst = 5$, respectively). The maximum overhead is less than 2% of the runtime of the two practical benchmarks (the image registration and digital receiver applications). Typically, different DSP applications have varying amounts of overall runtime, and it is acceptable to spend some amount of time during execution to realize a more efficient

solution, depending on the required application latency.

6.6 Related Work and Contribution

In [49], a heuristic is presented to adjust the granularity of a streaming graph by fusing actors with relatively small loads, and then separating the fused regions (clusters) based on data parallelism. Software pipelining is then implemented to balance the load on different processors. In [59], an integer linear programming formulation is presented to select the optimal amount of partial expansion to generate balanced pipelined stages. One expansion solution is then passed to a heuristic that schedules the order of execution of instances to processors in order to reduce the application latency. These two methods target applications where offline scheduling is performed and the runtimes of the actors can be accurately modeled. In PEG scheduling, however, we perform the scheduling step using execution on the embedded target platform as integral component, and we target applications where the actors' runtime can vary over time. Thus, our techniques are different in the solutions they provide are able to adapt over time as execution characteristics change.

Distribution of dynamic activations at runtime has also been considered in previous research. The authors of [60] targeted systems implemented using the Cell Broadband Engine (CBE) architecture. For this heterogeneous platform, a dynamic scheduler is suggested for implementation on the PowerPC Processing Element (PPE), with the rest of the system functions being implemented on the eight Synergistic Processing Elements (SPE). Data dependency is extracted by annotating the serial code with compiler prag-

mas, and the main scheduling work is performed by the compiler. A list of ready tasks is maintained at runtime for tasks that resolve data dependency. These tasks are sent later to the SPE for execution. In [61], dynamic scheduling is implemented to explore task and data parallelism for applications implemented on the CBE. The PPE offloads tasks to the SPE if the overall time for execution and communication is reduced. The amount of exposed parallelism is fine-tuned for the RAxML application, which infers large phylogenetic trees. To reduce communication overhead, task parallelism is first explored, and if the processor is under-utilized, data parallelism is explored next.

In contrast to these approaches, our PEG-based scheduling approach targets applications that are developed using model based design techniques — based on dataflow models of computation — as opposed to annotating legacy code. Furthermore, through its operation on coarse-grain dataflow graphs, as opposed to legacy code in procedural languages, our techniques are able to efficiently explore trade-offs and systematic integration among task, data, and pipeline parallelism.

In [7], the authors divide the mapping and evaluation steps into two separate modules. The evaluation is performed using different models for the architecture where different models represent different levels of accuracy. In contrast to this, and many other works that perform host-computer-based mapping evaluation, we perform evaluation of alternative mapping results directly on the targeted embedded platform. This allows for more efficient evaluation of mapping candidates, as well as more accurate measurements for the associated fitness functions.

6.7 Summary

In this chapter, we presented automated scheduling techniques for PEG-based system realization, and presented the results of our experimental testing and validation of these techniques.

We applied PSO methods to the PEG expansion problem, where each particle in the swarm encapsulates specific amounts of partial expansion to apply to the actors in the application. By using our proposed PSO technique, a design tool can efficiently evaluate alternative expansion solutions and explore the design space systematically.

We also proposed a mapping algorithm for partially expanding a graph and applying the results of such expansion in a manner that takes into account diverse sources of parallelism. In this approach, we group data and task parallel instances that can execute concurrently into units that we refer to as “delay parallel regions” (DPRs). Instances that belong to the same DPR are mapped onto different cores to exploit data and task parallelism. Pipeline parallelism is considered by trying to balance the overall load across different processors, and avoiding the introduction of feedback paths (cyclic data dependencies) across processors as actors and instances are mapped.

We demonstrated how to evaluate systems implemented using PEGs on PDSP platforms by developing a customizable generic solution for embedded schedule evaluation. Using this kind of generic solution, accurate profiling under dynamic conditions can be carried out in the scheduling loop. We evaluated our workflow using various benchmarks and we reported the overhead of our PEG-based mapper and our generic solution for embedded schedule evaluation.

Chapter 7

Conclusions and Future Work

With the continuous fast evolution of processing platforms and applications for signal processing systems, there is a growing need to allow designers to re-use prior investments in optimized software libraries tailored to specific types of processors. In this dissertation, we facilitated this important form of reuse by showing how to use model based approaches so that different types of applications can be migrated systematically and efficiently to contemporary multicore processors. We targeted two types of applications: 1) static systems, where the application graph can be analyzed thoroughly at compile time, and 2) dynamic systems, where actor execution times are not predictable and the system needs to be reconfigured at runtime.

