

ABSTRACT

Title of dissertation: **MODELING AND MAPPING OF OPTIMIZED SCHEDULES FOR EMBEDDED SIGNAL PROCESSING SYSTEMS**

Hsiang-Huang Wu, Doctor of Philosophy, 2013

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

The demand for Digital Signal Processing (DSP) in embedded systems has been increasing rapidly due to the proliferation of multimedia- and communication-intensive devices such as pervasive tablets and smart phones. Efficient implementation of embedded DSP systems requires integration of diverse hardware and software components, as well as dynamic workload distribution across heterogeneous computational resources. The former implies increased complexity of application modeling and analysis, but also brings enhanced potential for achieving improved energy consumption, cost or performance. The latter results from the increased use of dynamic behavior in embedded DSP applications. Furthermore, parallel programming is highly relevant in many embedded DSP areas due to the development and use of Multiprocessor System-On-Chip (MPSoC) technology. The need for efficient cooperation among different devices supporting diverse parallel embedded computations motivates high-level modeling that expresses dynamic signal processing behaviors and supports efficient task scheduling and hardware mapping.

Starting with dynamic modeling, this thesis develops a systematic design methodology that supports functional simulation and hardware mapping of dynamic reconfiguration based on Parameterized Synchronous Dataflow (PSDF) graphs. By building on the DIF (Dataflow Interchange Format), which is a design language and associated software package for developing and experimenting with dataflow-based design techniques for signal processing systems, we have developed a novel tool for functional simulation of PSDF specifications. This simulation tool allows designers to model applications in PSDF and simulate their functionality, including use of the dynamic parameter reconfiguration capabilities offered by PSDF. With the help of this simulation tool, our design methodology helps to map PSDF specifications into efficient implementations on field programmable gate arrays (FPGAs). Furthermore, valid schedules can be derived from the PSDF models at runtime to adapt hardware configurations based on changing data characteristics or operational requirements. Under certain conditions, efficient quasi-static schedules can be applied to reduce overhead and enhance predictability in the scheduling process.

Motivated by the fact that scheduling is critical to performance and to efficient use of dynamic reconfiguration, we have focused on a methodology for schedule design, which complements the emphasis on automated schedule construction in the existing literature on dataflow-based design and implementation. In particular, we have proposed a dataflow-based schedule design framework called the dataflow schedule graph (DSG), which provides a graphical framework for schedule construction based on dataflow semantics, and can also be used as an intermediate representation target for automated schedule generation. Our approach to applying the DSG in this thesis emphasizes schedule construction as a design process rather than an outcome of the synthesis process. Our

approach employs dataflow graphs for representing both application models and schedules that are derived from them. By providing a dataflow-integrated framework for unambiguously representing, analyzing, manipulating, and interchanging schedules, the DSG facilitates effective codesign of dataflow-based application models and schedules for execution of these models.

As multicore processors are deployed in an increasing variety of embedded image processing systems, effective utilization of resources such as multiprocessor system-on-chip (MPSoC) devices, and effective handling of implementation concerns such as memory management and I/O become critical to developing efficient embedded implementations. However, the diversity and complexity of applications and architectures in embedded image processing systems make the mapping of applications onto MPSoCs difficult. We help to address this challenge through a structured design methodology that is built upon the DSG modeling framework. We refer to this methodology as the DEIPS methodology (DSG-based design and implementation of Embedded Image Processing Systems). The DEIPS methodology provides a unified framework for joint consideration of DSG structures and the application graphs from which they are derived, which allows designers to integrate considerations of parallelization and resource constraints together with the application modeling process. We demonstrate the DEIPS methodology through cases studies on practical embedded image processing systems.

MODELING AND MAPPING OF OPTIMIZED SCHEDULES FOR
EMBEDDED SIGNAL PROCESSING SYSTEMS

by

Hsiang-Huang Wu

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 Manoj Franklin

Professor Andrew Harris

Professor Jeffery Hollingsworth

Professor Alan Sussman, Dean's Representative

© Copyright by
Hsiang-Huang Wu
2013

Dedication

This thesis is dedicated to my parents and sister, who support me to pursuit the Ph.D. degree. It is also dedicated to Jie Ni, who has accompanied me to get through all the hard time.

Table of Contents

List of Figures	v
1 Introduction	1
1.1 Key Problems in the Design and Implementation of Embedded Systems	1
1.2 Dataflow Models	4
1.3 Scheduling for Dataflow Models	7
1.4 Design Techniques for Multiprocessor Systems-on-Chip	8
1.5 Contributions of this Thesis	13
2 Rapid Prototyping for Digital Signal Processing Systems using Parameterized Synchronous Dataflow Graphs	16
2.1 PSDF Operational Semantics	16
2.2 PSDFsim	20
2.3 PSDF-based Design Methodology	21
2.4 Hardware Architecture Mapping	23
2.5 Case Study: Phase-Shift Keying	26
3 A Model-based Schedule Representation for Heterogeneous Mapping of Dataflow Graphs	32
3.1 Related Work	32
3.2 Core Functional Dataflow	34
3.3 The Dataflow Schedule Graph Representation	35
3.4 Reference Actors	36
3.5 Schedule Control Actors	40
3.6 Sequential Dataflow Schedule Graphs	41
3.7 Concurrent Dataflow Schedule Graphs	44
3.8 Adaptive Dataflow Schedule Graphs	48
3.9 Experimental Results	51
3.9.1 Simulation Time Improvement	52
3.9.2 Hardware Architecture Mapping from a DSG	55
3.9.3 Application to Software Implementation	58
3.10 Extensions	62
4 Multiprocessor System-On-Chip Implementation of Image Signal Processing Applications	66
4.1 Introduction and Related Work	66
4.2 Overview of Our Contribution	69
4.3 DSG-based Framework for Resource-constrained Implementation of Embedded Image Processing Systems	70
4.4 DEIPS-based Design for Multicore Programmable Digital Signal Processors	72
4.4.1 CDSG Construction	74
4.4.2 Implementation Details	76

4.5	Comparison to Other Dataflow-Based Modeling Techniques for Image Processing	79
4.6	Case Study: Background Subtraction on a Multicore Digital Signal Processor	80
4.6.1	Application Modeling	82
4.6.2	Macro-Pipeline Mapping Structure	84
4.6.3	Fork-Join Mapping Structure	87
4.6.4	Combination of Macro-Pipelined Execution and Fork-Join Parallelism	89
4.7	Case Study: Image Registration on a Multicore Digital Signal Processor	90
4.8	Summary	92
5	Conclusion and Future Work	102
	Bibliography	106

List of Figures

1.1	Shared memory model.	9
1.2	An example of the parallel programming patterns <i>map</i> and <i>reduce</i>	10
1.3	An example of dataflow modeling.	11
1.4	Fork-join model.	12
2.1	A PSDF Downsampler.	19
2.2	An illustration of parameter propagation in PSDFsim.	21
2.3	FPGA/ASIC design flow overview.	23
2.4	Interface and control architecture for a circuit block.	26
2.5	An illustration of subsystem-level hardware mapping.	27
2.6	Finite state machines for (a) a circuit block, (b) a graph controller, (c) consumption and production circuits, and (d) a subsystem controller.	28
2.7	PSDF-based model of PSK modulator and demodulator.	28
2.8	Hardware mapping for modulator.	30
3.1	The internal structure of an RA.	37
3.2	(a) An SDF graph (b) A design example of an SDSG for the single appearance schedule $(A(2B)C)$	45
3.3	The <i>snd</i> and <i>rec</i> actors.	46
3.4	An application graph and a partitioning of the graph across four processors.	48
3.5	A CDSG representation of a multiprocessor schedule that corresponds to the partitioning result shown in Figure 3.4.	49
3.6	(a) A dataflow-based <i>if-then-else</i> construct. (b) An adaptive DSG for this construct.	50
3.7	(a) A PSDF model for a reconfigurable phase shift keying application; (b) an ADSG representation for implementing this application.	54
3.8	Performance comparison between DSG-based quasi-static scheduling, and dynamic scheduling.	56
3.9	RPSK modulator.	57
3.10	Hardware architecture mapping for a DSG.	59
3.11	RPSK system and alternative DSGs.	61
3.12	Outline of software implementation structure for the DSG shown in Figure 3.11(c).	64
4.1	Architecture of the Texas Instruments TMS320C6678L Evaluation Module.	73
4.2	CDSG for Figure 1.3.	75
4.3	Illustration of synchronization for the partitioned dataflow graph of Figure 1.3.	76
4.4	The Concurrent Dataflow Schedule Graph (CDSG) for Figure 4.3.	77
4.5	(a)Temporal GMMs; (b)Spatial GMMs.	81
4.6	LiDE-based application model for macro-pipelined background subtraction.	84
4.7	(a)Macro-pipelined version of Figure 4.6; (b)the corresponding CDSG.	85

4.8	(a)Fork-join version of Figure 4.6; (b) a corresponding CDSG representation.	95
4.9	Integrated macro-pipelining and fork-join parallelization for multicore BG subtraction.	97
4.10	Experimental results for multicore BG subtraction: (a)–(c) show the input frames; (d)–(f) show the corresponding background subtraction results.	98
4.11	A dataflow graph model of Scale-Invariant Feature Transform (SIFT)-based image registration.	99
4.12	Partitioning of Figure 4.11 for mapping onto the target multicore platform.	99
4.13	The CDSG derived for Figure 4.12.	99
4.14	Heap usage measurements for SDSG 1.	100
4.15	Heap usage measurements for SDSG 2.	100

Chapter 1

Introduction

1.1 Key Problems in the Design and Implementation of Embedded Systems

Widely used in many different application areas, embedded systems are usually dedicated to specialized purposes, and need to satisfy diverse constraints. In recent years, as tablets and smart phones have emerged and spread around the world, an increasing variety of applications are developed for such devices, and as such applications evolve, they often require higher and higher levels of performance.

Such a trend towards higher performance embedded processing makes Multiprocessor System-On-Chip (MPSoC) technology increasingly attractive across many embedded system domains. At the same time, the rapid evolution of tablet and smart phone technology accelerates the product cycle in associated embedded systems. The need to manage such change with high productivity and low cost makes designers turn to abstract models for design process management and application development. Because embedded processing technology and sensor technologies are evolving rapidly, such abstract models need to be independent of the underlying hardware. Furthermore, the increasing levels of adaptivity and reactivity required in embedded applications call for embedded system implementations that can efficiently achieve dynamic reconfiguration, and associated

scheduling and parallelization of application programs.

Due to time-to-market pressures and standards (e.g., standards for media formats or data communication) that evolve concurrently with product evolution, ASIC cannot provide the design agility, turnaround time, and flexibility required for many embedded application areas. In such scenarios, reconfigurable devices, such as FPGA, can be used as for prototyping and experimentation because of their regular structure with lower price [58]. Furthermore, as the technology for such reconfigurable devices advances, it can provide solutions beyond the needs of the prototyping and experimental stages, and can be competitive, for certain application areas, in terms of the requirements of final implementations [51]. Software-Defined Radio (SDR) is an example of an application area where FPGA are employed extensively [7].

Dynamic reconfiguration allows for reconfiguration of processing structures at runtime, while an application is executing. Through support for dynamic reconfiguration, embedded systems allow customization of hardware structures both statically and at runtime, thus allowing streamlining of processing configurations in response to application requirements or data characteristics that are not known at design time. In addition to allowing for dynamic changes in system functionality, dynamic reconfiguration, when carried out effectively, can enhance performance, resource utilization, and energy efficiency (e.g., see [25]).

Dynamic reconfiguration in embedded systems provides valuable flexibility and opportunities for enhanced efficiency, but also leads to increased complexity in terms of design analysis and optimization. Existing approaches focus primarily on either abstract models with the capability of expressing dynamic reconfiguration at a high level or

techniques for low-level, platform-specific implementation. While both of these areas of advancement are important, there is an increasing need to bridge the gap between them in order to better realize the potential of dynamic reconfiguration technology.

Scheduling is of critical importance to embedded systems, and effects key metrics, including latency, throughput, memory management efficiency, and power consumption. Furthermore, scheduling disparate applications running on limited resources is very challenging. Past research has focused extensively on static scheduling, and has provided a variety of efficient solutions for different kinds of requirements or constraints (e.g., see [51]). However, research on dynamic scheduling, particularly in the context of efficient embedded system implementation, is relatively less mature.

Scheduling has been studied extensively in the context of dataflow-based modeling of DSP systems. Dataflow graph scheduling involves assigning actors to processors, and sequencing subsets of actors that share common processing resources. For dataflow scheduling of DSP systems, a “processor” in this context is typically taken to be a hardware resource on which execution is time-multiplexed by actors that are assigned to it. In addition to ensuring that dataflow graph dependencies are respected, scheduling is often geared towards exploiting parallelism (performance improvement) and efficient memory utilization (buffer management). Given the fundamental role of scheduling in dataflow-based design flows, and its heavy impact on key implementation metrics, a wide variety of techniques has evolved over the years and continues to evolve for scheduling DSP dataflow graphs. Such techniques target objectives such as buffer optimization [2], joint code and data minimization [8], quasi-static scheduling [20], adaptive scheduling [6, 55], and throughput optimization [18].

As multicore processors are deployed in an increasing variety of embedded systems, effective utilization of multiprocessor system-on-chip (MPSoC) technology becomes critical to developing effective embedded implementations. Besides the challenges of parallelizing embedded system applications, the exploitation of parallelism also increases the complexity of resource management. Characteristic of embedded systems, limited resources become a major bottleneck of performance improvement and of the management of overall design constraints, especially in computationally-intensive application areas, such as multidimensional signal processing. To help designers expose and exploit parallelism in tightly resource-constrained design scenarios, systematic methods are needed for integrated modeling and exploration of associated implementation trade-offs.

1.2 Dataflow Models

Dataflow modeling is widely used in the design and implementation of DSP systems (e.g., see [7]). A dataflow graph is composed of actors (nodes) and edges, which represent computational tasks and data dependencies, respectively. The complexity of computations represented as dataflow actors can have arbitrary granularity — e.g., ranging from a few lines of code in a high level language to hundreds or thousands of lines.

As a distributed model of computation, dataflow involves local control through the “firings” (discrete units of execution) of individual actors. An actor starts a firing when an enclosing scheduler or hardware controller dispatches it for execution, and sufficient data is available at its input ports. Such an asynchronous, concurrent model of computation allows naturally for simultaneous execution of multiple actors if sufficient input data and

sufficient resources are available [40].

As the complexity of DSP systems increases, we see a steadily increasing demand for more powerful dataflow models and associated techniques for analysis and optimization. Synchronous Dataflow (SDF), proposed in [35], is the first dataflow-based model of computation to gain broad acceptance in DSP design tools, and many useful techniques, such as efficient scheduling and buffer size optimization, have been developed in the context of SDF (e.g., see [8]).

Although an important class of useful DSP applications can be modeled effectively in SDF, the expressive power of SDF is restricted since SDF imposes a restriction of *static communication behavior*, which actors must adhere to. In particular, for any given input port p_i of an SDF actor, the number of data values (*tokens*) consumed from p_i is constant across all firings of the actor, and similarly, the number of tokens produced by the actor on each of its output ports is constant. In other words, SDF actors cannot produce and consume varying amounts of tokens on their output and input ports.

As the need to model dynamic communication behavior has increased, due to the increasing levels of flexibility and dynamics in signal processing applications, many extensions or alternatives to the SDF model have been proposed. In general, an important objective for these models is to accommodate a broader range of applications while maintaining a significant part of the compile-time predictability that is offered by SDF.

Cyclo-static dataflow [9], scenario-aware dataflow [55], and Enable-Invoke Dataflow (EIDF) [46] are examples signal processing oriented dataflow models of computation that have been designed for increased expressive power. An extensive survey of such modeling techniques and their associated trade-offs is provided in [51]. In this thesis, we

target a specific form of dataflow modeling referred to as Parameterized Synchronous Dataflow (PSDF), which offers valuable properties in terms of modeling systems with dynamic parameters, supporting efficient scheduling techniques, and natural integration with popular SDF modeling techniques [5].

PSDF can represent cyclo-static dataflow by making dataflow-related parameter variations occur according to periodic patterns. Compared to EIDF, PSDF has lower expressive power overall, but is equipped with streamlined scheduling techniques for the subclass of application models that are amenable to PSDF semantics. Compared to scenario-aware dataflow, PSDF can be viewed as having a more strict separation between data and parameters, which facilitates symbolic scheduling techniques based on parameterized looped schedules.

PSDF is based on *parameterized dataflow*, which is a meta-modeling technique that can significantly improve the expressive power of an arbitrary dataflow model that possesses a well-defined concept of a *graph iteration* [6]. Parameterized dataflow provides a method to systematically integrate dynamic parameter reconfiguration into such models, while preserving many of the original properties and intuitive characteristics of the original models.

The integration of the parameterized dataflow meta-model with SDF provides the model of computation that we refer to as PSDF. Efficient *quasi-static scheduling* techniques have been demonstrated previously for PSDF specifications [5]. Here, by quasi-static scheduling, we refer to a general approach to scheduling in which significant portions of schedule structure are fixed at compile time, while some amount of run-time schedule adjustment can be made in response to input data or changes in operational

requirements.

Functional DIF is a functional simulation environment, with useful applications to rapid prototyping, for DSP-oriented dataflow models of computation [46]. Functional DIF is based on the EIDF model of computation, which is Turing complete. Functional DIF allows actors whose internals are programmed in Java to be integrated with EIDF-based dataflow graphs that are specified using Dataflow Interchange Format (DIF). DIF is a textual language for specifying dataflow graphs in terms of arbitrary dataflow models of computation [22].

1.3 Scheduling for Dataflow Models

In DSP-oriented dataflow models of computation, applications are modeled as directed graphs, where actors represent computational modules for executing tasks, and edges represent first-in-first-out channels for storing tokens, and imposing data dependencies between actors. Whenever an actor fires, it consumes and produces tokens from its input and output edges, respectively. As discussed previously, scheduling is a fundamental process that must be addressed carefully to derive efficient implementations from dataflow graphs.

As the range of dataflow graph scheduling techniques continues to expand, based on the heterogeneity of application modeling styles and implementation objectives, and the increasing degree of dynamics in applications, it becomes increasingly important to develop a common representation for modeling and working with dataflow schedules. Such a representation is desirable to enable systematic reuse of design tool code, anal-

ysis techniques, and back-end implementation methodologies across various scheduling strategies. Furthermore, a formal representation helps to integrate different scheduling techniques so that they can be mixed and matched across different subsystems of a design based on characteristics and objectives associated with those subsystems.

1.4 Design Techniques for Multiprocessor Systems-on-Chip

The *shared memory* model, illustrated in Figure 1.1, is a popular parallel programming model. The shared memory model is widely used in small scale parallel computers, such as desktop personal computers and laptops. Parallel computing platforms that are based on the shared memory model are called SMP. In SMP, processors (threads) are given flexible access to shared memory, while writing conflicts (i.e., consistency problems) need to be resolved by programmers based on the available architectural support. For example, a write-through cache can update shared memory data as soon as possible, whereas a write-back cache generally takes longer to do so, thereby requiring additional considerations to handle consistency [53]. One of the most popular multiprocessors for embedded system, the ARM Cortex-A9 MPCore, is based on the shared memory model.

Instead of using a centralized concept of memory, the *message passing* programming model assumes that each processor has its own local memory and communicates with other processors through communication packets (messages). Message passing is suitable for large scale parallel computers due to better scalability compared to the shared memory model. However, memory references across different local memories can be difficult to resolve. This makes the approach error prone, and requires more effort in

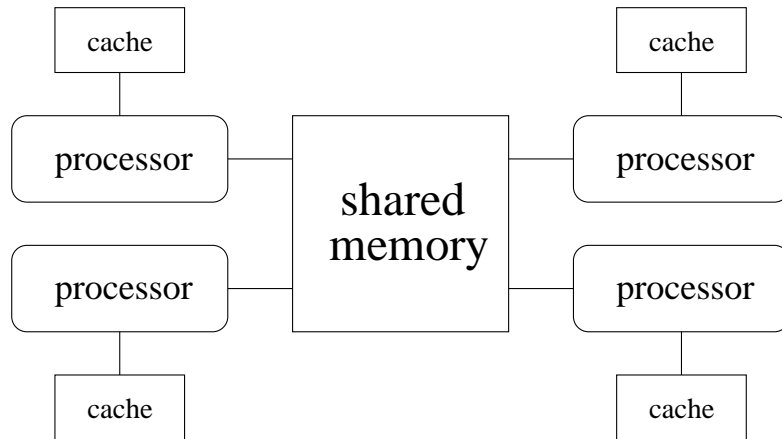


Figure 1.1: Shared memory model.

code maintenance. To support message passing, a library called Message Passing Interface (MPI) has been developed for C- and Fortran-based parallel programming. Furthermore, the implementation of message passing is not restricted to distributed memory systems. That is, message passing can be realized in shared memory systems as well.

In contrast to imperative programming, *skeleton programming* (also called structured programming) provides a different flavor for expressing parallelism [14]. Skeleton programming involves the use of skeleton codes, which can be viewed as generic and reusable functions (patterns) that are derived from functional programming languages [19]. Patterns for skeleton programming are discussed in [38].

Figure 1.2 demonstrates an example of the parallel patterns *map* and *reduce*. The function `sum`, which takes two operands, computes $(1 + 2 + 3 + 4 + 5)$ in parallel by creating four sub-summations $\text{sum}(1, 2)$, $\text{sum}(3, 4)$, $\text{sum}(\text{sum}(3, 4), 5)$ and $\text{sum}(\text{sum}(1, 2), \text{sum}(\text{sum}(3, 4), 5))$. In Figure 1.2, $((3 + 4) + 5)$ represents $\text{sum}(\text{sum}(3, 4), 5)$ for short. The sub-summation $\text{sum}(1, 2)$ and $\text{sum}(3, 4)$ can

be executed concurrently.

Algorithmic skeleton frameworks have been integrated with object oriented programming languages, such as Java and C++. *MapReduce*, along with its implementation developed by Google [15], is one well known programming model of this kind.

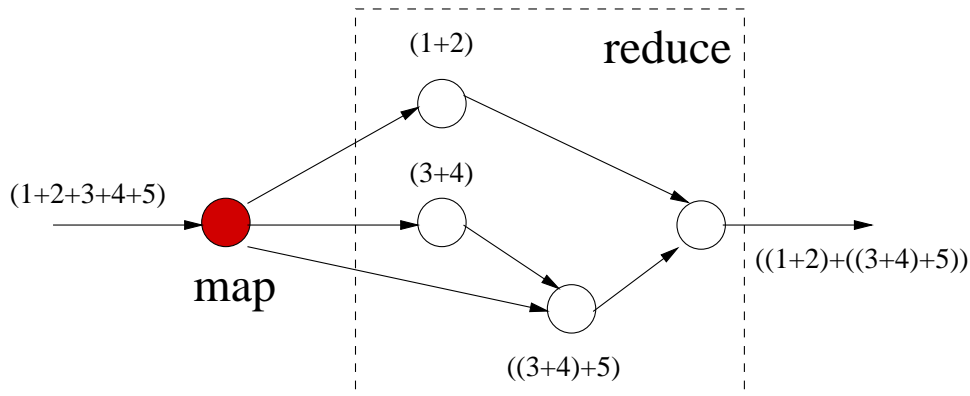


Figure 1.2: An example of the parallel programming patterns *map* and *reduce*.

Dataflow models of computation, as the name suggests, emphasize data and its flow more than computation. They are also considered as first order functional programming models. The function `sum` in Figure 1.2 modeled in terms of SDF semantics is illustrated in Figure 1.3. Here, actor *A* produces five tokens containing the data values 1, 2, 3, 4 and 5 on its three outgoing edges. Consuming two tokens from its input edges, actor *B* (*C*) produces one token, which encapsulates the sum $(1 + 2)$ ($(3 + 4)$). Actor *E* consumes the tokens produced from actors *B* and *D* to obtain the result of the overall computation. To exploit parallelism from this dataflow graph, actors *B* and *C* can be executed simultaneously.

Conforming to the shared memory model, *portable operating system interface* (POSIX) threads, also known as *Pthreads*, is a widely used library that supports SMPs [11].

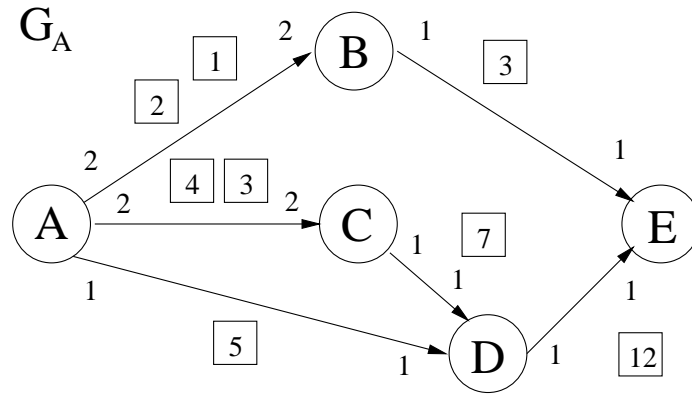


Figure 1.3: An example of dataflow modeling.

Based on sequential programming languages, such as C and Fortran, Pthreads helps to realize the potential performance gains of parallel computing by providing programmers with an extensive library of APIs for thread management and thread synchronization. In this context, threads can be viewed as subtasks of procedures that have their own program counters and function-call stacks. Thus, threads can execute concurrently and share resources.

Some programming languages, such as Java and C#, incorporate support for threads into their associated development frameworks. In such frameworks, parallelism can be expressed through constructs in the programming language, instead of through use of libraries. Although thread programming is a popular way to achieve performance gains from parallel computing, the approach is fraught with problems in terms of understandability, predictability, and determinism, which are the essential and appealing properties of sequential computing [34].

To help simplify parallel programming, OpenMP (Open Multi-Processing) adopts a concept of *incremental parallelization*, where the compiler undertakes the parallelization

of sequential codes based on programmer-specified *directives*. Parallelization is realized through a fork-join model [16], which creates multiple threads at selected instants during execution (fork), and collects threads created by fork operations when they are completed (join). Figure 1.4 illustrates the operation of fork and join operations in OpenMP.

Using OpenMP, programmers start with sequential codes, and insert OpenMP directives (pragmas) to specify which regions of the sequential codes are to be parallelized and synchronized. Such an approach allows programmers to parallelize their sequential codes incrementally, increasing the amount of parallelism exploited as they gain confidence in their evolving implementations.

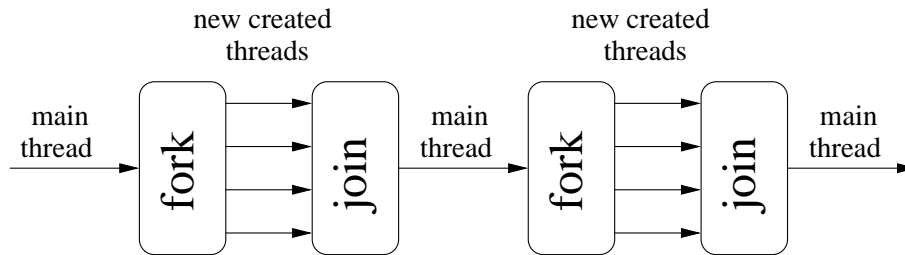


Figure 1.4: Fork-join model.

The execution of a parallel program can be categorized into two styles according to whether or not the number of parallel activities (e.g., threads) is constant at runtime. Fork-join operations spawn parallel activities at some time and then, terminate them at another time. Therefore, the number of parallel activities is not fixed. In this thesis, we are generally concerned with dataflow graph scheduling strategies that involve dynamically determined numbers of parallel activities, although not limited to a fork-join style of parallelism.

In many areas of embedded signal processing, utilizing the underlying memory sys-

tem efficiently has an major effect on important metrics, including real-time performance, cost, and energy consumption. In [44], the authors present an overview of how the memory architectures of embedded systems can be customized to enhance performance.

Additionally, input/output (I/O) interfacing is an important part of embedded system design, and needs to be considered carefully to derive efficient implementations. Various studies have investigated the streamlining of I/O. For example, to improve the latency of file I/O, stream-enabled file I/O is proposed in [30]. This approach allows applications to access files concurrently while they are being transferred.

1.5 Contributions of this Thesis

In this thesis, we present a number of contributions to help improve the mapping of dataflow graphs for DSP applications into efficient parallel implementations. First, we present a novel simulation tool that integrates the Java-based actor programming capability of Functional DIF with PSDF-based graph specification in the DIF language in Chapter 2. This provides the first implementation of a comprehensive simulation environment for PSDF. Such an environment is useful for exploring the capabilities of dynamically reconfigurable SDF modeling and quasi-static scheduling offered by PSDF, and for applying these methods more deeply into design flows for FPGA and digital system implementation.

Building on our newly developed PSDF simulator, which we refer to as *PSDFsim*, we propose a comprehensive PSDF-based design methodology that covers modeling of the target application, simulation of functionality, and hardware architecture mapping.

PSDFsim is applied as a core part of this methodology to help validate the high-level PSDF modeling architecture before committing to lower-level implementation decisions, and later on, to help validate derived hardware description language (HDL) implementations.

To support the scheduling of dynamic reconfiguration, we address the problem of schedule design by introducing a formal framework, called the Dataflow Schedule Graph (DSG) for precisely representing, analyzing, manipulating, and interchanging schedules in Chapter 3. We have designed the DSG representation with two major objectives — 1) it should be rooted in formal dataflow semantics, and 2) it should accommodate a wide range of schedule classes, including static, quasi-static, and dynamic schedules, as well as both sequential and parallel schedule formats. Furthermore, because they are based on the same dataflow semantic framework as the application representations from which the schedules are derived, DSG can naturally represent structures in which schedules are adapted dynamically (e.g., in response to changes in input data characteristics).

As motivated above, resource-constrained implementation of embedded systems requires careful attention to a variety of design aspects, including parallelizing computations, memory management, efficient I/O interfacing, and efficient and fair management of limited resources. Embedded image processing is a domain of signal processing where the challenges in such a multi-faceted implementation process are especially difficult, due to the large volumes of data involved, and the stringent real-time performance constraints. To help address these challenges, we present in Chapter 4, a new design methodology, called the DEIPS (DSG-based design and implementation of Embedded Image Processing Systems) methodology. The DEIPS methodology builds on the DSG model to provide

a structured framework for design and implementation of embedded image processing applications. The DEIPS methodology provides an integrated methodology for addressing the issues that include parallelization of signal processing operations, memory management, I/O interfacing, and efficient management of multidimensional (e.g., image- or block-oriented) dataflow behavior. We demonstrate the DEIPS methodology through two detailed case studies that involve mapping different image processing applications onto a state-of-the-art multicore digital signal processor platform.

Chapter 2

Rapid Prototyping for Digital Signal Processing Systems using Parameterized Synchronous Dataflow Graphs

In this chapter, we present a comprehensive simulation environment for signal processing systems that are modeled as parameterized synchronous dataflow (PSDF) graphs. PSDF was introduced in [6] as a model of computation that augments synchronous dataflow (SDF) semantics with structured methods to dynamically configure dataflow graph parameters, including but not limited to parameters that affect token production and consumption rates of actors. Our proposed new PSDF simulation environment is useful for exploring the capabilities of dynamically reconfigurable SDF modeling and quasi-static scheduling offered by PSDF, and for applying these methods more deeply into the design and implementation for signal processing systems.

2.1 PSDF Operational Semantics

The PSDF operational semantics allows subsystem behavior to be controlled by sets of parameters that can be configured dynamically. Some basic concepts and terminology associated with PSDF modeling and semantics are described as follows. For more details, we refer the reader to [5].

1. A *PSDF specification* is composed of three cooperating PSDF graphs, which are referred to as the *init*, *subinit*, and *body* graphs of the specification. Actors and

edges in PSDF graphs can be parameterized with arbitrary parameters that can be changed at run-time. For any fixed setting of parameters, the PSDF graph yields an SDF graph.

2. A PSDF specification can be nested within a higher level PSDF graph. Such nesting is achieved by encapsulating the specification as a hierarchical PSDF actor in the higher level graph.
3. Parameters of actors and edges in a PSDF graph can only change between *iterations* of the graph. The precise boundaries between iterations can in general be user-defined; typically, in PSDF they correspond to boundaries between *periodic schedules* of the underlying SDF graph. A periodic schedule of an SDF graph is a sequence of actor firings that executes each actor at least once, and returns the graph to its initial state (the initial set of token populations on the edges) [35].
4. The *interface dataflow behavior (IDB)* of a nested PSDF subsystem (i.e., the numbers of tokens produced or consumed at input and ports of the subsystem) can only be changed by the init graph of the subsystem. The init graph executes once during each iteration of the parent (hierarchically enclosing graph). In general, the init graph is allowed to configure (change parameters in) the corresponding subunit and body graphs. Such parameter changes are achieved by mapping the associated parameters to appropriate actor output ports in the init graph.
5. The subunit graph executes once during each execution of the corresponding PSDF subsystem (each firing of the enclosing PSDF actor if the subsystem is nested). During such an execution, the subunit graph executes; new parameter values com-

puted at outputs of the subinit graph are propagated to corresponding parameters in the body graph; and then the body graph executes based on the updated set of parameters.

6. Parameter changes that are computed by the subinit graph cannot modify the IDB of the body graph or enclosing PSDF actor. This ensures that any parent graph has a consistent view of the subsystem throughout an iteration of the parent graph. Such a consistent view facilitates efficient quasi-static scheduling and associated analysis [5, 41].
7. Based on 4, 5 and 6, parameter changes produced by the subinit graph can generally be viewed as more frequent, but more restricted compared to those computed by the init graph.

We use the downsampler example shown in Figure 2.1 to illustrate these concepts. Here, actor H is a hierarchical PSDF actor that encapsulates a PSDF representation (subsystem) of a dynamically reconfigurable downsampler. Actor D in the body graph of the subsystem represents the core downsampling functionality. This actor is parameterized by the *factor* and *phase* parameters, which represent, respectively, the downsampling ratio F and the phase P of the downsampler ($P < F$). In each firing, D consumes F tokens from its input edge, and produces a single token, which is a copy of the $(P + 1)$ th token consumed during the firing.

Since the input of D is connected as input of the enclosing subsystem, changes to the factor parameter in general affect the consumption rate of the subsystem and therefore its IDB. Thus, the factor parameter can be configured by the init graph, but *not* the subinit

graph. On the other other hand the phase parameter does not affect the IDB, and therefore, this parameter can be configured by either the init graph, the subinit graph, or both.

Actors A , B , and C in Figure 2.1 represent SDF actors. The production rates of A and B and the consumption rate of C are statically fixed at unity. These actors represent data sources and a data sink, respectively, which can be used, for example, to drive the subsystem with test data and collect the corresponding test output for subsequent validation.

As part of the init graph, actor E executes once before each iteration of the parent graph of the PSDF subsystem corresponding to H . Thus, E can be used to perform initialization of the factor parameter, as well as to perform periodic updates to this parameter.

For a more elaborate tutorial discussion of PSDF semantics, we refer the reader to [5].

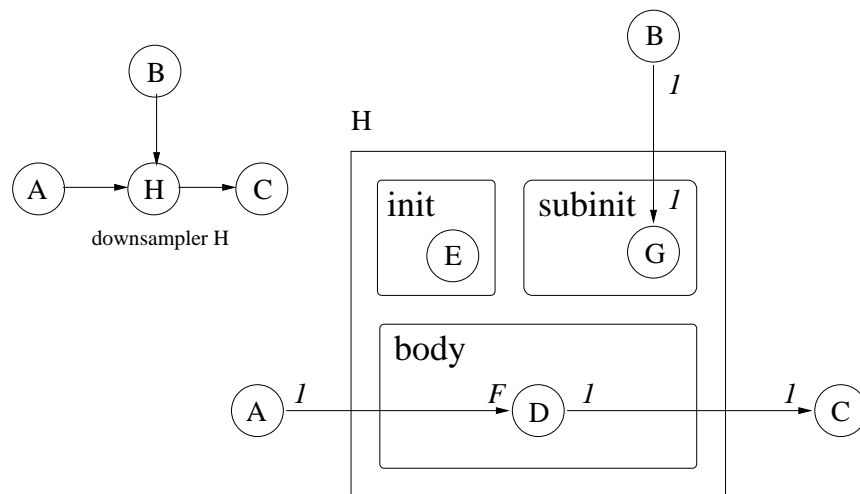


Figure 2.1: A PSDF Downsampler.

2.2 PSDFsim

In this section, we introduce *PSDFsim*, which to our knowledge is the first comprehensive functional simulator for PSDF-based application modeling and design. PSDFsim generates the schedule (simulation sequence) whenever the init graph determines the dataflow behavior that it controls, and then simulation starts. Using PSDFsim, one can validate and test the PSDF modeling architecture at a high level of abstraction before committing to lower-level design decisions, such as detailed hardware-level modeling for the actor internals. Such a two-phase approach to PSDF-based implementation helps to separate the high-level (inter-actor) dataflow architecture design (in terms of PSDF semantics) from the fine grained control and dataflow structures involved in the individual actor implementations, and to allow the former to be applied systematically as a testbench for the latter. More details on this PSDF-based implementation approach are described in Section 2.3.

PSDFsim supports two different forms of parameter propagation — *internal* and *hierarchical* propagation — for dynamic parameter changes to actors and edges. For example, consider Figure 2.2. Each dashed edge in this figure represents a parameter propagation path. Edges (A, a) , (B, b) , and (C, c) correspond to internal propagation paths, which are explained further in Section 2.1. On the other hand, edges (D, d) and (E, e) in Figure 2.2 represent paths for hierarchical parameter propagation. Such hierarchical propagation paths provide channels to update parameters based on new parameter values that are computed from higher level subsystems. Based on properties derived from PSDF semantics, updates through hierarchical propagation override any corresponding

configurations that have been made through internal propagation.

These two forms of parameter propagation facilitate code reuse by allowing arbitrary actors to be applied and adapted in different kinds of contexts through different forms parameter initialization and reconfiguration structures.

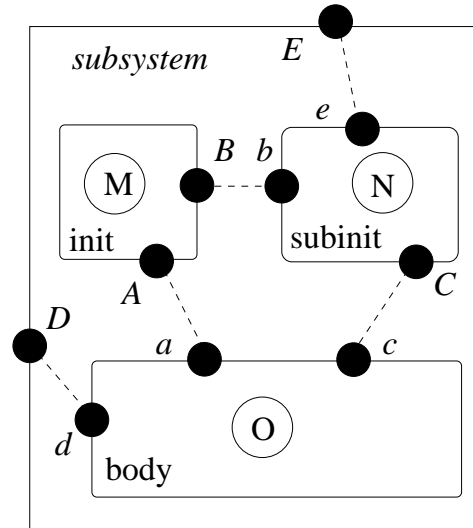


Figure 2.2: An illustration of parameter propagation in PSDFsim.

2.3 PSDF-based Design Methodology

PSDF-semantics can be applied for model-based design in the front end of the FPGA/ASIC design flow shown in Figure 2.3. Such an approach can provide a structured framework to control dynamic functionality and make corresponding adaptations to scheduling strategies and resource allocations. Such a PSDF-based approach involves two phases — high level modeling and validation (*modeling*) and hardware architecture mapping (*mapping*). These two phases can in general be applied iteratively to implement dataflow based parallel processing structures for FPGA- or ASIC-based signal processing

systems.

The modeling phase ensures correct application functionality as well as the correct formulation of the functionality in terms of dataflow and PSDF principles. Through its direct connection to the concurrency modeling capabilities of dataflow, this phase helps provide a framework for efficient implementation even though the focus on this phase is on functional validation rather than detailed hardware mapping. In this phase, procedural software is used to specify the internal functionality of the actors, while a dataflow language is used to specify the high-level (inter-actor) application model. In PSDFsim the Java and DIF languages are used for these purposes of intra-actor and inter-actor, modeling-phase specification, respectively.

In the mapping phase, the designer applies the individual actor models as functional references to derive corresponding hardware implementations using a hardware description language (HDL). The functionality of these “hardware actors” can be validated using the same testbenches as those used in the modeling phase. Similarly, edges in the DIF-based application (application graph) model are mapped into corresponding FIFO implementations using the targeted HDL and associated design library.

By developing the actors based on PSDF principles, and connecting them through standard FIFO semantics, functional correctness of the overall, application-level hardware implementation follows directly from correctness of the original PSDF application model, and correct mappings of the individual actor models into hardware. Additionally, the application level model from the modeling phase can be used as a testbench to begin application-level testing of the hardware, where both functional and timing constraints must be taken into account. Insight from timing analysis of the hardware implementation

can then be used to optimize the hardware actors and possibly to iterate back to the modeling phase to explore refinements or alternatives to the high level dataflow architecture.