7.1 Static Systems

For static systems, we presented a workflow to migrate such systems to heterogeneous platforms consisting of general purpose processors equipped with vector processing units, and general purpose graphics processing units (GPUs) that follow a single instruction multiple thread programming paradigm.

In the broad signal processing domain, static scheduling remains a popular and useful tool in the design process (e.g., see [62]). We applied our statically-oriented workflow to software defined radio (SDR) systems, where designers often attempt to leverage spe-

cial purpose multicore platforms in complex applications, and need to be able to quickly arrive at an initial prototype to understand relevant performance trade-offs. In this context, we have presented a design flow that extends a popular SDR environment called GNU Radio, lays the foundation for rigorous analysis based on formal models, and provides a stand-alone library of GPU accelerated actors that can be applied within existing applications. GPU integration into an SDR-specific programming environment using GRGPU allows application designers to quickly evaluate GPU accelerated implementations and explore the design space of possible solutions at the system level. We also showed how efficient utilization of SIMD cores can be achieved by applying extensive block processing in conjunction with efficient mapping and scheduling.

We systematically decomposed the targeted scheduling problem into two parts: one part that considers vectorization to make use of pipeline parallelism and utilize the available cores of vector- and graphics-oriented processors (e.g., GPU and SSE extensions), and a second part that reduces the latency of multiple dataflow graph iterations on the available set of heterogeneous multicore processors. We formulated the second part using Mixed Linear Programming (MLP).

For coarse grain dataflow graphs, MLP provides optimal solutions in a timely manner for many key SDR applications. In particular, our application on coarse grain graphs, where operations are higher level signal processing operations, such as digital filters or FFT units, allows MLP to operate with reasonable problem sizes that permit solutions within reasonable time frames. We have shown that using actual GNU Radio benchmarks, solutions can be obtained within a 24 hour period. Such a one-day turnaround time is acceptable in many embedded system domains because the implementations are intended

to be fixed or modified only very rarely once they are derived.

We showed how the time required to solve our MLP formulation scales using a 64-node version of the mp-sched benchmark, and how different types of processors can improve the underlying MLP solver's runtime. Additionally, we have developed the overall problem formulation in such a way that a heuristic can be used as an alternative to MLP. In particular, such a heuristic can be applied to yield efficient, although generally sub-optimal solutions with faster turnaround (e.g., for rapid prototyping scenarios). Also, since an input to our workflow is the profile of actor execution times, GPU-targeted actor implementation using CUDA, OpenCL, or other tools can be integrated by using an appropriate simulator or evaluation platform to provide the required, platform-specific execution time estimates.

In contrast to previous studies on GNU Radio benchmarking, which has focused on different homogeneous, multicore GPP implementations, our experiments explored the application of heterogeneous platforms to GNU Radio. Our comparison of the amount of improvement over a baseline homogeneous GPP implementation shows how meaningful speedups of up to 612% on 8 heterogeneous processors can be achieved.

Useful directions for future work for statically-oriented DSP system design (i.e., design involving SDF graphs in which actor execution times are reasonably predictable) include the following.

- Graph transformation techniques for handling cyclic graphs — i.e., dataflow graphs that incorporate feedback.
- Extension of GRGPU to multi-GPU platforms by customizing GPU actors to com-

municate and launch on specific GPUs.

- Experimenting with and optimizing the efficiency of our static system design workflow on different platforms and programming models.

7.2 Dynamic Systems

For dynamic applications, we showed how the strategic formulation and use of implementation models to realize DSP systems can bridge the gap between oversimplifying the system by using only abstract models, and losing the capacity for systematic design space exploration by working only at the level of a customized implementation.

We presented a new intermediate model and associated implementation strategy called the partial expansion graph (PEG). The PEG overcomes conventional problems associated with exponential growth of SDF graph expansions, allowing parallelism to be exposed and exploited judiciously based on levels that match reasonably to the target platform. We have shown that significant speedups on a state-of-the-art multicore DSP platform can be achieved using the proposed PEG methodology, and demonstrated higher speedups, compared to classical round robin scheduling, by using PEG-based dynamic scheduling techniques. We also presented experimental analysis that quantifies various trade-offs associated with PEG-based implementation, and discussed integration with off-the-shelf RTOSs.

Empirical results show that our PEG-based design strategy can 1) achieve significant speedups on a state-of-the-art multicore PDSP platform for static dataflow applications with predictable execution times, and 2) exceed classical scheduling speedups for

applications having execution times that can vary dynamically. This ability to handle variable execution times is especially useful as DSP applications and platforms increase in complexity and adaptive behavior, thereby reducing execution time predictability.