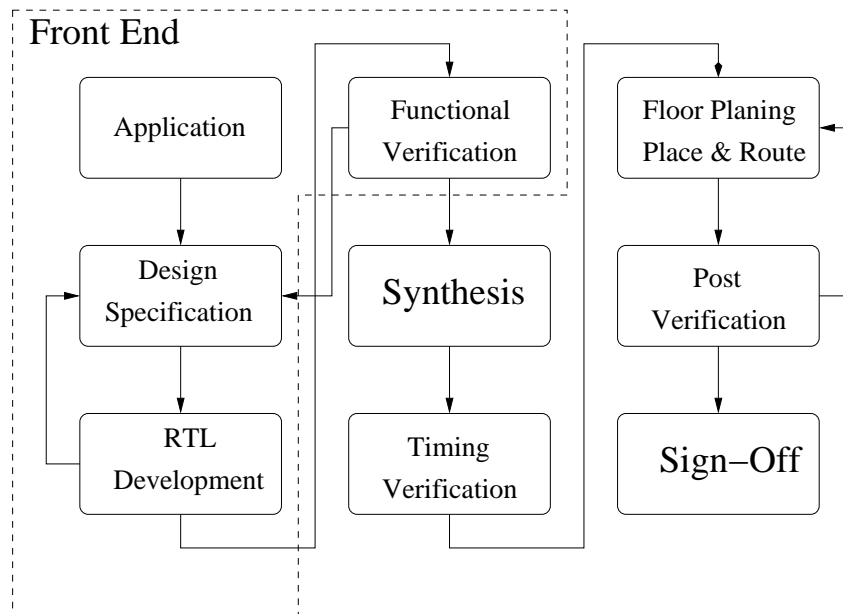


Figure 2.3: FPGA/ASIC design flow overview.

2.4 Hardware Architecture Mapping

In this section, we present details on the mapping phase of our proposed design methodology, including the steps involved in hardware architecture mapping of PSDF actors, graphs, and specifications. Previous work on mapping dataflow structures into hardware include the work on VLSI dataflow arrays [31], SystemC [21], and multidimensional arrayed dataflow [37]. The methods developed in this thesis are different from these approaches in their support for parameterized dataflow modeling, and the novel features of dynamic parameter reconfiguration and reconfigurable dataflow modeling that are provided by PSDF semantics [5]. Due to the potential for applying parameterized dataflow

semantics with arbitrary dataflow models of computation (subject to suitable definitions of graph iterations, as described in Chapter 1), the integration of the techniques presented in this thesis with the models used in the aforementioned works is an interesting direction for further study.

PSDF and PSDFsim modeling constructs — in particular, PSDF actors, edges, schedules, parameter propagation paths, and operational semantics — map naturally into corresponding hardware structures. The buffer sizes can be determined by the schedule used in the hardware, and other hardware components are generic and reusable (not application-specific). Table 2.1 summarizes this mapping.

Table 2.1: Mapping PSDF constructs to hardware.

PSDF Model	Hardware Components
actor	circuit block
edge	buffer (e.g., FIFO)
schedule	graph controller
parameter propagation path	wire
operational semantics	subsystem controller

Although the complexity of circuit blocks can vary widely, the top-down application of PSDF principles provides a standardized design style for the interaction between different circuit blocks and for the interaction between circuit blocks and the associated PSDF control for scheduling and parameter management for the blocks. This allows for significant reuse of parameterized HDL “glue code”, as well as corresponding streamlin-

ing of verification effort.

We employ self-timed scheduling and control of dataflow actors, where actors can fire as soon as they have sufficient data on their input buffers; sufficient empty spaces on their output buffers; and up-to-date values available for their parameters, as determined by the associated subunit and init graphs. Such self-timed hardware mapping is natural for signal processing oriented dataflow models of computation, and avoids bottlenecks and scheduling restrictions due to the alternative of fully static (globally clocked) scheduling (e.g., see [51]).

Figure 2.4 illustrates the architecture of a standard wrapper for PSDF-based interfacing of actor circuit blocks. Here, the blocks labeled *counter*, *controller*, and *loop count* handle control and iteration management within the functional unit of the actor, which can be of arbitrary complexity. The blocks labeled *cons circuit* and *prd circuit* handle input and output interfacing of the actor based on dataflow rates that may be parameterized and dynamically configured.

The structure of hardware mapping at the PSDF subsystem level is illustrated in Figure 2.5. The dashed lines indicate wires for parameter configuration, and the circuit blocks *B* and *D* are parameterized by the init and subunit graph, respectively. The controllers associated with the structures of Figure 2.4 and Figure 2.5 are illustrated in Figure 2.6.

The circuit block control, illustrated in Figure 2.6(a), is a key part of self-timed, PSDF hardware implementation. At the beginning of a control iteration (the state labeled *PARAM*), the circuit block configures any dynamically managed parameters based on the current settings and tries to consume data from the buffer (*CONS* state). The controller

will block in the *CONS* state until all data has arrived from the corresponding producer actor, and has been consumed for processing by the circuit block. Then the controller enters the *EXE* state and activates the function unit to process the input data and generate any output values. When the output data is ready, the *prd* circuit pushes it onto the corresponding output edges during the *PRD* state. Finally, after all output data has been written, the controller enters the *DONE* state. In the *DONE* state, if the firing count within the current loop execution matches the loop count, then the controller goes back to the *PARAM* state and waits for another circuit block iteration before proceeding; otherwise, the controller goes to the *CONS* state to consume tokens for the next firing.

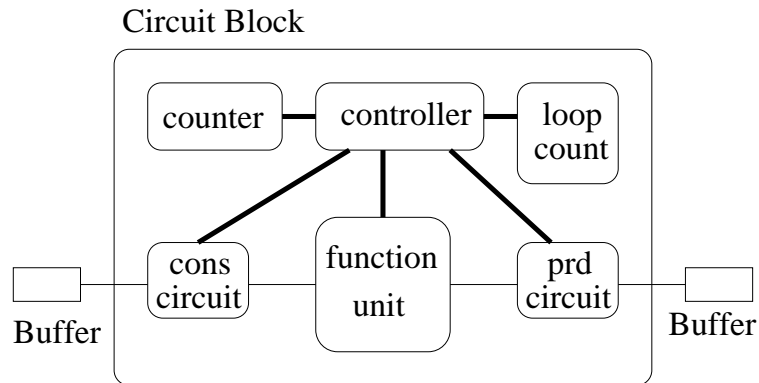


Figure 2.4: Interface and control architecture for a circuit block.

2.5 Case Study: Phase-Shift Keying

In this section, we demonstrate our PSDF-based design methodology using a reconfigurable *phase-shift keying (PSK)* application that can be configured as binary PSK (BPSK), quadrature PSK (QPSK) or 8PSK. We construct PSDF models of the modulator and demodulator for this system, and develop Java-based functional DIF code to specify

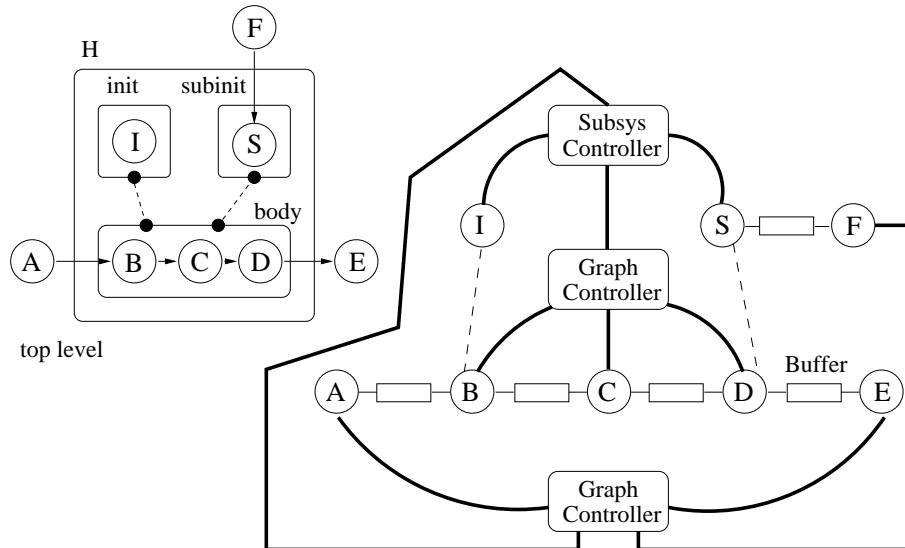


Figure 2.5: An illustration of subsystem-level hardware mapping.

the internal functionality of each actor. The resulting PSDF program is then simulated and tested using PSDFsim, and then hardware mapping is applied to the modulator to derive a Verilog implementation. HDL simulation and synthesis is then applied to validate the evaluate the derived hardware.

Figure 2.7 illustrates our PSDF model of the targeted system for reconfigurable PSK. Here, D represents an input interface that injects samples from the incoming data stream into the dataflow graph; T and P are parameterized lookup tables; $I1$ is an actor that configures the consumption rate (based on M) of T ; $S2$ and $S4$ provide trigonometric functions that are selected based on a dynamic parameter setting; $I3$ configures the production rate of P ; A is an adder; $X12$ and $X34$ are constant multipliers whose associated constants (scaling factors) are managed as dynamic parameters; and B is an output interface for the storing or further processing of the resulting binary sequence. The input interface D makes two copies of each input token on its output since two separate

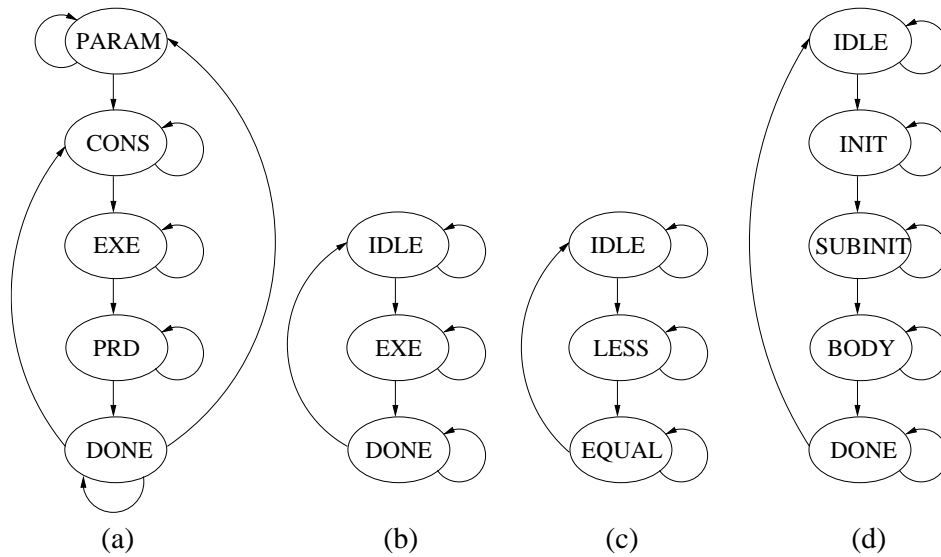


Figure 2.6: Finite state machines for (a) a circuit block, (b) a graph controller, (c) consumption and production circuits, and (d) a subsystem controller.

multiplications are required for each input sample.

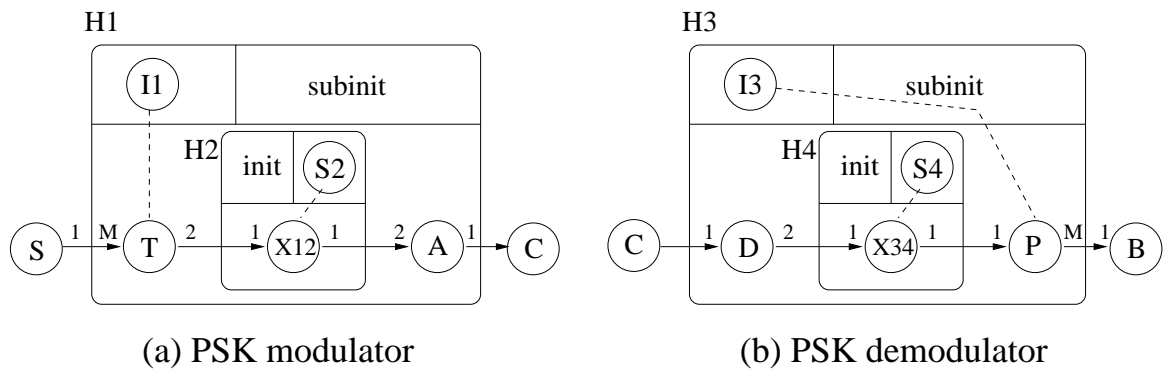


Figure 2.7: PSDF-based model of PSK modulator and demodulator.

Our PSDF model involves a parameter M , which determines which form of PSK to employ. For $M = 1, 2, 3$, an SDF graph associated with BPSK, QPSK and 8PSK, respectively, is effectively activated. After the system model is constructed, we use a

PSDFsim to simulate the system and validate the functionality for the different values of M . This initial simulation is performed assuming no distortion of data in the channel.

Since channel quality is critical to the choice of PSK, we can modify actor C to model the noise in the channel and analyze the simulation results under different PSK configurations. PSDFsim enables such multi-mode application simulation to be executed in an integrated manner — i.e., as a single simulation that includes all PSK configurations along with simulation control functionality that dynamically changes the configuration.

Our hardware mapping of the modulator is illustrated in Figure 2.8. Here, the *filler* block represents an actor that is inserted to help maintain PSDF operational semantics. Since the init and subinit graphs here both contain one node each, their associated graph controllers can be removed. Note also that the circuit blocks associated with blocks T and $X12$ are parameterized and receive parameter value updates from circuit blocks $I1$ and $S2$. This case is implemented manually; however, the implementation is such that all controllers can be reused easily in future designs.

A comparison of the simulation time for the PSK modulator between PSDFsim and ModelSim SE 6.5 is shown in Table 2.2. The time required by PSDFsim to compute the dataflow graph schedule is not included in the time reported here for PSDFsim. This is because this schedule computation is not specific to a single simulation — the schedule can be reused across multiple simulations for the same dataflow graph. The derived schedule is also an important part of the hardware mapping process, and is used (without any recomputation effort) by the lower level, ModelSim simulation. The time taken by PSDFsim in our experiments to compute the schedule for the PSK system is 125 ms.

The improvements in simulation time using PSDFsim help to demonstrate the utility

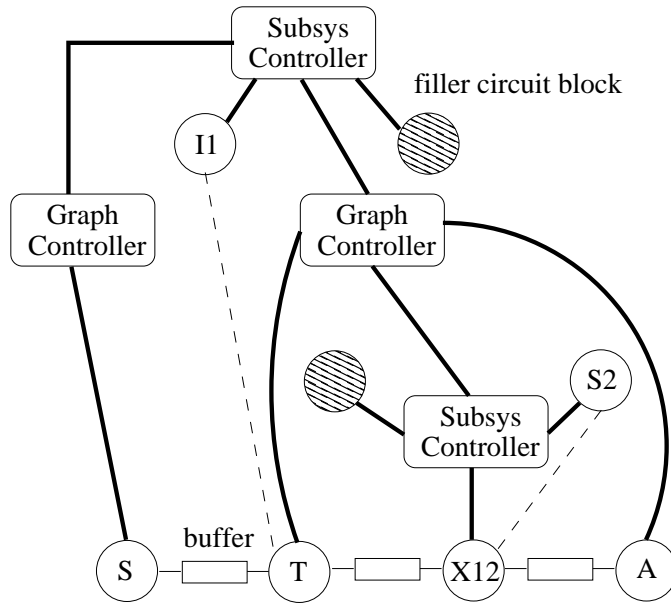


Figure 2.8: Hardware mapping for modulator.

of using PSDF for rapid prototyping and early-stage design. In particular, PSDF allows for faster simulation and design exploration early in the design phase when the high level dataflow architecture is being developed, and detailed HDL simulation (e.g., that provided by ModelSim) is not needed.

Note that these experiments are based on an initial Java-based implementation of PSDFsim that has not been optimized for speed. We expect that with optimization for speed, the simulation time speedup achieved using PSDFsim can be improved significantly.

To provide an area comparison, we instantiate three separate PSK circuits that support BPSK, QPSK and 8PSK individually using SDF-based models. We compare this pure-SDF-based implementation with our PSDF implementation that is derived using PSDFsim and our proposed design methodology. Synthesis results generated by the Cadence

Encounter RTL Compiler are shown in Table 2.2. Although there is some area overhead in the PSDF implementation due to the controllers and auxiliary circuits used for the init and subinit graphs, this overhead is more than compensated for by the hardware reuse that is facilitated by the flexible, dynamic parameterization capabilities of PSDF.

Table 2.2: Comparisons for PSK modulator system.

Simulation time of PSDFsim and ModelSim		
PSDFsim (ms)	ModelSim (ms)	Speedup
47	93	1.98X
Area of PSDF design and SDF design (100 MHz)		
PSDF (cell)	SDF (cell)	Reduction
20004	33602	44.67% (1.68X)

Chapter 3

A Model-based Schedule Representation for Heterogeneous Mapping of Dataflow Graphs

In this chapter, we introduce a formal framework, called the *dataflow schedule graph (DSG)*, for precisely representing, analyzing, manipulating, and interchanging schedules that are associated with high-level dataflow specifications of signal processing systems. We have designed the DSG representation with two major objectives — 1) it should be rooted in formal dataflow semantics, and 2) it should accommodate a wide range of schedule classes, including static, quasi-static, and dynamic schedules, as well as both sequential and parallel schedule formats. Furthermore, because they are based on the same dataflow semantic framework as the application representations from which the schedules are derived, DSG can naturally represent structures in which schedules are adapted dynamically.

3.1 Related Work

A number of dataflow schedule representations have been explored previously. The *generalized schedule tree (GST)* representation provides a tree-based representation of arbitrary looped schedules [29]. A novel schedule format based on dynamic loop counts that is geared towards SDF buffer memory minimization is developed in [43]. The *interprocessor communication graph* and *synchronization graph* models provide dataflow-based

schedule representations for parallel schedules of homogeneous SDF (HSDF) graphs [51]. HSDF is a restricted form of SDF in which the dataflow rate on each input and output port is always equal to 1 [35].

A distinguishing characteristic of our proposed DSG representation is that it is both dataflow based, and capable of handling dynamic schedule structures as well as dynamic dataflow application models. This is in contrast to execution-sequence based representations, which can usually be characterized formally but lack dataflow semantics and are often restricted to static schedules.

The most closely related modeling technique is the synchronization graph model. In this model, self-timed multiprocessor schedules are represented as interacting dataflow graph cycles, where each cycle corresponds to the periodic execution of the actors that are assigned to a given processor [51]. A significant body of theory and algorithms has been developed for this model. We are therefore motivated to generalize the synchronization graph concept beyond self-timed schedules, and HSDF graphs.

The DSG can be viewed as such a generalization. The DSG model can represent dynamic schedules, which can be applied to static or dynamic application models to improve flexibility (e.g., load balancing robustness or data dependent control structures). Furthermore, the model is fully based on dataflow principles, which together with its accommodation of dynamic dataflow semantics, allows for integration with dynamic parameter control methods for dataflow graphs, such as those provided by parameterized dataflow [6] and scenario-aware dataflow [55].

The DSG representation can be used in conjunction with existing task graph scheduling techniques, such as those developed in [17, 27, 32, 50, 57]. For example, the DSG can

be used to model the sequencing structures derived by the scheduling techniques (e.g., as a standard interface for code generation) or to bridge subsystems that are scheduled using different techniques. Indeed, exploring the optimized integration of DSG based schedule control with new and existing task graph scheduling techniques is an interesting direction for further investigation, and one that is especially relevant in the area of heterogeneous computing systems.

3.2 Core Functional Dataflow

For concreteness, we develop the DSG in the context of a specific form of dataflow — the *core functional dataflow (CFDF)* model of computation, which can be viewed as a deterministic sub-class of *enable-invoke dataflow graphs* [46]. CFDF is a highly expressive (Turing complete), dynamic dataflow model. In Section 3.10, we discuss how the DSG model can be adapted to other forms of dataflow (beyond CFDF).

In CFDF, actors are specified as sets of *modes*, where each mode has a fixed production and consumption rate associated with each input and output port, respectively. Each actor has an associated *current mode*, which is maintained as part of its state. When an actor is invoked, it executes its current mode, produces and consumes data (as in other dataflow models), and updates its current mode. Since different modes of an actor can have different production and consumption rates, dynamic dataflow can be modeled flexibly in CFDF.

A distinguishing aspect of CFDF (and the non-deterministic superset EIDF) is that *separation* of enable and invoke functionality for actors is defined as a first class charac-

teristic of the model. Specifically, each actor has an associated *enable* function, which can be called at any time between firings (e.g., by a run-time scheduler), and returns a Boolean value indicating whether or not there is sufficient data available on the actor input ports to fire (invoke) the actor in its current mode. Since such an isolated enable check is available, the invoke function of an actor assumes that sufficient data is present, and reads its input data without blocking reads.

In the implementation of dataflow tools, functionalities corresponding to the enable and invoke methods are often interleaved — for example, an actor firing may have computations that are interleaved with blocking reads of data that provide successive inputs to those computations. In contrast, there is a clean separation of enable and invoke capabilities in EIDF. This separation helps to improve the predictability of an actor invocation (since availability of the required data can be guaranteed in advance by the enable method), and in prototyping efficient scheduling and synthesis techniques (since enable and invoke functionality can be called separately by the scheduler). This separation also leads naturally to a concept of *guarded execution*, whereby an actor firing is conditionally executed depending on whether or not it is enabled.

3.3 The Dataflow Schedule Graph Representation

Given a CFDF representation G_A of an application, a *dataflow schedule graph* (*DSG*) is a dataflow graph that satisfies certain technical constraints (described later in this section), and represents the time-multiplexed execution of G_A across a set of hardware resources. Here, a hardware resource represents an arbitrary computational resource,

such as a processor core, dedicated accelerator or FPGA subsystem, that executes actors sequentially. Constraints imposed on the DSG ensure that each hardware resource can execute at most one actor from G_A at any given time. Tokens that flow along edges of the DSG serve to enable actors for execution (as it becomes their turn to execute). DSG tokens can also contain values that are manipulated and queried during execution of the DSG to achieve various forms of data- or parameter-dependent schedule control.

In DSGs, special actors, called *schedule control actors (SCAs)* and *reference actors (RAs)*, are selected or developed as an integral part of the schedule modeling framework. In contrast to conventional dataflow actors, which represent functional components from the original application specification (*application actors*), *SCAs* are dataflow actors that are dedicated to coordinating control flow in derived schedules. On the other hand, *RAs* can be viewed as “pointers” to application actors. These pointers are equipped with optional auxiliary computations. Intuitively, an RA represents a scheduling “wrapper” that specifies the computation that is executed when the corresponding actor is “visited” during schedule execution. The simplest form of RA is one that simply performs a guarded execution of the actor that it points to. However, more capabilities can be incorporated into RAs using the optional auxiliary computations mentioned above.

3.4 Reference Actors

An RA has a single input port and a single output port. An RA is a homogeneous synchronous dataflow actor in the enclosing DSG — that is, it consumes a single token on each firing from its input, and produces a single token on its output.

Given an RA A , we represent the application graph actor pointed to by A with the symbol $ref(A)$, and we refer to $ref(A)$ as the *referenced actor* of A .

As illustrated in Figure 3.1, an RA A consists of two functions pre_A and $post_A$, which are executed, respectively, before and after the *guarded execution* phase of A . This guarded execution phase, represented by the block labeled “guarded firing” in Figure 3.1, represents the guarded execution of A in terms of CFDF semantics (see Section 3.2).

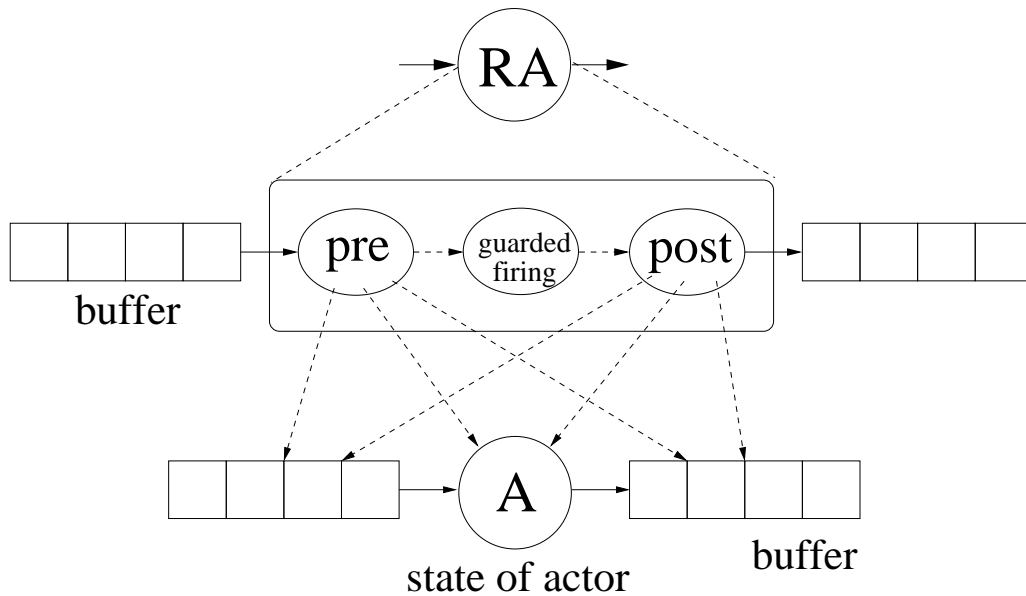


Figure 3.1: The internal structure of an RA.

We refer to the functions pre_A and $post_A$ as *subfunctions* of the enclosing RA. Intuitively, the RA subfunctions provide a mechanism to process and manipulate data that is used throughout the graph to control execution of actors (e.g., to facilitate conditional execution or data dependent iteration in various parts of the graph). The data manipulated by RA subfunctions is encapsulated within the DSG tokens that are produced and consumed by the enclosing RA.

To clarify the operational structure of DSGs, it is useful to emphasize that the tokens flowing on a DSG are strictly for schedule control purposes. Furthermore, because actors in the application graph are allowed to execute only when they have sufficient data (as specified by the CFDF enabling conditions), and CFDF is a deterministic dataflow model, schedule control by DSGs does not violate determinacy — such control only dictates how actors are time multiplexed when they are mapped to the same hardware resource.

RAs can contain internal state. Such local (actor-specific) state is widely known to be compatible with dataflow representations since in dataflow graphs, state can be modeled as self loops with delays (initial tokens) [35, 28]. Thus, the use of state in RAs does not violate our ability to interpret DSGs as genuine dataflow representations.

The following categories of data can be used as inputs in RA subfunctions:

- The value represented by the current DSG token — i.e., the DSG token that is consumed by the enclosing RA firing (pre_A only). This value can be of any type. The type is a design issue of the particular DSG control structure that is being developed for a specific schedule or the particular class of control structures that is being targeted by a particular scheduling tool.
- The state of the enclosing RA.
- The state of the referenced actor.

The following categories of data serve as outputs for (i.e., can be modified by) RA subfunctions:

- The state of the enclosing RA.

- The value of the token that is produced by the RA ($post_A$ only).

Firing of an RA involves the following sequence of steps:

1. The RA consumes a token from its input edge. This token is passed as input to pre_A , which executes, and updates the state of RA.
2. A guarded execution of ref_A is carried out. That is, ref_A is fired once if it is enabled.
3. An execution of $post_A$ is carried out. This execution operates on the state of the RA.

The output value from this execution is produced as the output of the RA firing.

The general purpose of pre_A and $post_A$ is to manipulate DSG tokens. The values of DSG tokens, in conjunction with SCAs, contribute to overall schedule control. Computations in pre_A and $post_A$ are optional. For example, an RA can simply execute the referenced actor unconditionally, maintain no internal (RA) state, and pass input DSG values from input to output without modification. Such “lightweight” RAs are typical in the construction of static scheduling structures, as well as in dynamic structures where dynamic schedule control is managed by SCAs. When code is generated from DSGs, such lightweight RAs can easily be detected and “optimized away” so that they do not result in run-time overhead.

An example of a non-lightweight RA is one that updates DSG tokens with estimates of the amount of energy or execution time taken by the associated firings. Such information can then be used by the enclosing DSG to adapt overall schedule control —

e.g., when the DSG is embedded within a parameterized dataflow system or other kind of reconfigurable dataflow graph framework (e.g., see [6, 55]).

3.5 Schedule Control Actors

To model dynamic scheduling structures, SCAs generally play an important role in conjunction with RAs. An SCA is an actor that can have any positive number of input ports and any positive number of output ports. In other words, an SCA must have at least one input port and output port, and may have any number of additional input or output ports. The dataflow behavior of an SCA exhibits the following *lumped homogeneous synchronous dataflow (LHSDF)* condition: for every firing f of an SCA C , we have that $n_c = n_p = 1$, where n_c represents the total number of tokens consumed by C across all input ports during f , and n_p represents the total number of tokens produced across all output ports during f .

Note that an SCA C can have internal state, and if we model that state as a self-loop edge for C , then this edge is treated independently of the LHSDF condition — i.e., such a self-loop edge is a standard HSDF edge whose dataflow does not “count towards” the values of n_c and n_p .

A token in a DSG can be interpreted loosely as an “actor level program counter” for a given target processor. The LHSDF condition for SCAs along with the HSDF semantics of RAs guarantee that there is only one such program counter (thread of control) that is “demanded of” each target processor. This ensures that the schedule execution modeled by the DSG conforms to the assumption that individual target processors execute actors

sequentially.

Note that while our proposed DSG model is used to model schedules for CFDF graphs, SCAs and hence DSGs do not necessarily conform to CFDF semantics. The primary requirement for SCAs in the context of the associated actor level program counter concept is most naturally captured by LHSDF semantics as opposed to CFDF.

We introduce several types of SCA actors that will be used in this thesis. Table 3.1 summarizes properties of these actors. The *loop* actor has two pairs of inputs and outputs. One pair is used to perform computations within the loop repeatedly, while the other pair is used for conditionally branching into and exiting the loop based on certain control conditions. Since there is only one DSG token, execution always proceeds unambiguously either inside or outside the loop.

SCA actors can be paired with other SCA actors to provide special control functions that involve their coordination. For example, *if* and *fi* provide DSGs with the capability of selecting computations conditionally. The number of outputs for a given *if* actor must match the number of inputs to the corresponding *fi* actor to provide conditional selection of the computations that are enclosed by the matching *if* and *fi* pair.

The pair *snd* and *rec* is used for interprocessor communication and synchronization in concurrent DSGs (CDSGs), which are discussed further in Section 3.7.

3.6 Sequential Dataflow Schedule Graphs

A DSG for a single-processor schedule represents the time-multiplexed (sequential) execution of a set of actors on a single processing resource. Execution of the DSG

Table 3.1: Examples of SCAs.

SCA	# of inputs	# of outputs
<i>loop</i>	2	2
<i>if</i>	1	≥ 2
<i>fi</i>	≥ 2	1
<i>snd</i>	1	2
<i>rec</i>	2	1

models the evolution of actor firings in the associated sequential schedule. To preserve this sequential execution property, a *sequential DSG* (SDSG) imposes the restriction that *at most one token* can be present in the entire DSG at any given time. This requirement formally captures the interpretation of DSG tokens as actor level program counters in the context of single-processor schedules. Just as the program counter in a conventional processor “points to” a single instruction at any given time, the unique SDSG token points to a single SDSG actor, which is the next actor to execute.

For example, consider the class of single appearance schedules for SDF graphs [8]. These schedules are represented in terms of *looped schedules* such that each actor appears exactly once, implying, for example, minimal code size under inline implementation. For example, the looped schedule $(3(2ab)c)$, involving 3 actors a, b, c , and 2 loops represented by the two nested, parenthesized terms, represents the firing sequence $ababcababcababc$.

To demonstrate SDSGs for single appearance schedules, we apply the *loop* SCA

that was introduced in Section 3.5. Figure 3.2(a) shows an SDF graph (G_A) and an associated single appearance schedule ($A(2B)C$). A simple SDSG (G_S) is shown in Figure 3.2(b). In this example, $loop_1$, which is an instance of the *loop* actor, implements an outer loop that models a finite *blocking factor* J . This blocking factor value gives the number of times that the schedule is to be repeated. If the schedule is to be repeated indefinitely ($J = \infty$), then $loop_1$ should be removed, and the output of R_C should be connected directly to R_A .

The actor $loop_2$, which is also an instance of the *loop* SCA defined in Section 3.5, implements control for an inner loop that corresponds to the nested subschedule ($2B$). A token in this SDSG does not carry any values; it simply points to the next actor in the SDSG that is to be executed.

The “D” symbols on the graph in Figure 3.2 correspond to *delays*, and are implemented as initial tokens in the graph. Functionally, a delay corresponds to the z^{-1} operator in signal processing.

Execution of the SDSG shown in Figure 3.2(b) proceeds as follows. The delay (initial token) on the edge ($R_C, loop_1$) causes execution to begin with a firing of $loop_1$. This actor $loop_1$ has one input port, one output port, and an internal state that maintains a loop iteration count n_o , which corresponds to the number of remaining schedule iterations, and is initialized to the blocking factor value J . Each time $loop_1$ fires, it first checks the value of n_o . If $n_o = 0$, then the firing completes with an output token produced on the output edge that is connected to END . On the other hand, if $n_o > 0$, then the value of n_o is decremented, and the firing completes with a token produced on the output edge that is connected to R_A .

This token has the effect of passing processor control to R_A , which then fires the referenced actor A once and passes control (through its output token) to $loop_2$.

The actor $loop_2$ has two input ports $in1$ and $in2$ and two output ports $out1$ and $out2$, as shown in Figure 3.2(b). $loop_2$ also has a state variable n_i , which maintains the number of iterations remaining in the current inner loop invocation.

When $loop_2$ consumes a DSG token from $in1$, it resets n_i to 2, and produces an output token on $out1$ to enable R_B . On the other hand, when $loop_2$ consumes its input from $in2$, it first decrements the value of n_i . If after this decrement operation $n_i > 0$, then it again produces an output token on $out1$; otherwise, it produces an output token on $out2$, which effectively exits the inner loop, and passes control to R_C .

Actors R_B and R_C , like R_A , operate by consuming a single token each from their unique input edges, firing their associated referenced actors, and producing a single output token on their unique output edges. In the case of R_C , the output token produced has the effect of passing control to the next invocation of the outer loop iteration control.

We emphasize that under correct operation, an SDSG contains at most one token. Thus, for an enabled SCA that has multiple input edges, there is never ambiguity about which input edge the next firing will consume data from — the SCA will simply consume the input token from the unique edge that has a nonzero buffer population.

3.7 Concurrent Dataflow Schedule Graphs

Efficient parallel computation is an important motivation for use of dataflow graphs in many implementation contexts. For this purpose, the concept of the DSG can be nat-

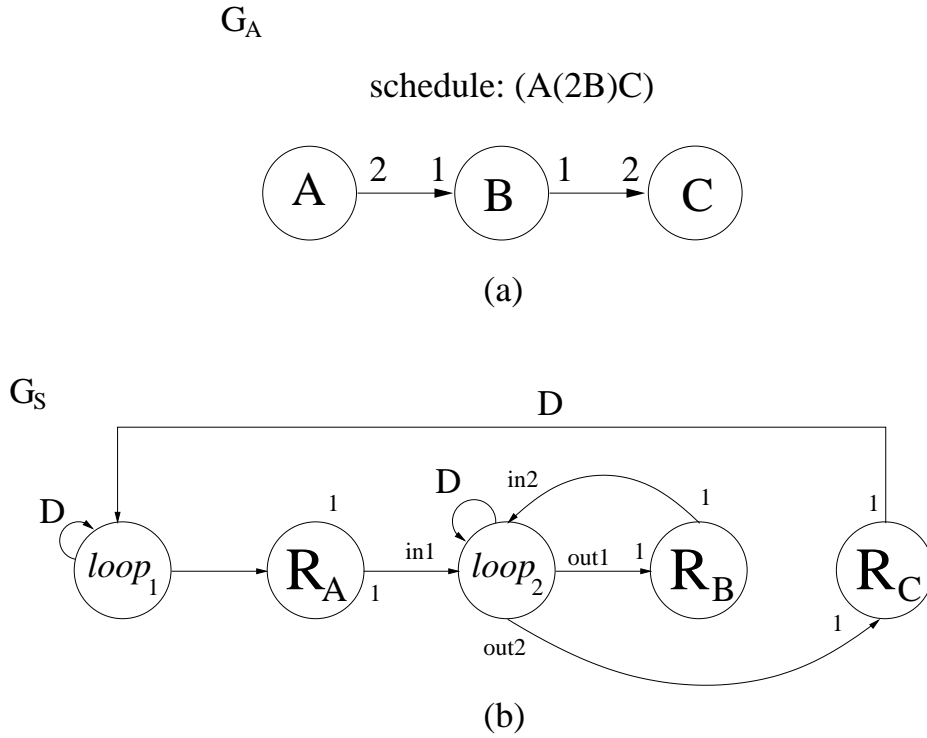


Figure 3.2: (a) An SDF graph (b) A design example of an SDSG for the single appearance schedule $(A(2B)C)$.

usually extended to handle concurrent execution of multiple SDSG “threads”. Multiple SDSGs can be integrated to execute concurrently through the use of a special kind of actor called an *inter-SDSG coordination actor (ICA)*. We refer to the resulting class of communicating, concurrent SDSGs as *concurrent DSGs (CDSGs)*.

Two specific ICAs are *snd* and *rec*, which perform communication and associated synchronization of data that is passed between different processors. As shown in Figure 3.3, *snd* and *rec* both have one pair of input and output ports each — IN_{PC} and OUT_{PC} — for the execution-enabling SDSG token (i.e., the token that is analogous to a program counter or “PC”, as described in Section 3.6). Additionally, the *snd* actor has

a second output port that is used to send data to another processor, and similarly, the *rec* actor has a second input port that is used to receive interprocessor communication (IPC) data. We refer to these output and input ports as OUT_{IPC} and IN_{IPC} , respectively.

Every instance of a *snd* actor is paired with a corresponding *rec* actor in the sense that the OUT_{IPC} port of each *snd* actor is connected to the IN_{IPC} port of the corresponding *rec* actor. The *snd* represents the communication of a single token, including any necessary synchronization functionality (e.g., checking for available buffer space) from the sending processor to the processor on which the corresponding *rec* actor resides. Similarly, the *rec* represents receipt of a single token, including any associated synchronization functionality (e.g., to check whether the corresponding interprocessor communication buffer is non-empty before reading).

In general, the synchronization and data communication features of the *rec* and *snd* actors can be decoupled into more specialized ICAs that separately perform communication and synchronization. Such decoupling of synchronization and IPC operations can lead to opportunities for significantly reduced synchronization overhead (e.g., see [51]). Design and application of ICAs for such decoupled synchronization and IPC is a useful direction for further work.

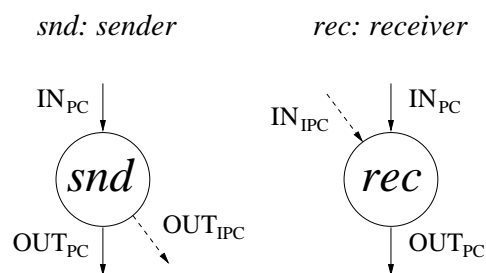


Figure 3.3: The *snd* and *rec* actors.

Figure 3.4 illustrates an HSDF application graph, and a partitioning of this graph across four processors. Figure 3.5 illustrates a CDSG representation of a multiprocessor schedule that is based on this partitioning result. In Figure 3.5, the schedule for each processor is embedded within an infinite loop to achieve an iterative execution of indefinite duration, which is a common execution format for DSP dataflow graph applications. Such infinite loops can easily be replaced by finite-iteration loops if needed by appropriate reconfiguration of the four loop SCAs.

Recall that the “D” symbols in our dataflow graph drawings correspond to delays, which are equivalent to initial tokens. Note also that each of the four concurrent SDSGs in Figure 3.5 has an edge directed from the last actor in the associated actor chain back to the first actor, which is a loop actor. This “feedback edge” represents the transfer of execution from the end of a given loop iteration on the processor back to the beginning of the next iteration. The delay on each of these feedback edges indicates that the execution on the given processor starts with the loop actor.

Each edge in Figure 3.4 that crosses the boundary of two processors can be viewed as an *interprocessor communication edge (IPC edge)*, and is mapped to a corresponding pair of *snd* and *rec* actors in the CDSG of Figure 3.5. For example, the edge (E, I) in Figure 3.4 represents an IPC edge between Processor 1 and Processor 4. In the CDSG, this IPC edge is implemented by snd_1 and rec_4 , which are connected, respectively to the output of the reference actor for E and the input of the reference actor for I .

In summary, the CDSG provides a formal, dataflow-based representation for modeling multiprocessor schedules of dataflow application graphs. Although other representations exist for managing schedules, the CDSG provides a novel combination of features

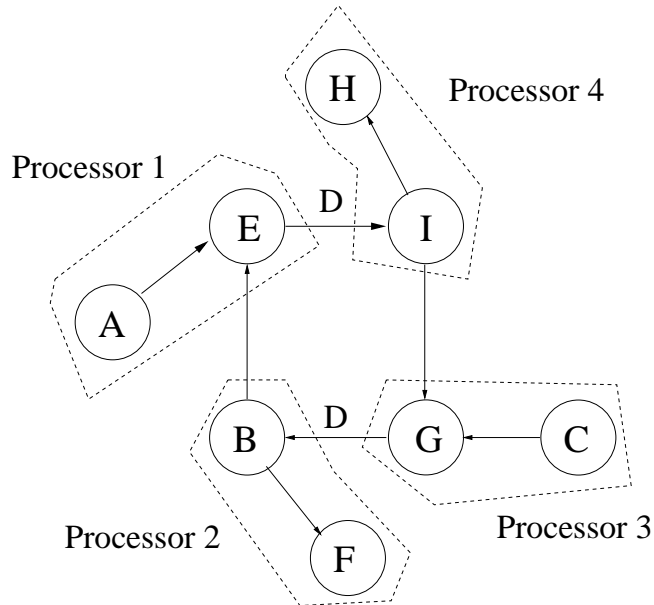


Figure 3.4: An application graph and a partitioning of the graph across four processors.