We presented a method for representing and exploring different amounts and types of parallelism in DSP applications that are targeted for implementation on multicore platforms. By using particle swarm optimization (PSO), we are able to explore the design space in terms of controllable amounts of expansion for each data parallel actor. Our novel mapping heuristic can achieve high speedups for many applications by selectively integrating different types of parallelism in application models. The effectiveness of our proposed approach to evaluate different solutions is validated by implementing the PSO, the mapping heuristic, and a customizable generic solution on a state-of-the-art PDSP platform. In addition to providing automated code generation, we have shown how designer experience can be incorporated within the automatically-derived solutions through various graph element parameters, which can be configured, for example, to impose designer-specified constraints.

Useful directions for future work in dynamic DSP system design that are motivated by this thesis include the following.

- Exploration of different techniques to implement buffer managers for runtime coordination of PEG-based implementations. Such techniques could be developed, for example, to provide different kinds of trade-offs, and platform-based customizations.
- Investigation of distributed buffer management techniques as an alternative to the

centralized buffer manager concept developed in this thesis.

- The systematic use within model based design frameworks of PDSP hardware features such as the queue manager subsystem in the recently introduced Texas Instruments KeyStone processor family [63].

Bibliography

- [1] H. El-Rewini, H. H. Ali, and T. G. Lewis, “Task scheduling in multiprocessing systems,” *IEEE Computer Magazine*, vol. 28, no. 12, pp. 27–37, 1995.
- [2] Y-K Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard, “Signal processing on platforms with multiple cores: Part 1 — overview and methodologies,” *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 24–25, November 2009, Guest Editors’ Introduction.
- [3] Y-K Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard, “Signal processing on platforms with multiple cores: Part 2 — design and applications,” *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 20–21, March 2010, Guest Editors’ Introduction.
- [4] *NVIDIA CUDA C Programming Guide*, April 2012, Version 4.2.
- [5] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, “Software synthesis for DSP using Ptolemy,” *Journal of VLSI Signal Processing*, vol. 9, no. 1, January 1995.
- [6] W. Thies, M. Karczmarek, and S. Amarasinghe, “StreamIt: A language for streaming applications,” in *Proceedings of the International Conference on Compiler Construction*, 2002.
- [7] M. Pelcat, P. Menuet, S. Aridhi, and J.-F. Nezan, “Scalable compile-time scheduler for multi-core architectures,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2009, pp. 1552–1555.
- [8] 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.
- [9] 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.
- [10] J. Kim, S. Hyeon, and S. Choi, “Implementation of an SDR system using graphics processing unit,” *IEEE Communications Magazine*, vol. 48, no. 3, March 2010.
- [11] E. A. Lee and D. G. Messerschmitt, “Synchronous dataflow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.
- [12] J. Kennedy and R. C. Eberhart, “Particle swarm optimization,” in *Proceedings of the IEEE International Conference on Neural Networks*, November 1995.
- [13] E. Blossom, “GNU radio: tools for exploring the radio frequency spectrum,” *Linux Journal*, June 2004.

- [14] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.
- [15] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.
- [16] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs using the token flow model," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.
- [17] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.
- [18] 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.
- [19] Y. Lin, M. Kudlur, S. Mahlke, and T. Mudge, "Hierarchical coarse-grained stream compilation for software defined radio," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis of Embedded Systems*, 2007, pp. 115–124.
- [20] V. Marojevic, X. R. Balleste, and A. Gelonch, "A computing resource management framework for software-defined radios," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1399–1412, 2008.
- [21] K. Zheng, G. Li, and L. Huang, "A weighted-selective scheduling scheme in an open software radio environment," in *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 2007, pp. 561–564.
- [22] Y.-K. Kwok, *High-Performance Algorithms for Compile-Time Scheduling of Parallel Processors*, Ph.D. thesis, The Hong Kong University of Science and Technology, 1997.
- [23] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real time DSP," in *Proceedings of the Global Telecommunications Conference*, November 1989.
- [24] W. Plishker, N. Sane, and S. S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, April 2009, pp. 111–116.
- [25] W. Plishker, G. Zaki, S. S. Bhattacharyya, C. Clancy, and J. Kuykendall, "Applying graphics processor acceleration in a software defined radio prototyping environment," in *Proceedings of the International Symposium on Rapid System Prototyping*, Karlsruhe, Germany, May 2011, pp. 67–73.