— in particular, 1) full adherence to dataflow semantics, which helps to unify the model with the associated application representation, and 2) flexible integration of control constructs (through SCAs), which allows for modeling of a wide range of static, quasi-static and dynamic schedules.

3.8 Adaptive Dataflow Schedule Graphs

A major benefit of the SDSG model is that in addition to accommodating static schedules, it provides a common, formal framework for representing a wide variety of dynamic dataflow schedules — i.e., schedules in which firing sequences are adapted dynamically, based on characteristics of the input data or operating environment.

We refer to an SDSG model of a dynamic dataflow schedule as an *adaptive dataflow schedule graph (ADSG)*. Since ADSGs form a subclass of SDSGs, an ADSG can contain

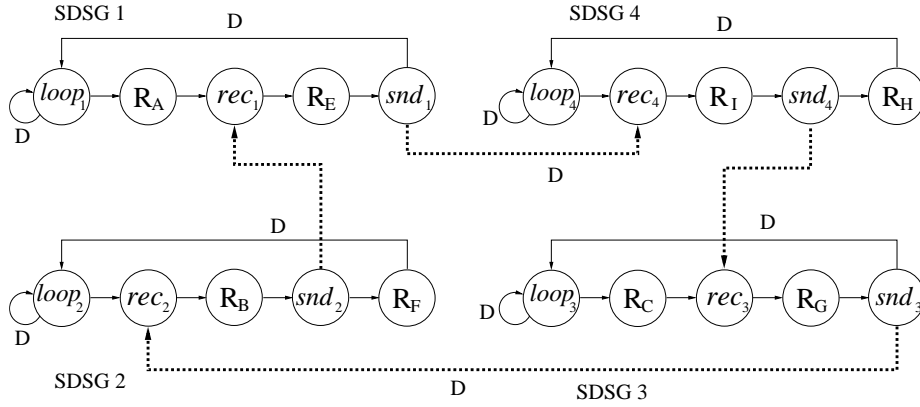


Figure 3.5: A CDSG representation of a multiprocessor schedule that corresponds to the partitioning result shown in Figure 3.4.

at most one token across all of its edges at any given time.

As a simple example, consider the dataflow-based `if-then-else` construct illustrated in Figure 3.6(a). All actors in Figure 3.6(a) produce and consume one token each except for the `switch` (S_W) and `select` (S_E) actors. Although the `switch` and `select` actors are commonly associated with the Boolean dataflow model [10], they can be mapped conveniently into CFDF semantics [47].

Both S_W and S_E consume the Boolean token produced by actor E to determine whether Path 1 or Path 2 will be followed subsequently. Although the path is determined at run time, a schedule for each path can be determined at compile time — (AES_WBS_ED) and (AES_WCS_ED) are schedules corresponding to Path 1 and Path 2, respectively.

Figure 3.6(b) shows a design example of an ADSG for the application graph shown in Figure 3.6(a). In other words, Figure 3.6(b) shows an ADSG model of a specific quasi-static schedule for the application graph in Figure 3.6(a).

Intuitively, the cycle in Figure 3.6 that encapsulates the actors (with feedback edge

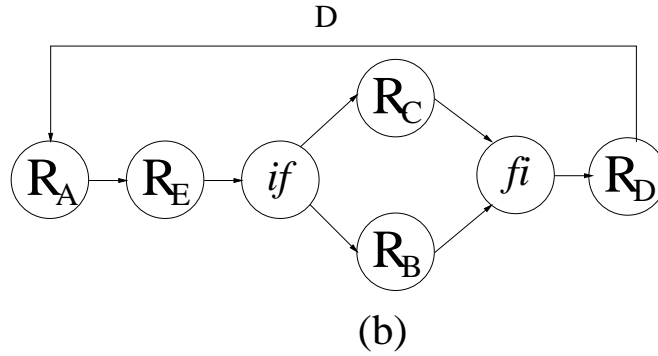
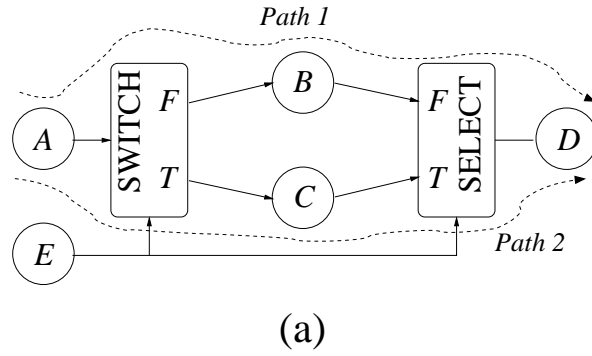


Figure 3.6: (a) A dataflow-based `if-then-else` construct. (b) An adaptive DSG for this construct.

from R_D to R_A) models an infinite, quasi-periodic schedule.

In this ADSG, the output token that is produced by R_E encapsulates the data value that is produced by the corresponding firing of E . This is different from the DSG tokens in our earlier examples, where the tokens carried only control (enabling) information and had no values associated with them. The data encapsulation at the output of R_E can be ensured by the *post* function associated with R_E .

The `switch` actor, modeled by the SCA *if*, examines the output value v from E (through the DSG token that encapsulates its value), and produces a token on one of its output edges depending on whether v is `true` or `false`. This output token, like all other

tokens in this DSG except for those at the output of R_E , does not have any associated data value.

The RAs R_B and R_C are “minimal” RAs that simply perform guarded executions of their associated referenced actors. By design of the quasi-static schedule that is modeled by the enclosing DSG, the enabling conditions for these guarded executions will be satisfied whenever the corresponding reference actors are fired.

On each firing, the SCA f_i consumes the token from its unique, non-empty input edge (which is determined by the “output path” taken by the preceding invocation of f_i), and passes control to the RA R_D .

3.9 Experimental Results

Heterogeneous computing systems integrate different kinds of hardware and software to work together based on the given application requirements. Benefits of heterogeneous computing are often achieved at the expense of ad-hoc, error prone integration processes due to diverse code bases and the lack of unifying formal models. The DSG representation developed in this thesis helps to alleviate this integration problem, leading to more systematic design and implementation of solutions that leverage heterogeneous computing platforms.

In this section, we demonstrate through design examples that the DSG is an efficient schedule representation, which provides robustness and flexibility to the back-end of dataflow-based design processes. Our experiments examine the application of DSGs to improving simulation performance of dataflow graphs, as well as to improving the pro-

cesses of hardware mapping and software implementation from dataflow graphs. Overall, the experiments show the utility of DSG-based design and implementation across a heterogeneous variety of platforms.

3.9.1 Simulation Time Improvement

High level system simulation is a useful application of dataflow graphs in DSP system design. Simulation time for complex dataflow models is often dominated by the computation time of the schedule [59]. For some applications, this overhead can be reduced with well-designed quasi-static schedules, which trade off relatively large amounts of static schedule computations with relatively small amounts of run-time schedule adjustments [51].

In this section, we apply the DSG representation to model a quasi-static schedule, and demonstrate improvement in simulation time achieved by this schedule.

Figure 3.7(a) demonstrates a Boolean-parameterized downsampler H , which can be used to achieve dynamic changes in sampling rate for different parts of a data stream. The actor H consumes α tokens and then sends one of the consumed tokens to either actor B or C , as determined by the value of the actor's `selection` parameter. The values of parameter α and the selection parameter are generated by actor I and S , which are enclosed within the subsystems labeled `init` and `subinit`. The operation of these subsystems as well as the periodic generation and updating of new parameter values are based on parameterized dataflow semantics [6]. For more details on parameterized dataflow, we refer the reader to [6].

To accommodate dynamic changes to α and dynamic selections between actors B or C based on the `selection` parameter, we construct the ADSG representation shown in Figure 3.7(b). Here, we utilize the ability to embed control information within DSG tokens to achieve the dynamic reconfiguration required by the given application.

The RA R_I determines the updated value of the parameter α , which we denote (with a minor abuse of notation) by $\alpha(t)$, and embeds this value in the DSG token that is output by R_I . This value is then used to control the number of iterations in the nested loop SCA labeled as $loop_2$. The *if* and *fi* SCAs perform conditional execution of actor B or C based on the current value of the `selection` parameter. The current value of this parameter is embedded in the DSG control token that is output by R_S so that it can be queried by the subsequent execution of the *if* SCA.

Experimental results with different numbers of application graph iterations (processed blocks of data samples) are given in Table 4.3, and a corresponding chart is shown in Figure 3.8(b). Intuitively, an iteration in this context ends when the DSG token returns to the delay element on the feedback edge in Figure 3.7. That is, an iteration starts when the DSG token leaves from the delay element on the feedback edge and ends the next time the DSG token returns back to the delay element. The experiments are performed using the PSDFSim simulation environment, which can be adapted to implement and experiment with different types of schedules for PSDF graphs [59].

The quasi-static schedule provided by the DSG is compared to the standard PSDF scheduling approach, which can be viewed as a *dynamic scheduling approach*, of re-computing the schedule dynamically every time graph parameters change. The dynamic scheduling approach is more general and easier to apply, while a quasi-static approach has

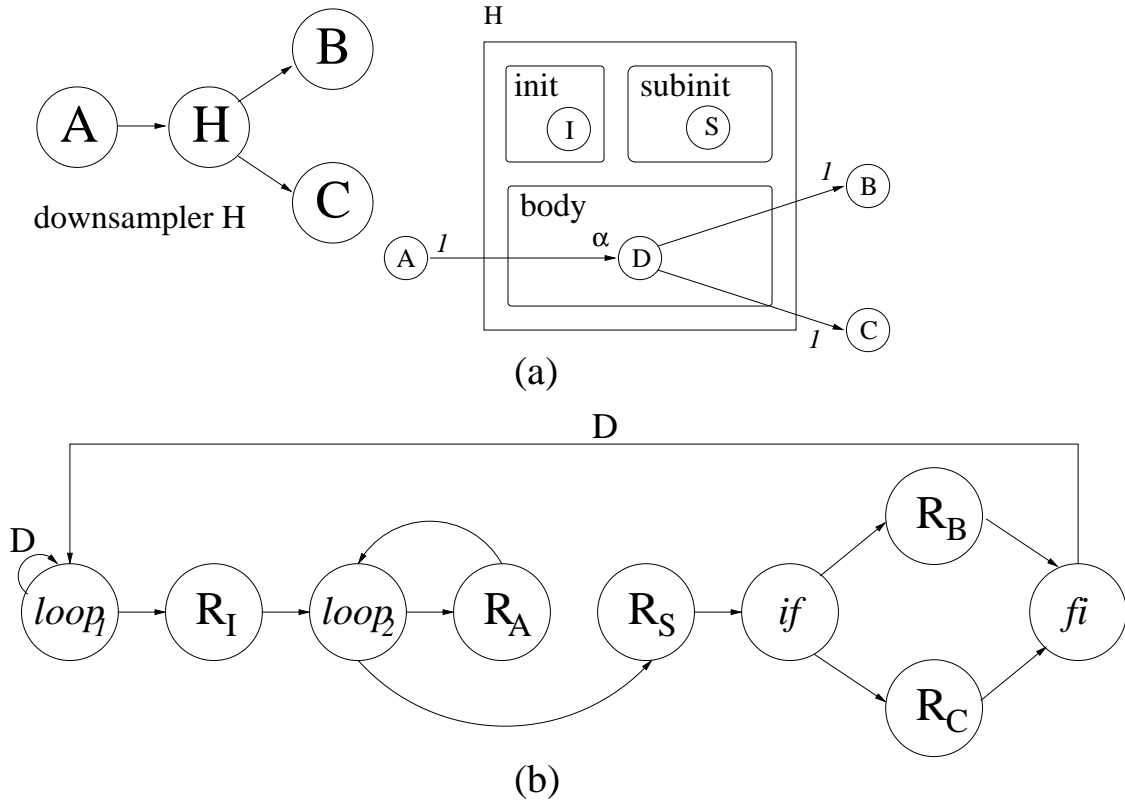


Figure 3.7: (a) A PSDF model for a reconfigurable phase shift keying application; (b) an ADSG representation for implementing this application.

the potential for significant performance improvements by exploiting application-specific structure in the schedule. The DSG representation helps to capture this structure in a standard, dataflow-based format that is easily integrated into the PSDFSim environment.

The performance of the quasi-static schedule is consistently better than the performance of the dynamic schedule. The degree of performance improvement generally increases with increasing numbers of iterations, which correspond to increasing numbers of input samples that are processed in the simulation. This is due to overhead in construction of the DSG representation that is more effectively amortized across the input data set as the size of the data set increases. Thus, for larger numbers of iterations, the DSG-based

quasi-static schedule significantly outperforms the dynamic schedule.

Simple linear regressions for the dynamic schedule (PSDFSim) and quasi-static schedule (DSG) are shown in Figure 3.8(b). As the iteration count increases beyond the intersection of these two lines, the DSG achieves increasingly larger performance improvement compared to PSDFSim. This intersection, based on the linear regressions, occurs when the iteration count is 393.5, which matches the trend observed in our experiments.

In practice, this kind of simulation often requires large numbers of iterations. From our experimental results, we see that PSDFSim is useful for debugging or simulation across small numbers of iterations, whereas the DSG is suitable to achieve simulation time improvement across larger iteration count values — e.g., when performing higher level functional validation of an application.

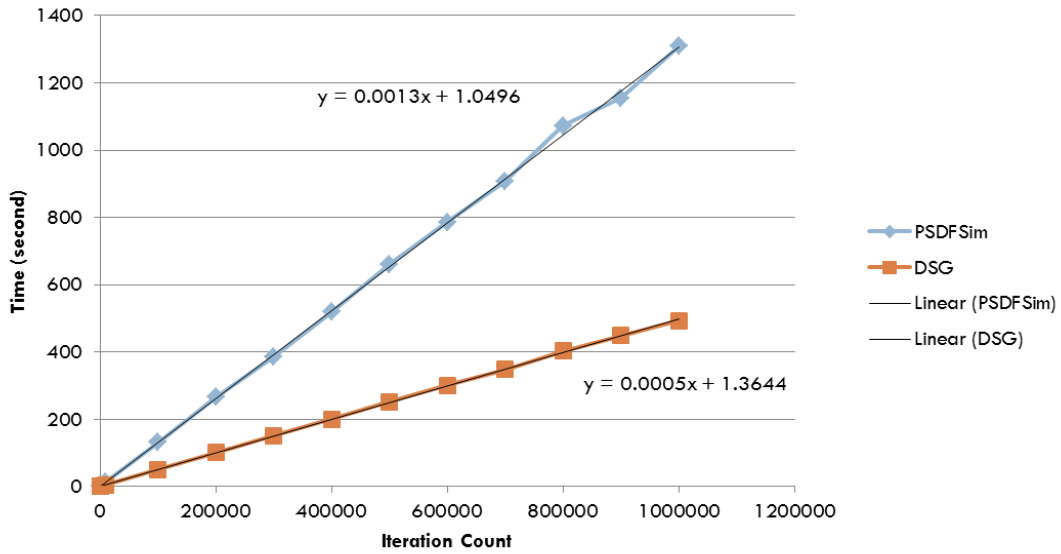
3.9.2 Hardware Architecture Mapping from a DSG

In this section, we experiment with a *reconfigurable phase-shift keying (RPSK)* modulator application, which can be configured as binary PSK (BPSK), quadrature PSK (QPSK) or 8PSK based on the desired trade-off between communication quality and performance. As in the previous section, we apply the parameterized synchronous dataflow (PSDF) model of computation for application modeling and scheduling.

Figure 3.9 shows our PSDF-based model of the RPSK modulator. Two parameters are employed for dynamic reconfiguration — β (analogous to the α parameter in Section 3.9.1) provides the consumption rate of actor T , and ν , a parameter of actor X_{12} ,

Dynamic schedule (Sec.)								
Iteration Count	1	10	10^2	10^3	10^4	10^5	2×10^5	3×10^5
CPU Time	0.556	0.636	1.100	2.668	14.465	131.032	266.345	385.792
Iteration Count	4×10^5	5×10^5	6×10^5	7×10^5	8×10^5	9×10^5	10^6	
CPU Time	519.66	661.581	786.217	907.725	1073.393	1155.039	1309.274	
Quasi-static schedule (DSG) (Sec.)								
Iteration Count	1	10	10^2	10^3	10^4	10^5	2×10^5	3×10^5
CPU Time	0.604	0.608	0.788	1.352	5.736	50.411	101.542	151.169
Iteration Count	4×10^5	5×10^5	6×10^5	7×10^5	8×10^5	9×10^5	10^6	
CPU Time	200.697	251.988	301.135	348.038	402.665	448.164	493.295	

(a) Simulation results for DSG-based quasi-static scheduling.



(b) Performance chart from simulation.

Figure 3.8: Performance comparison between DSG-based quasi-static scheduling, and dynamic scheduling.

provides the modulation frequency. Since ν does not affect the dataflow (production and consumption) rate of its associated actor, it does not show up in the dataflow rate annotations of Figure 3.9.

In a previous study with this RPSK application, we defined a general methodology for mapping PSDF graphs into hardware, and demonstrated synthesis results for the RPSK application using this methodology [59]. Analogous to the dynamic scheduling approach described in Section 3.9.1, this methodology is easy to apply due to its generality, and is also useful as it provides a standard method to realize hardware implementations of PSDF graphs. The DSG provides a complementary method, which can be used (e.g., in later stages of the design process) to specialize the hardware mapping for a specific application, and capture the structure of such specialized mappings in an abstract form that can be targeted subsequently to platform-specific, hardware control structures.

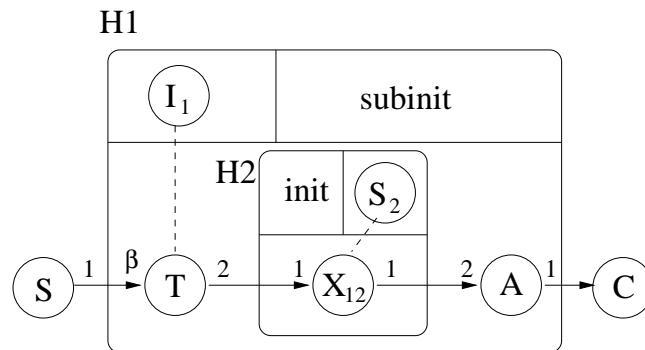


Figure 3.9: RPSK modulator.

From its formal, dataflow-based structure, the DSG is well-suited for transformation into optimized finite state machine (FSM) structures that provide control logic for hardware implementation of the associated schedules. Figure 3.10(b) illustrates a DSG

representation for the RPSK targeted application, along with an FSM that is derived from the DSG. Most of the states map to distinct RAs, and execute the functionality associated with the associated RAs. Since the loop iteration count of $loop_2$ is fixed, the state R_{S_2} is designed to implement loop control as well as firing the actor S_2 .

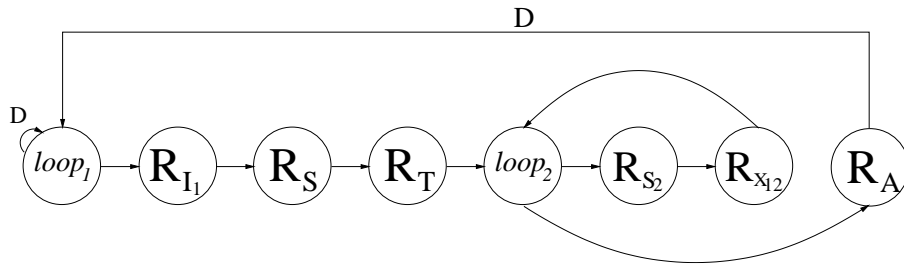
In our experiments with hardware mapping, we targeted ASIC implementation using the Cadence Encounter RTL Compiler for back-end synthesis. The results reported here are synthesis results only (the design was tested thoroughly but not actually fabricated). Table 3.2 shows the improvement in area that is achieved by the streamlined DSG representation compared to the general-purpose PSDF-to-hardware mapping approach of [59]. This improvement is accompanied by a formal, dataflow based representation of schedule logic, which can be retargeted systematically to other types of platforms for rapid prototyping and experimentation with platform-specific implementation trade-offs.

Table 3.2: Area comparison for RPSK modulator under constant speed (100 MHz).

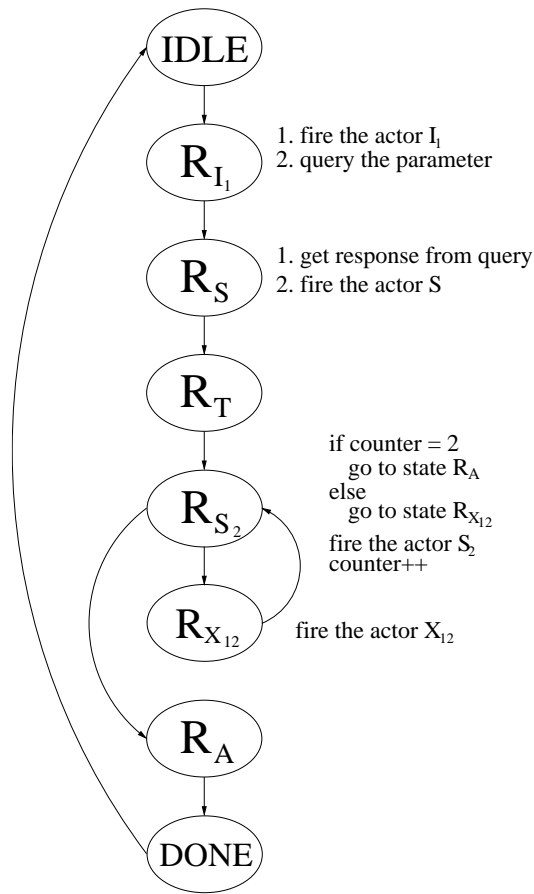
	DSG	General-purpose mapping	Reduction
Area (cell)	18949	20004	5.27%

3.9.3 Application to Software Implementation

We use the core functional dataflow (CFDF) model of computation (see Section 3.2), and the lightweight dataflow (LWDF) programming method [48] for software design and implementation of the RPSK application described in Section 3.9.2, and for DSG-based



(a) A DSG for the RPSK modulator of Figure 3.9.



(b) An FSM for the DSG in Figure 3.10(a).

Figure 3.10: Hardware architecture mapping for a DSG.

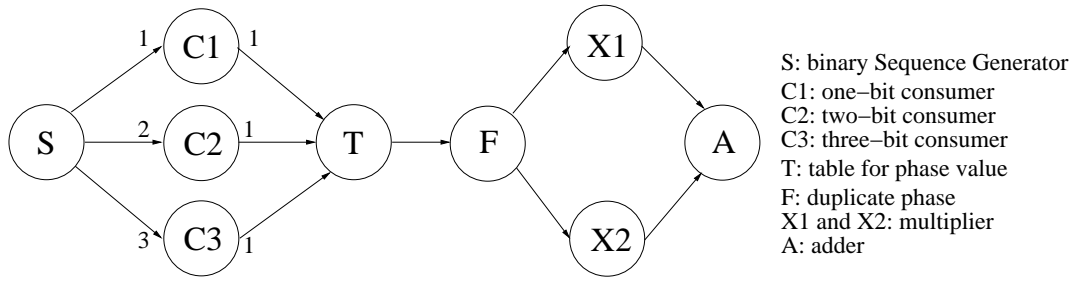
experimentation with alternative schedules in this software context. LWDF can be viewed as a “minimalistic” approach for integrating coarse grain dataflow programming structures into arbitrary simulation- or platform-oriented languages, such as C, C++, CUDA, Java, Verilog, and VHDL. For more details on LWDF programming, we refer the reader to [48].

Figure 3.11(a) illustrates a C language implementation using CFDF and LWDF. In Figure 3.11(a), actors S and T are CFDF actors with three modes each. The variable M is set to 1, 2 or 3 depending on whether the current communication mode is BPSK, QPSK or 8PSK, respectively. Depending on the modes of S and T , data is routed to one of the actors $C1$, $C2$ or $C3$. The consumption rates of these actors are different, as the annotations in Figure 3.11(a) show. In Figure 3.12, the function `guarded_execution` carries out a CFDF guarded execution of the given actor, and returns `true` if the associated actor firing was carried out (i.e., if the actor was enabled to begin with).

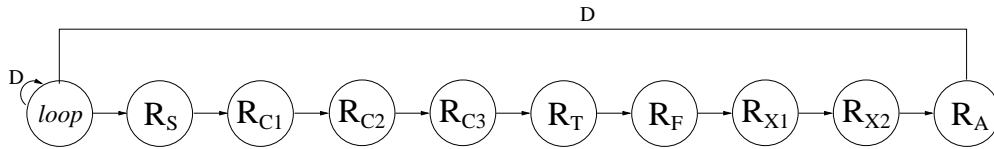
A DSG representation of a *canonical schedule* is shown in Figure 3.11(b). A canonical schedule for a CFDF graph can be viewed as a simple, brute force way to schedule the graph [47]. Canonical schedules usually have high run-time overhead, but can be useful for rapid prototyping purposes because they can be constructed very easily and quickly.

Compared to the canonical schedule, the schedule modeled by the DSG in Figure 3.11(c) is more efficient. This schedule model employs SCAs to direct control flow based on the active communication mode, and minimize run-time overhead due to fireability (enable condition) checking.

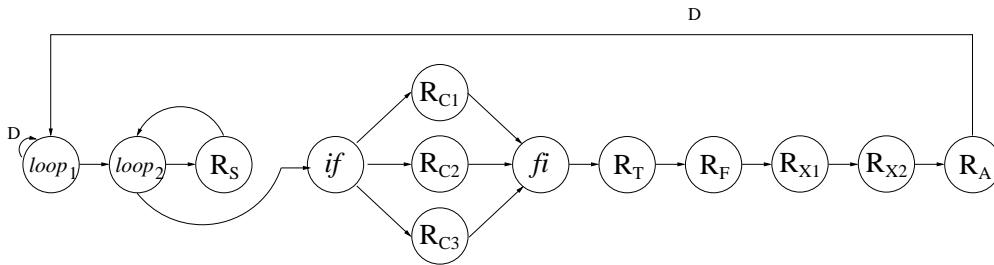
Experiments with these schedules were carried out on a Windows-based desktop computer with a 2.8 GHz CPU and 1GB RAM. The `gcc version 3.4.4` compiler



(a) Application graph for RPSK system.



(b) Canonical schedule represented by an SDSG.



(c) A more efficient schedule represented by an ADSG.

Figure 3.11: RPSK system and alternative DSGs.

was used in the back end of the implementation process.

Table 3.3 compares the performance of the canonical schedule DSG (denoted by *C.Sched*), and the DSG of the more efficient schedule (denoted by *E.Sched*). The overall performance measurement is performed using pre_S and $post_A$, which record the starting and stopping time for execution, respectively. Such implementation of performance measurement functionality represents a useful application of RA subfunctions, which in this case help to modularize, cleanly separate, and formally connect performance instrumentation code with respect to application (actor) and schedule code. The difference in

performance between the two schedules is largely due to the higher frequency of guarded execution failures (i.e., calls to the `guarded_execution` function that return `false`) that result from the canonical schedule.

This experiment helps to demonstrate how the DSG representation can be used as a common framework for experimenting with alternative schedules for software implementation. In this case, the DSG representation is used for initial functional validation using the canonical schedule, followed by a natural progression to a more sophisticated schedule, which provides opportunities for performance optimization once initial functional validation has been achieved. Recall from the formal semantics of dataflow graphs that for all valid schedules (i.e., schedules that respect the dataflow properties of the application), functional correctness is independent of the schedule. Thus, such a progressive or incremental approach to schedule exploration is attractive from the viewpoints of separating concerns, structuring the design process, and improving overall productivity. These are all useful viewpoints to help designers leverage the power of heterogeneous computing platforms.

3.10 Extensions

For concreteness, we have presented the DSG model in the context of CFDF semantics. However, the DSG model can be adapted to other dataflow models or environments that can support a notion of *guarded execution* — i.e., a check for fireability followed by execution of the associated actor if it is found to be fireable.

In contrast, models in which fireability checking and actor invocation are inter-

leaved (with blocking reads) cannot be integrated directly into the proposed DSG framework. However, a more restricted form of DSG can be employed in which RAs fire their associated actors unconditionally. Such DSGs require more care in their construction (to avoid run-time deadlock), and can be useful for modeling static or quasi-static scheduling structures where significant information is available at compile time for DSG derivation.

```

do {
    progress ← 0
    M = query(S)
    for 1 to M{progress ← progress OR guarded_execution(S)}
    switch(M){
        case 1: progress ← progress OR guarded_execution(C1) break
        case 2: progress ← progress OR guarded_execution(C2) break
        case 3: progress ← progress OR guarded_execution(C3) break
    }
    progress ← progress OR guarded_execution(T)
    progress ← progress OR guarded_execution(F)
    progress ← progress OR guarded_execution(X1)
    progress ← progress OR guarded_execution(X2)
    progress ← progress OR guarded_execution(A)
} while (progress)

```

Figure 3.12: Outline of software implementation structure for the DSG shown in Figure 3.11(c).

Table 3.3: Performance comparison between alternative schedules based on DSG modeling. The units of time in this table are seconds.

# of bits	3×10^3	3×10^4	3×10^5	3×10^6	3×10^7
M = 1 (BPSK)					
C.Sched	0.64	0.89	3.49	28.05	275.96
E.Sched	0.63	0.83	3.21	25.69	251.72
Improv.	1.56%	6.74%	8.02%	8.41%	8.78%
M = 2 (QPSK)					
C.Sched	0.64	0.83	3.10	25.16	264.46
E.Sched	0.63	0.73	2.05	14.79	142.26
Improv.	1.56%	12.05%	33.87%	41.22%	46.21%
M = 3 (8PSK)					
C.Sched	0.62	0.81	2.91	23.82	234.18
E.Sched	0.62	0.71	1.60	10.83	103.88
Improv.	0.00%	12.35%	45.02%	54.53%	55.64%

Chapter 4

Multiprocessor System-On-Chip Implementation of Image Signal

Processing Applications

4.1 Introduction and Related Work

Image processing is important for many modern embedded application areas, such as Computer Vision (CV). Such applications involve processing of individual images or sequences of images, where each image consists of numerous pixels, and the relevance of different pixels in an image can vary significantly across different regions in the image. For example, an 800x600 Bitmap Image File (BMP) file contains 800×600 pixels, or 480,000 pixels. The pixel values are numeric, and not every pixel value may contribute significantly to the information we want to know. Taking face recognition as an example, the pixels that make up the eyes are more relevant than the neck.

Extraction of the relevant information for a CV application may require more than one image or “frame”, and therefore, demands more intensive computation and may result in significantly increased memory requirements. As the technology of CV matures, more and more CV applications are being deployed as embedded systems. For example, the CV tool OpenCV [1] has recently been extended with support for the Android platform, which is employed in smart phones and tablet computers.

Multiprocessor Systems-On-Chip (MPSoC) devices are becoming increasingly rel-

evant to the implementation of embedded image processing applications (e.g., see [3]). To support a wide range of applications with diverse functional requirements and operational constraints, MPSoC devices typically incorporate significant amounts of heterogeneity in their architectures. Such heterogeneity in turn creates large design spaces that must be considered carefully for effective embedded software implementation.

In the context of MPSoC-based design and implementation, the process of searching for a suitable hardware/software architecture is known as design space exploration (DSE). In [33], input applications are modeled as directed acyclic graphs, and alternative architectures are explored to support these applications models. Different target architectures are considered with varying numbers of processors and varying topologies for interprocessor communication. The candidates resulting from this architecture exploration are represented as a Pareto curve in terms of three objectives — execution time, critical delay and area cost. In [33], the cost function for each of these objectives is assumed to be given as an input to the DSE process.

A similar flavor of DSE, proposed in [56], deploys simulation for rapid performance evaluation of different MPSoC architectures. Here, simulation is used to evaluate the relevant design evaluation metrics, including performance- and area-related metrics. The simulation is driven by a set of *scenarios*, which capture relevant inputs and operating conditions for the application being implemented. Both the works of [33] and [56] adopt evolutionary algorithms to generate Pareto solutions.

The Daedalus environment augments a simulation-driven design flow [42] with implementation-level DSE, which is targeted to the Xilinx MicroBlaze family of FPGA-targeted, soft processor cores. The Daedalus environment is shown to achieve significant

speed-ups compared to single processor solutions through its integrated simulation- and implementation-based DSE approach.

The Tightly-Coupled Thread Model (TCT), proposed in [24], applies a novel programming model and execution model to derive performance improvement from the combination of functional pipelining and task parallelism. The TCT programming model allows the designer to partition a sequential program into parallel processes that are encapsulated as “thread scope” blocks. Support for synchronization is provided through a joint architecture/compiler solution, where special instructions for synchronization are provided in the targeted hardware (TCT MPSoC), and the compiler generates code to implement synchronization using these instructions. Based on the TCT MPSoC platform, a tool called MAPS [12] models and analyzes coarse-grained application parallelism by constructing *Weighted Statement Control Data Flow Graphs*, which are composed in terms of units of execution called *Coupled Blocks (CBs)*. Each CB can be viewed as a schedulable, coarse-grained functional module.

In [23], the authors develop methods for exploiting parallelism in synchronous dataflow (SDF) graphs to accelerate multithreaded simulation of communication systems. This approach integrates both compile-time and run-time scheduling techniques. The scheduling process includes construction and analysis of *inter-thread communication (ITC)* graphs, which model the effects of partitioning an application-level SDF graph into a set of concurrent threads. Using ITC graphs, schedules for the threads are generated in the compile-time scheduling phase based on trade-offs among synchronization overhead, throughput and buffer memory requirements.

This scheduling is based on careful analysis of SDF dataflow properties (i.e., the

production and consumption rates of individual actors and subsystems), which can exhibit significant variation in communication systems due to the underlying multirate signal processing structures. Each thread that is derived in this way is mapped to a single processing unit and executes its actors in a manner that ensures the property of *bounded-buffer fireability*. An actor satisfies bounded-buffer fireability if it has sufficient data in its input buffers, and enough space in its output buffers to produce its results. The scheduling techniques developed in [23] provide significant speedup of SDF-based simulations while guaranteeing bounded-buffer fireability and deadlock-free operation.

4.2 Overview of Our Contribution

The research works discussed in Section 4.1 focus either on abstract analysis and optimizations in terms of application-level properties or rely on customized hardware features. For example, the TCT framework must be used with the specialized TCT MPSoC, which supports the associated TCT programming and execution models in hardware, and the approach of [23] emphasizes analysis and optimization of abstract SDF properties (production and consumption rates). Furthermore, such methods are not geared toward the special challenges of image processing applications, which involve stringent performance constraints and manipulation of large data streams.

In this chapter, we help to bridge this gap by applying the dataflow schedule graph (DSG), which we introduced in Chapter 3, as a formal model between software and hardware. Our application of the DSG in this manner enables coarse-grain dataflow analysis that can be calibrated and optimized in terms of relevant target architecture characteris-

tics without being tied to a specific architecture. Based on this approach, we develop a methodology for mapping dataflow programs for image processing systems to enable powerful optimization based on the stringent resource and performance constraints involved in their embedded implementation.

Our methodology demonstrates how high-level application- and schedule-level models can be used cooperatively to efficiently parallelize embedded image processing systems based on relevant parameters of the target architecture. We demonstrate our methods through concrete case studies involving practical image processing applications.

4.3 DSG-based Framework for Resource-constrained Implementation of Embedded Image Processing Systems

The techniques, tools, and platforms used for embedded system implementation are diverse. For example, devices in the ARM processor family are configurable and extensible in different ways for different types of smart phones. To help structure the mapping of applications for important classes of applications (e.g., applications for broadband communication terminals, sensor nodes, or security devices), domain-specific tools and APIs are increasingly employed — e.g., see [7] for a discussion of domain-specific design methods and tools for various application domains within the broad area of signal processing systems. Such structured, domain-specific methods help to streamline design processes and move away from the ad-hoc and error prone methodologies that have been common in conventional embedded system development. The models and methods presented in this chapter provide the foundation for a new class of domain-specific tools

that are geared toward mapping embedded image processing applications onto MPSoC platforms.

Building on the developments in Chapter 3 in this thesis, we apply in this chapter the DSG framework as an abstract interface between tools and applications for embedded image processing. Joint consideration of DSG structures and the application graphs from which they are derived allow us to integrate considerations of parallelization and resource constraints together with the application modeling process. This provides designers with a formal framework to co-design their dataflow programs (dataflow-based application models) together with the schedules that will implement these programs on the targeted MPSoC device. Through such a co-design approach, designers can iteratively model, analyze, simulate, and adapt or optimize the use of dataflow graph application structures (e.g., different kinds of filtering topologies or block processing configurations), along with MPSoC-level implementation concerns, such as parallel execution, synchronization, and memory management.

Furthermore, our overall methodology is independent of the hardware during the modeling phase. Thus, our methods can be retargeted across different families of MPSoC devices. At the same time, individual modeling components (e.g., instances of dataflow graph actors, schedule control actors, and interprocessor communication actors) can be characterized in terms of how they execute or how much memory or other resources they consume on the specific MPSoC that is being targeted. Each component in our applied models has precise, execution-related meaning for implementation while preserving any formal properties (e.g., production and consumption rates, bounded memory execution, or deadlock free execution) that can be derived from the application-level dataflow graph

models.

In the remainder of this chapter, we refer to our new DSG-based methodology for design and implementation of embedded image processing systems as *DEIPS* (DSG-based design and implementation of Embedded Image Processing Systems). We present the DEIPS methodology in the context of a state-of-the-art MPSoC platform that is relevant in the embedded image processing domain — the Texas Instruments (TI) TMS320C6678L embedded multicore digital signal processor platform, using the TI TMS320C6678L Evaluation Module [54]. We demonstrate the DEIPS methodology through two case studies, each of which involves mapping of a relevant image processing application onto the targeted multicore TI platform. The two applications that these case studies are based on are image background subtraction, and image registration. Throughout the case studies, we demonstrate how the DEIPS methodology enables designers to experiment with a variety of important design concerns and implementation issues — including parallel computation, memory management, I/O interfacing, and multidimensional dataflow functionality — in a unified framework that is rooted in formal models and methods.

4.4 DEIPS-based Design for Multicore Programmable Digital Signal Processors

In this section, we develop the DEIPS-based design methodology in the context of the TI TMS320C6678L Evaluation Module, which provides an experimentation and prototyping environment for an important family of multicore digital signal processors. The

underlying multicore processor contains eight cores that can run at 1 GHz. Each core has L1 cache and L2 cache. The L1 cache is made up of separate parts for program and data, while the L2 cache provides unified space for program and data. The memory subsystem includes 512 MB memory (DDR3), which we employ as local memory, and 4 MB SRAM (MSMCSRAM), which we employ as shared memory among processors. Programmers can allocate memory space in the L2SRAM, MSMCSRAM or DDR3 through heaps that handle them. A simplified view of the architecture is shown in Figure 4.1.

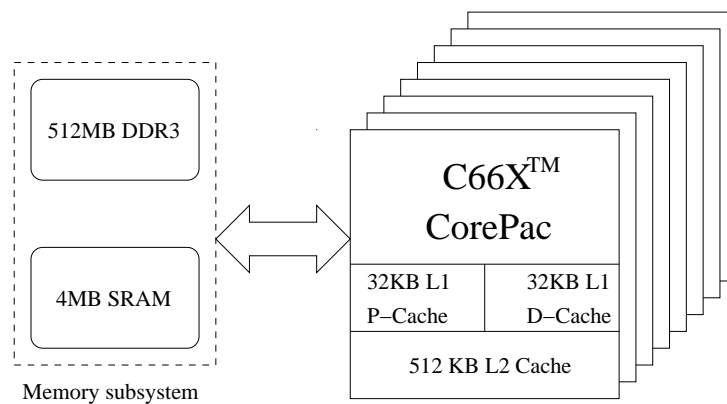


Figure 4.1: Architecture of the Texas Instruments TMS320C6678L Evaluation Module.

This platform provides significant flexibility to programmers and high level design tools to manage thread definitions, memory partitioning for threads, and inter-processor communication.

In the DEIPS methodology, we map each thread to an Sequential Dataflow Schedule Graph (SDSG). The memory usage of each thread, which can be analyzed or simulated efficiently using the underlying SDSG model, is then used to determine the size of the corresponding block of partitioned memory.

Additionally, we implement two pairs of special actors to provide more accurate

DSG representations targeted to the multicore TI platform. These actors implement data synchronization and control synchronization, respectively, on the TI platform.