- [26] G. Zaki, W. Plishker, S. Bhattacharyya, C. Clancy, and J. Kuykendall, “Vectorization and mapping of software defined radio applications on heterogeneous multi-processor platforms,” in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Beirut, Lebanon, October 2011, pp. 31–36.
- [27] Y. Yang, P. Xiang, J. Kong, and H. Zhou, “A GPGPU compiler for memory optimization and parallelism management,” in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2010, pp. 86–97.
- [28] Intel Corporation, *Intel SSE4 Programming Reference*, July 2007.
- [29] S. Ritz, M. Pankert, and H. Meyr, “High level software synthesis for signal processing systems,” in *Proceedings of the International Conference on Application Specific Array Processors*, August 1992.
- [30] D. Applegate and W. Cook, “A computational study of the job-shop scheduling problem,” *ORSA Journal on Computing*, vol. 3, no. 2, pp. 149–156, 1991.
- [31] R. Niemann and P. Marwedel, “Hardware/software partitioning using integer programming,” in *Proceedings of the European Design and Test Conference*, 1996, pp. 473–479.
- [32] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal, “Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs,” in *Proceedings of the Design Automation Conference*, 2007.
- [33] 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.
- [34] A. H. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, “MacroSS: macro-SIMDization of streaming applications,” in *Symposium on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 285–296.
- [35] G. Zaki, W. Plishker, T. OShea, N. McCarthy, C. Clancy, E. Blossom, and S. S. Bhattacharyya, “Integration of dataflow optimization techniques into a software radio design framework,” in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 2009, pp. 243–247, Invited paper.
- [36] 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.
- [37] A. Makhorin, “Modeling language GNU mathprog — language reference, draft edition, for GLPK version 4.34,” Tech. Rep., Moscow Aviation Institute, December 2008.

- [38] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, December 1974.
- [39] M. Lin, L. Dung, and P. Weng, "A cardinal image compressor for capsule endoscope," in *Proceedings of the IEEE Biomedical Circuits and Systems Conference*, November 2006.
- [40] H. Berg, C. Brunelli, and U. Lucking, "Analyzing models of computation for software defined radio applications," in *Proceedings of the International Symposium on System-on-Chip*, 2008.
- [41] G. F. Zaki, W. Plishker, S. S. Bhattacharyya, and F. Fruth, "Partial expansion graphs: Exposing parallelism and dynamic scheduling opportunities for DSP applications," in *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, Delft, The Netherlands, July 2012, pp. 86–93.
- [42] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for DSP applications," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1995, pp. 122–126 vol.1.
- [43] Y. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *Journal of the Association for Computing Machinery*, vol. 31, no. 4, pp. 406–471, December 1999.
- [44] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, CRC Press, second edition, 2009.
- [45] S. Stuijk, M. Geilen, and T. Basten, "Exploring tradeoffs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, July 2006.
- [46] 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.
- [47] R. Cole, "Tight bounds on the complexity of the Boyer-Moore string matching algorithm," in *ACM-SIAM Symposium on Discrete Algorithms*, 1991.
- [48] T. Grandpierre and Y. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings of the International Conference on Formal Methods and Models for Codesign*, 2003.
- [49] 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.

- [50] O. Arnold and G. P. Fettweis, “On the impact of dynamic task scheduling in heterogeneous MPSoCs,” pp. 17–24, 2011.
- [51] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, “Flex-tream: Adaptive compilation of streaming applications for heterogeneous architectures,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2009.
- [52] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, “Lazy binary-splitting: a run-time adaptive work-stealing scheduler,” in *Proceedings of the Symposium on Principles and Practices of Parallel Programming*, 2010.
- [53] O. Certner, Z. Li, P. Palatin, O. Temam, F. Arzel, and N. Drach, “A practical approach for reconciling high and predictable performance in non-regular parallel programs,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2008.
- [54] M. Pelcat, J.-F. Nezan, and S. Aridhi, “Adaptive multicore scheduling for the LTE uplink,” in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, 2010.
- [55] Y. Oliva, M. Pelcat, J.-F. Nezan, J.-C. Prevotet, and S. Aridhi, “Building a RTOS for MPSoC dataflow programming,” in *Proceedings of the International Symposium on System-on-Chip*, 2011.
- [56] D. Baudisch, J. Brandt, and K. Schneider, “Out-of-order execution of synchronous data-flow networks,” in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2012, pp. 168–175.
- [57] R. M. Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units,” *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 225–33, 1967.
- [58] H.-H. Wu, *Modeling and Mapping of Optimized Schedules for Embedded Signal Processing Systems*, Ph.D. thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, 2013.
- [59] M. Kudlur and S. Mahlke, “Orchestrating the execution of stream programs on multicore platforms,” in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2008, pp. 114–124.
- [60] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, “CellSs: a programming model for the Cell BE architecture,” in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.
- [61] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos, “Dynamic multigrain parallelization on the cell broadband engine,” in *Proceedings of the Symposium on Principles and Practices of Parallel Programming*, 2007, pp. 90–100.

- [62] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, 2010.
- [63] Texas Instruments, Inc., *66AK2H12/06 Multicore DSP+ARM KeyStone II System-on-Chip (SoC) Data Manual*, November 2012.