4.4.1 CDSG Construction

Inter thread interactions are modeled as communication and synchronization actors between pairs of communicating SDSGs, and the resulting system-level schedules are modeled as CDSGs. When more than one processor is employed in the schedule, the CDSG model includes more than one SDSG and provides the nexus of the different SDSGs to coordinate and synchronize their concurrent execution.

For example, consider Figure 1.3. Concurrent execution for this example can be achieved using a schedule that is represented as a CDSG that consists of two SDSGs, as shown in Figure 4.2.

The SCA actors *snd* and *rec* are used to synchronize pairs of communicating SDSGs. Such implementation of interprocessor communication is complicated on the targeted TI platform since it requires handshaking involving heaps in shared memory, and creation of correct heap-based communication mechanisms. To simplify interprocessor communication from the designer's point of view, and to make such communication more reliable, we integrate the handshaking functionality into pre-defined, reusable, TI-targeted *snd* and *rec* actor components. Designer's can then integrate such interprocessor communication components as needed in their DSG structures without having to bother with the low level implementation details associated with interprocessor communication on the targeted device. Each time an *snd* or *rec* is instantiated in a CDSG, the associated inter-processor

communication is effectively instantiated and appropriately configured based on the surrounding CDSG context.

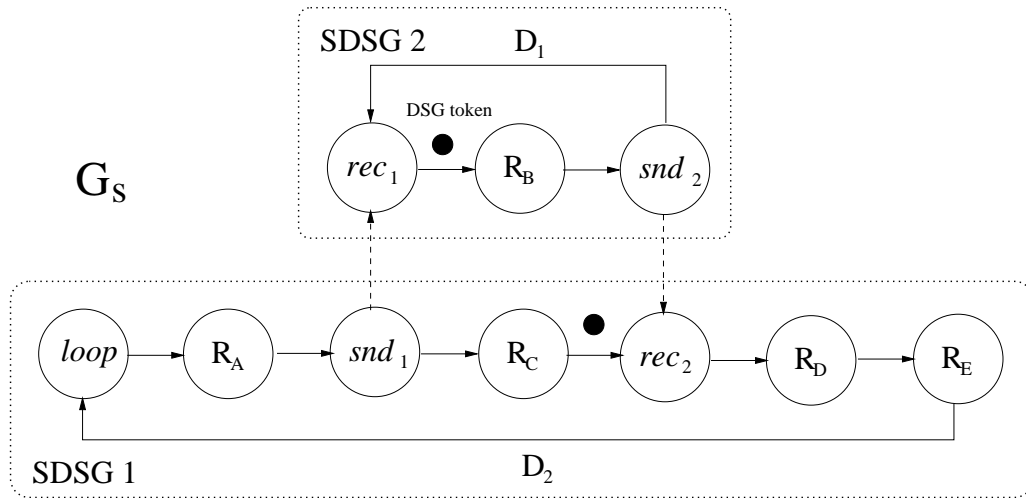


Figure 4.2: CDSG for Figure 1.3.

Similarly, data synchronization is modeled in the CDSG through appropriate actors that implement the required communication functionality on the targeted TI device. This kind of communication modeling is illustrated in Figure 4.3. Here, the dataflow graph application model is mapped onto a CDSG composed of two connected subgraphs, where each processor executes one subgraph. Actors $data_{snd1}$ and $data_{rec1}$, and similarly the actor pair $data_{snd2}$ and $data_{rec2}$, are associated in a pairwise fashion for synchronization and communication — e.g., Actor $data_{rec1}$ is allowed to receive data from $data_{snd1}$ based on how the associated data synchronization actors are implemented on the TI platform.

Figure 4.4 shows the CDSG for Figure 4.3. We denote the RAs pointing to actors $data_{snd}$ and $data_{rec}$ as R_{ds} and R_{dr} , respectively. These actors, R_{ds} and R_{dr} , are used for data synchronization, whereas actors snd and rec perform control synchronization to help ensure correctness of the associated data synchronization.

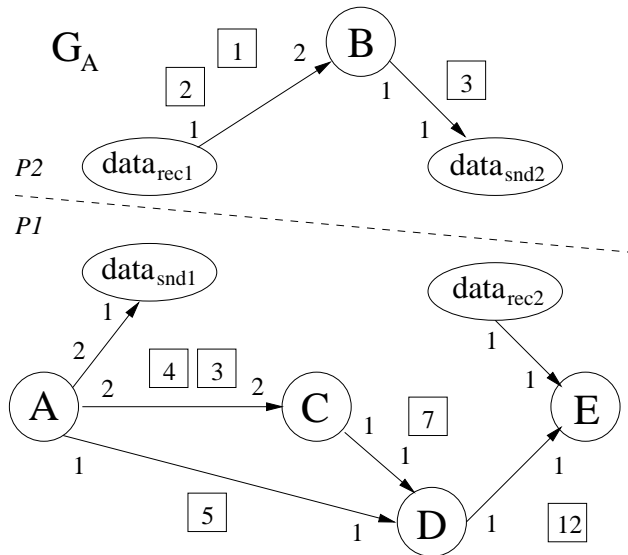


Figure 4.3: Illustration of synchronization for the partitioned dataflow graph of Figure 1.3.

We emphasize that this CDSG representation models one particular schedule for the associated dataflow application graph. In general, a given application graph can have multiple alternative CDSG refinements, and in fact, the number of distinct CDSG possibilities can grow exponentially with the size of the application graph. This potential for exponential growth is related to the NP-hardness of relevant forms of the multiprocessor scheduling problem (e.g., see [51]).

4.4.2 Implementation Details

Each processor on the targeted TI platform loads the same program and identifies which part of the program it should execute. The thread for each processor is created through the DSP/BIOSTM real-time operating system kernel. Among the low level inter-processor communication packages provided by TI, we employ the MessageQ module to develop the $data_{snd}$ and $data_{rec}$ actors, as well as the snd and rec actors for synchroniza-

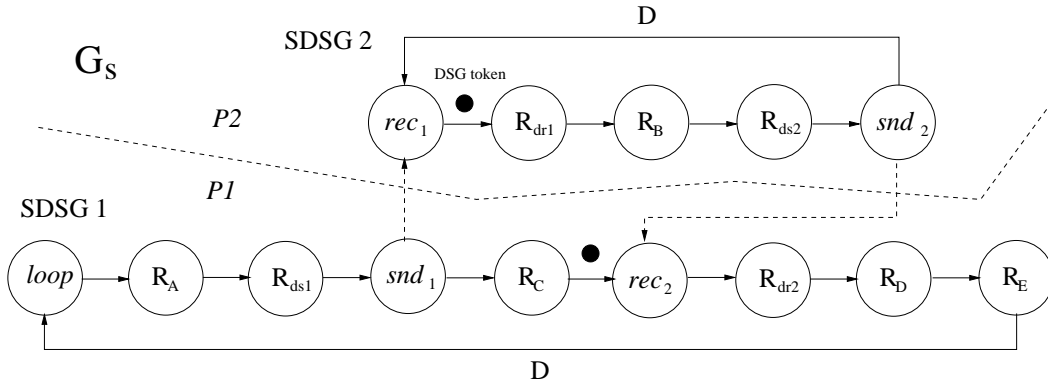


Figure 4.4: The CDSG for Figure 4.3.

tion.

Figure 1 shows a simplified pseudocode sketch for the implementation of a dataflow application graph and a CDSG that executes this application on the targeted TI platform. This example utilizes three of the processors on the target platform, and thus, the pseudocode references three different SDSGs.

Joint Test Action Group (JTAG), also known as *boundary scan*, is widely used for functional testing of integrated circuits. Many embedded platforms are therefore equipped with JTAG interfaces for debugging and prototyping. The core of a JTAG interface can be viewed as a chain of flip-flops through which programs are shifted in and data is shifted in and out. Because of the shifted, serial nature of JTAG communication, JTAG typically dominates the performance of stream processing applications that employ JTAG interfaces. However, some amount of speedup may be possible in such JTAG implementation contexts by dedicating one or a small subset of processors to only to I/O services, and through careful parallelization of the application tasks across the remaining processors. Such optimization can be useful to enhance the efficiency of the prototyping and

Program 1 Implementation on the targeted TI platform of a dataflow application graph
and a CDSG that executes this application.

```
Void main_tsk_func(UArg arg0, UArg arg1){

    //construction of application graph and CDSG

    if (MultiProc_self() == 1) {

        construction of G_A1;

        construction of SDSG 1;

    } else if (MultiProc_self() == 2) {

        construction of G_A2;

        construction of SDSG 2;

    } else {

        construction of G_A3;

        construction of SDSG 3;

    }

    //execution of CDSG

    if (MultiProc_self() == 1) {

        execution of SDSG 1;

    } else if (MultiProc_self() == 2) {

        execution of SDSG 2;

    } else {

        execution of SDSG 3;

    }

}

int main(int argc, char **argv) {

    Int status;

    status = Ipc_start();

    if (status < 0) {

        System_abort("Ipc_start failed\n");

    }

    BIOS_start();

}
```

experimentation processes that are often executed across JTAG interfaces.

4.5 Comparison to Other Dataflow-Based Modeling Techniques for Image Processing

A variety of domain-specific dataflow modeling techniques have been proposed in the past for design and implementation of image processing systems (e.g., see [7]). For example, in [39], a multi-dimensional extension to SDF, called *multidimensional synchronous dataflow* (Multidimensional Synchronous Dataflow (MDSDF)) is presented. By introducing multidimensional (vector-valued) dataflow production and consumption rates for graph actors, MDSDF provides for more detailed and flexible modeling of inter-actor communication.

Windowed Synchronous Data Flow (WSDF) is an alternative approach to dataflow-based modeling of image processing applications [26]. WSDF incorporates rigorous support for sliding window operations, which are important in many types of image processing computations.

Distinguishing characteristics of our DEIPS methodology and its underlying dataflow modeling techniques are its integrated application- and schedule-level modeling capabilities, as enabled by the DSG representation, and the flexible support for dynamic dataflow behavior, as enabled by the underlying core functional dataflow (CFDF) model of computation.

Useful directions for future work that are motivated by this chapter include exploring adaptations of the integrated image processing models and scheduling techniques

developed in this chapter to work with models such as multidimensional synchronous dataflow and windowed synchronous dataflow. For example, the identification and optimized mapping of efficient DSG scheduling structures for these alternative image processing oriented dataflow models appears to be a promising direction for future work. Another interesting direction for exploration is the integration of sliding window support, as motivated by the WSDF model, into the DEIPS methodology and the underlying DSG and CFDF models.

4.6 Case Study: Background Subtraction on a Multicore Digital Signal Processor

Video surveillance is widely used for security enhancement and environmental monitoring. As video surveillance methods become more sophisticated, the volume of data that must be analyzed for surveillance applications increases as well. Pattern recognition helps to incorporate automation in this analysis process, and make it more practical with limited human resources for monitoring surveillance data.

In a workload analysis study of video surveillance systems, it has been shown that the most expensive computation is background subtraction (BG subtraction) [13]. BG subtraction algorithms generally involve two phases — training and differentiating. In the training phase, the construction of the BG model is based on features extracted from a set of training frames. This model construction process involves determining appropriate threshold values for pixels. Then, in the differentiating phase, the BG model is applied to recognize the foreground — if a given pixel of the current frame exceeds the associated

threshold value, it is recognized as a foreground pixel; otherwise, it is recognized as a background pixel. The training methods and threshold values vary with different algorithms and applications, and careful tuning of these key aspects is typically important to achieve high accuracy [45].

One method for BG subtraction, proposed in [52], adopts a Gaussian Mixture Model (GMM) for each pixel. As shown in Figure 4.5(a), the GMMs for the background model are constructed using temporal information (i.e., the sequences of values for a given pixel across successive image frames). We construct another set of GMMs for the background model using spatial information. These spatially-oriented GMMs, as illustrated in Figure 4.5(b), are constructed in terms of blocks of pixels rather than individual pixels.

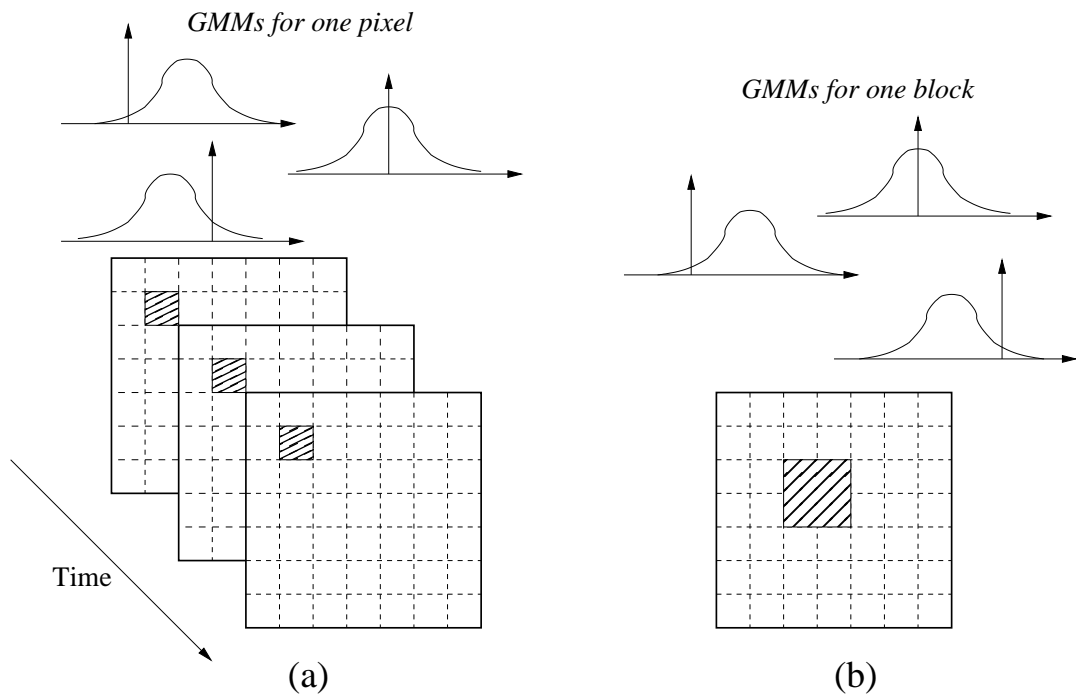


Figure 4.5: (a)Temporal GMMs; (b)Spatial GMMs.

In the construction of the background model, each pixel and block needs three GMMs and five GMMs respectively, thereby requiring large amounts of memory during processing. We let each processor “own” a dedicated block of memory that is partitioned from the 512MB DDR3 memory and managed as a heap on the targeted TI platform. Such memory space is local and dedicated to only one processor for each partition block.

4.6.1 Application Modeling

Background subtraction on images in a video sequence can be structured as a macro-pipeline where incoming frames are processed concurrently with the processing of previous frames, and generation of output results from these frames. In I/O-dominant implementation scenarios, such as our JTAG-based scenario, it can be useful dedicate processors to the I/O functionality associated with such a pipeline.

Early work on scheduling techniques for macro-pipelined implementation of signal processing systems was presented in [4]. The DEIPS methodology provides an integrated approach for modeling, mapping, experimenting with, and iteratively optimizing this class of schedules as well as many other kinds of schedules, for different signal processing applications and platforms.

Figure 4.6 illustrates a high level dataflow graph model that can be employed to experiment with such a macro-pipeline based application configuration. This model is developed using the lightweight dataflow environment (LiDE) [48, 49], which provides utilities and retargetable APIs that designers can apply to experiment with signal processing oriented dataflow design and implementation techniques on a variety of platforms. In

Figure 4.6, two I/O actors are used for reading input images and writing output results, respectively, and one actor, which has two modes, is responsible for the construction of the background models (training mode) and extraction of the foreground (detection mode). Each data token in this application model corresponds to a memory reference that points to an image frame. Thus, the application dataflow graph is modeled at a relatively high level of abstraction, where the actors operate on individual frames as the basic units of computation.

When execution of the application begins, the `Image Reader` actor repeatedly reads frames in BMP format and passes associated frame-level memory references to the actor `Background Subtraction`. In the training mode, the `Background Subtraction` actor constructs the GMMs for background detection. Once application execution switches to the detection mode, the `Background Subtraction` actor extracts foreground pixels using the GMMs derived from the training phases. The results of this foreground extraction process are output to the `Image Writer`, which implements the required I/O functionality across the JTAG interface. Note that actor `Background Subtraction` generates results only in the training mode, and thus, the underlying CFDF actor model is one of a dynamic dataflow actor, where the production rates vary across the two actor modes.

In our experiments with this BG subtraction application, we used 57 frames for training, where each frame had a size of 240x320 and the block size was set to 15x20. The total number of frames used in our experiments, including frames for training and detection, was 365.

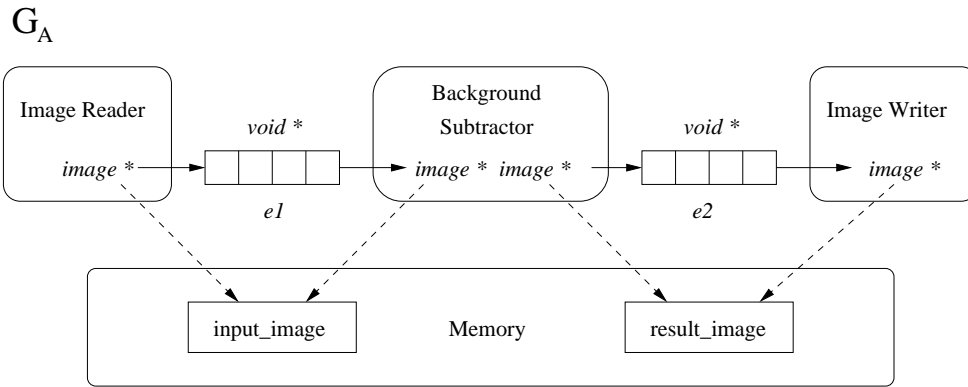


Figure 4.6: LiDE-based application model for macro-pipelined background subtraction.

4.6.2 Macro-Pipeline Mapping Structure

In this section, we discuss the structure and operation of our macro-pipelined implementation in detail, as well how this aspect of our implementation process was modeled and managed systematically through the DEIPS methodology.

Starting from Figure 4.6, we consider each actor as a macro-pipeline stage, where each stage has a single processor in the target platform that is dedicated to it. Communication between pipeline stages is restricted, through appropriate buffer bounding constraints, such that at most one token (image frame reference) can reside on an edge at any given time. This restriction helps to keep to overall memory requirements bounded, and can be relaxed through looser buffer bounding constraints if the image size is smaller or more memory is available on the target platform.

The macro-pipelined version of the application graph and the corresponding CDSG are shown in Figure 4.7(a) and Figure 4.7(b), respectively. In Figure 4.7(b), abbreviations are used as shorthand for the RAs (e.g., IR is used as shorthand for the RA that points to the Image Reader). The pair of actors if and fi select (enable control for) the top

path when the first 57 frames are being processed (i.e., when the system is in the training mode), and switches to the bottom path for the subsequent frames (after the system transitions to detection mode).

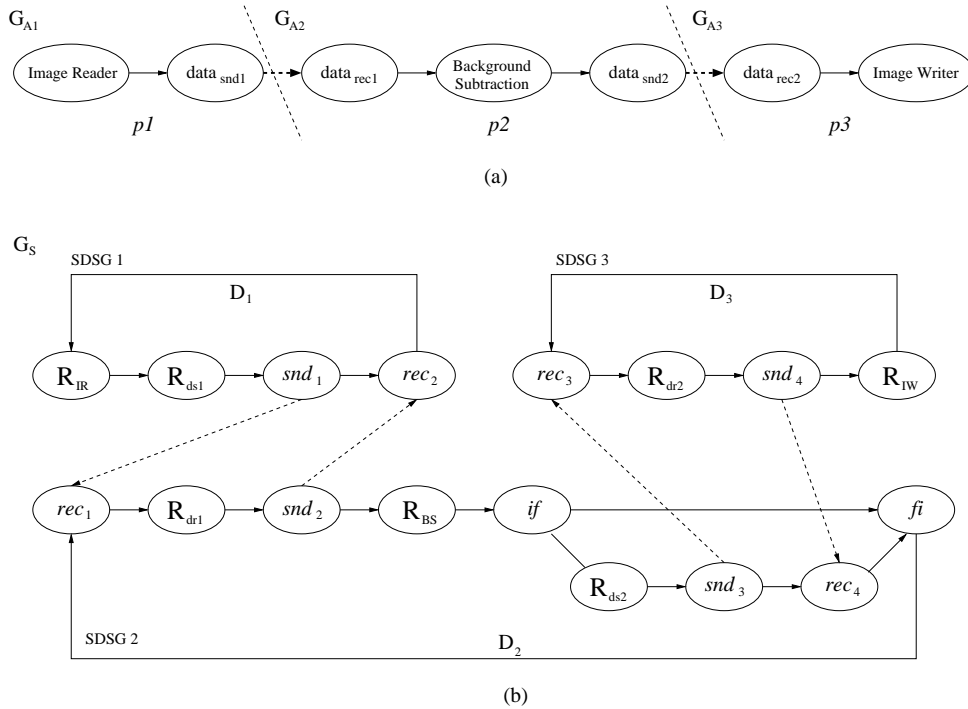


Figure 4.7: (a)Macro-pipelined version of Figure 4.6; (b)the corresponding CDSG.

Actors Image Reader and Image Writer in Figure 4.6 read and write image frames from and to the host PC through the JTAG interface. As shown in Table 4.1, the shifting of image frames into and out of the digital signal processor subsystem consumes significant time while the processor is kept idle. The I/O modeling capabilities of our DEIPS methodology — e.g., through the dataflow actor encapsulation of interface actors — helps to model, analyze, and instrument such interfacing overhead in a systematic and integrated manner. This is in contrast to conventional approaches to DSP architecture/system optimization, which focus primarily on the core computational aspects.

Table 4.1: Actor execution time measurements (ms).

Actor		Time	Actor	Time
Image Reader		27358.031	data sender	0.122
Background	training	11125.280	data receiver	0.122
Subtraction	detection	46.231	Partitioner	3.526
Image Writer		30168.770	Merger	0.011

The performance improvement of the macro-pipeline is dominated by the critical stage — i.e., the stage that takes the most time. The ideal speedup in this context can be formulated as the total execution time divided by the execution time of the critical pipeline stage.

In the training phase, there are two pipeline stages, which are the stages corresponding to actors `Image Reader` and `Background Subtraction`, respectively. Here, from the measured execution time values in Table 4.1, we see that the critical pipeline stage is `Image Reader`, and the ideal speedup can thus be formulated as

$$\frac{27358.031 + 11125.280}{27358.031} = 1.41. \quad (4.1)$$

On the other hand, the ideal speedup in the detection phase (again based on the values in Table 4.1) can be calculated as

$$\frac{27358.031 + 46.231 + 30168.770}{30168.770} = 1.91. \quad (4.2)$$

The difference between the ideal and actual speedup arises due to the effects of system startup and other transient effects, which are not taken into account in the steady state processing that the ideal speedup is calculated based on. For more discussion on transient effects in macro-pipelines derived from dataflow graphs, we refer the reader to [51].

The ideal speedup is approached as the number of frames increases and the steady state system behavior dominates the overall execution time. As shown in Table 4.2, the actual performance improvement observed in our experiments, including the combined effects of the training and detection phase, is 1.26. This speedup is based on the limited number of frames used in our experiments (57 training frames and 308 frames in the detection phase). In addition, the speedup is enhanced as the number of frames in the detection phase increases. This speedup is also limited by our use of the JTAG interface on the targeted multicore platform. As discussed earlier, this interface, while useful for prototyping and experimentation, induces significant performance penalties due to its serial operation.

4.6.3 Fork-Join Mapping Structure

In Section 4.6.2, we showed how we applied our DEIPS methodology to design and implement a macro-pipelined structure for the multicore BG subtraction application, including the application and schedule models for the targeted macro-pipeline. In this section and in Section 4.6.4, we show how an additional method for performance enhancement, the use of fork-join parallelism, can be integrated into our DEIPS-based im-

plementation. While fork-join parallelism is a well-known form of parallelism in many domains, our contribution here and in Section 4.6.4 is to demonstrate the ability of the DEIPS methodology to provide a unified framework for applying and integrating fork-join parallelism together with other forms of system analysis and implementation, including the previously-discussed macro-pipelined and JTAG interfacing aspects of our design, that are relevant for embedded signal processing applications.

In fork-join parallelism, a task is decomposed into parts that are operated on in parallel, followed by a post-processing phase where the partial results computed by the individual parts are combined to produce the output of the overall task.

In our multicore BG subtraction application, we apply fork-join parallelism to the construction of the background model in the training phase. In particular, we accelerate background model construction by applying two processors that operate concurrently on different parts of the input images. Figure 4.8(a) illustrates this application of fork-join parallelization. Here, two new actors, `Partitioner` and `Merger`, are integrated into the DEIPS system model to provide for partitioning and merging of each input image.

A CDSG representation for our fork-join version of the BG subtraction application graph is illustrated in Figure 4.8(b). Here, subgraph (SDSG) G_{A1} dispatches decomposed images to SDSG G_{A2} and SDSG G_{A3} , and then collects the partial results back from them. The 240x480 image is split into 120x480 parts before being dispatched for fork-join-based concurrent processing.

Performance measurements from our fork-join version of multicore BG subtraction are reported in Table 4.3. The measured speedup is 2.13. We speculate that the superlinear speedup here is due in part to a reduction in the cache miss rate (i.e., from the localized

processing facilitated by the partitioning of each input image into two disjoint parts.)

4.6.4 Combination of Macro-Pipelined Execution and Fork-Join Parallelism

In this section, we build on our DEIPS-based multicore BG subtraction system design to combine macro-pipelined execution and fork-join parallelism. This allows us to obtain performance improvement from both forms of parallel execution, which are complementary in this application.

To derive the CDSG for this integrated implementation, we separate the `Image Writer` from the SDSG G_{A1} in Figure 4.8(a) to form the new SDSG G_{A4} shown in Figure 4.9(a). In the resulting CDSG, there is a total of four processors running concurrently. Similar to Figure 4.7, each of G_{A1} and G_{A4} are individual macro-pipeline stages. In addition, G_{A2} and G_{A3} together form a single pipeline stage, and also provide the parallelism for the targeted application of fork-join parallelism.

Table 4.4 summarizes performance results from the implementation modeled in Figure 4.9. The overall speedup for the computation time is 2.15, which is better than both of the implementations discussed previously (macro-pipelining only and fork-join only). However, the improvement compared to the fork-join only implementation is small (1.11%). This can be attributed to the dominance of I/O, which constitutes the critical pipeline stage in the macro-pipelined implementation. The overall speedup in terms of total execution time is 1.24, which is slightly lower than the corresponding speedup for the macro-pipelining only case. This reduction in speedup can be attributed to the double

buffering between the shared memory and local memories, which results in a corresponding doubling of memory transfers.

The results output by the GMM in this implementation are shown in Figure 4.10.

4.7 Case Study: Image Registration on a Multicore Digital Signal Processor

In this section, we demonstrate the versatility of our DEIPS methodology by applying it to a different image processing application — that of image registration. The target platform is the same TI multicore digital signal processor that we employed in Section 4.6.

In *image registration*, two or more images of the same scene are integrated or transformed into a common coordinate system. We specifically examine image registration involving two images. In this case, one of the images, referred to as the *target image*, is geometrically aligned in terms of the coordinate system of the other image, which is referred to as the *reference image*. We experiment with a specific image registration algorithm called the SIFT algorithm [36]. SIFT is a well-known algorithm that is used across a wide range of image registration scenarios. For details on the SIFT algorithm, we refer the reader to [36].

Figure 4.11 shows a CFDF-based dataflow model of the SIFT application, which we use as a starting point in this case study. As with the case study of Section 4.6, we have implemented this dataflow graph in the LiDE environment [48, 49].

As shown in Figure 4.12, the application graph is partitioned into SDSGs G_{A1} and

G_{A2} , which are mapped respectively to processors $p1$ and $p2$ in the target platform. An extra actor, called `broadcast`, is included to reduce synchronization overhead and promote modularity. Note that the application graph is no longer connected in a graph-theoretic sense, while $data_{snd}$ and $data_{snd}$ provide an implicit bridge across the two connected components.

Figure 4.13 illustrates the CDSG that we have designed for the image registration application. Again, abbreviations are used to denote reference actors, so for example, $IR(R)$ is used as shorthand for “Image Reader (reference)”.

The SIFT-based image registration application requires a large amount of memory. On the targeted TI multicore platform, we construct two heaps to manage the DDR3 memory used for processors $p1$ and $p2$. The heap sizes for $p1$ and $p2$ are configured as 336MB and 176MB respectively.

In our experimentation with the SIFT application, we used reference image and target image sizes of 640×480 , and we used the BMP file format for all of the input and output images.

Since memory requirements in this application are significant, we measured the heap usage after execution of each actor through the underlying DSG model. The results are shown in Figure 4.14 and Figure 4.15. For example, in Figure 4.14, the column labeled D shows the amount of free space in Heap 1 immediately after the delay operates on the associated DSG token, and the difference between columns $IR(T)$ and $S(T)$ give the amount of memory allocated by actor SIFT(T). Since the granularity of the measurement here is at the actor level, the resulting memory usage measurements can be used as feedback to the actor designer, and also as input, in the form of actor characterizations, to

higher level analysis and optimization.

Table 4.5 shows the execution time of each application actor on the targeted multicore platform, excluding the time required for I/O. The computation time of the SIFT actor dominates the performance of the application. However, the synchronization time in this implementation is significant. Thus, when experimenting with alternative implementations, special care should be taken to ensure that the overhead associated with synchronization does not overshadow any benefits achieved through parallel execution. In the DEIPS methodology, the use of dedicated actors to represent synchronization functionality helps in the modeling and analysis associated with such trade-off exploration.

The overall performance improvement achieved for this implementation is 1.27. This amount is again limited due to the effects of JTAG interfacing in the underlying platform.

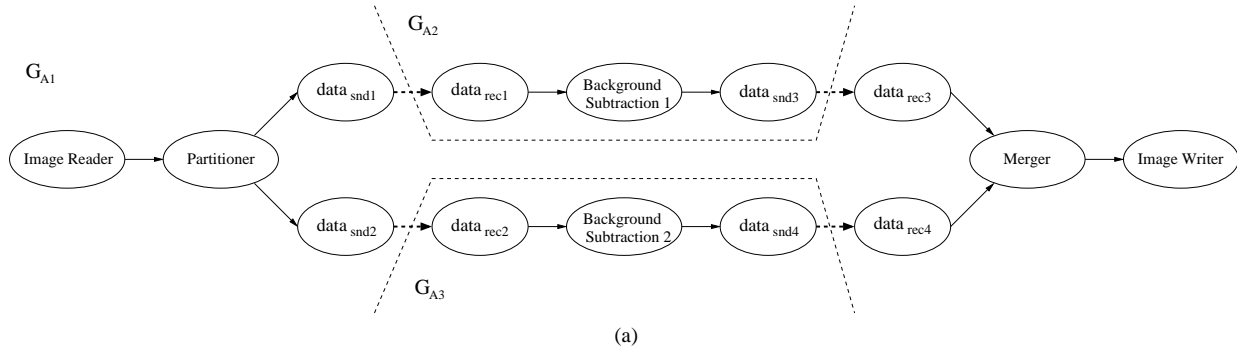
4.8 Summary

In this chapter, we have presented the DEIPS (DSG-based design and implementation of Embedded Image Processing Systems) design methodology, which builds on the DSG model developed in Chapter 3, and provides a structured framework for design and implementation of embedded image processing applications. Our DEIPS framework provides an integrated methodology for design and implementation of embedded image processing systems, including issues related to multidimensional dataflow functionality, parallel processing, memory management, and I/O interfacing. We have demonstrated the DEIPS methodology using cases studies involving a background subtraction application,

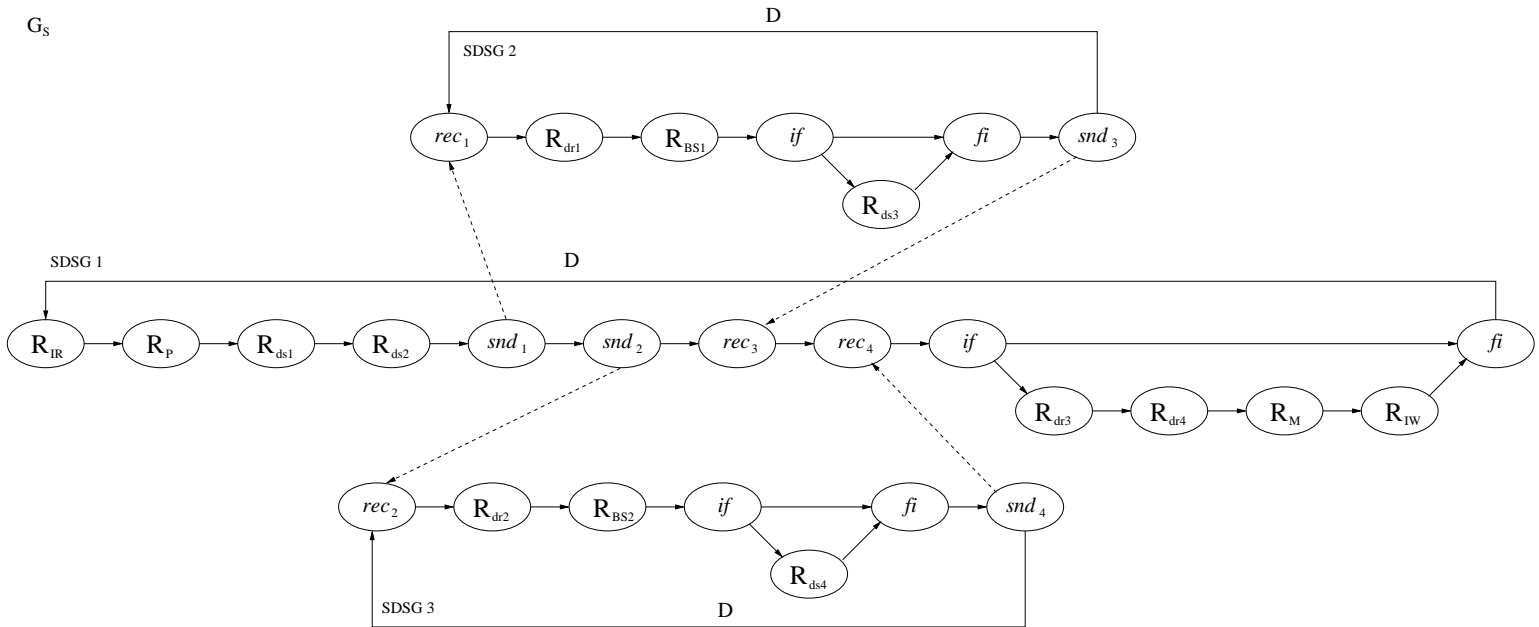
and an image registration application. In both of these case studies, we employed a target platform based on a state-of-the-art multicore digital signal processor.

Table 4.2: Execution time of the SDSG and CDSG in their macro-pipelined configurations (seconds).

Training (57 frames) and Detection (43 frames)		
	System Time	Improvement
SDSG	6306.300	NA
CDSG in pipeline	5299.412	1.19
Training (57 frames) and Detection (143 frames)		
	System Time	Improvement
SDSG	12830.746	NA
CDSG in pipeline	10517.005	1.22
Training (57 frames) and Detection (308 frames)		
	System Time	Improvement
SDSG	22186.749	NA
CDSG in pipeline	17558.713	1.26



(a)



(b)

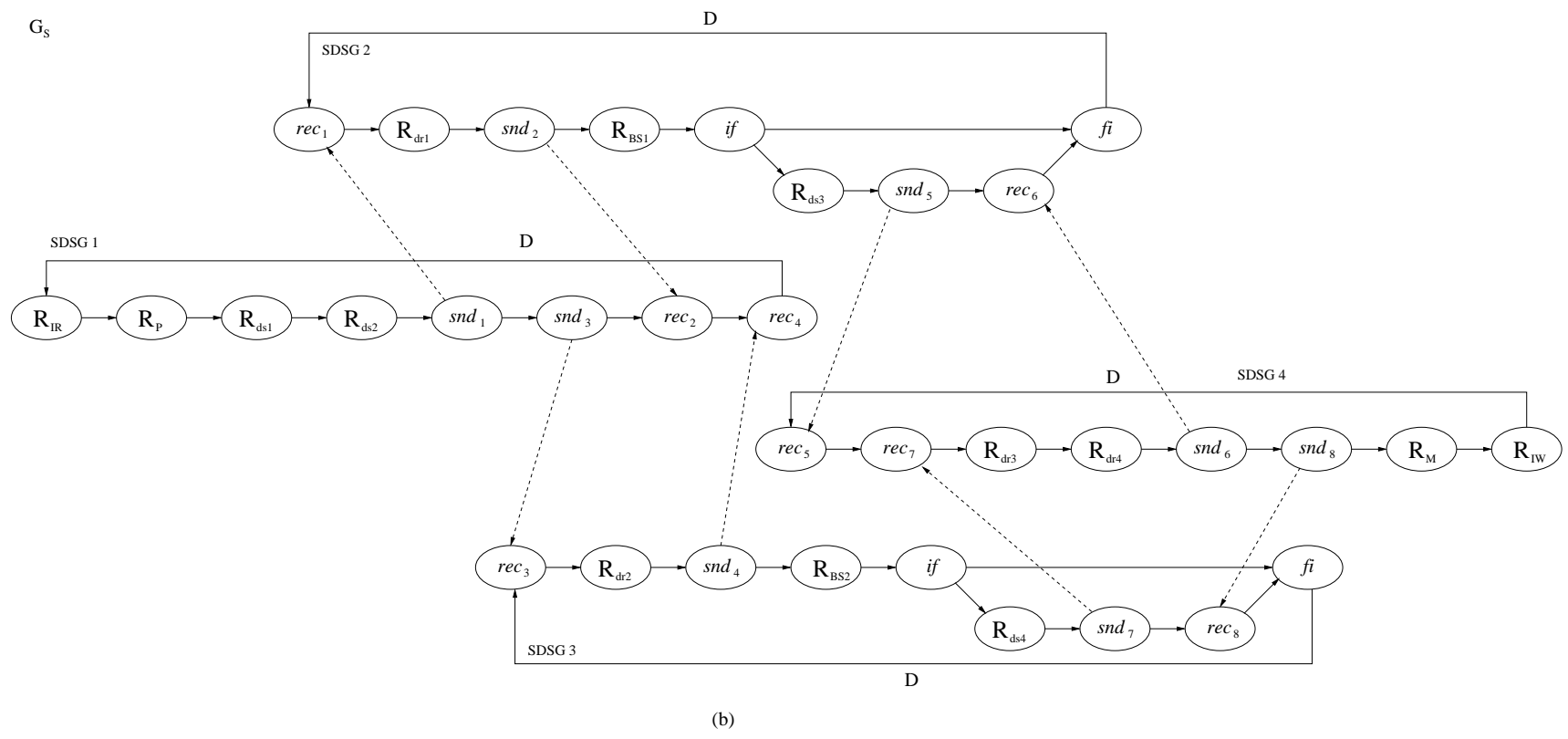
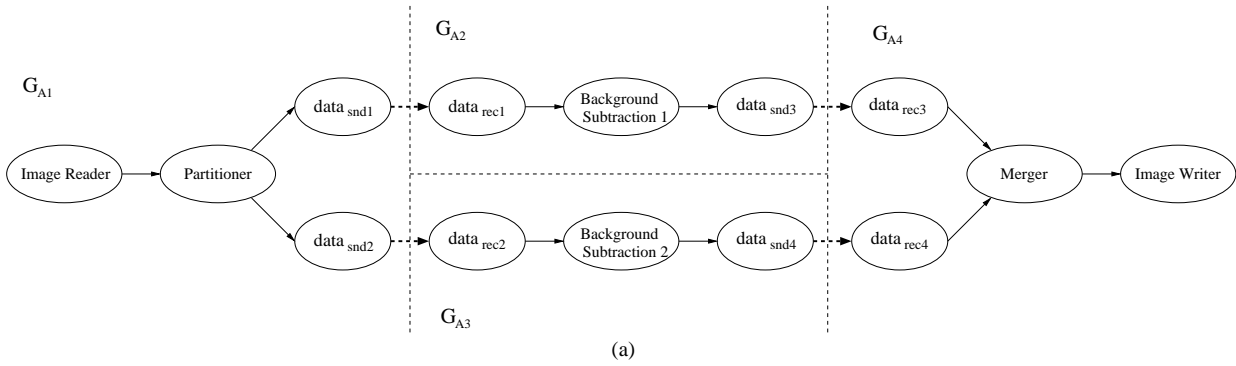
Figure 4.8: (a) Fork-join version of Figure 4.6; (b) a corresponding CDSG representation.

Table 4.3: CPU time for the SDSGs and CDSG in the fork-join version of the multicore BG subtraction application (seconds).

DSG type	Time	Improvement
SDSG	632.285	NA
CDSG in fork-join	296.986	2.13

Table 4.4: Performance measurements for the integrated application of macro-pipelined implementation and fork-join parallelism to the multicore BG subtraction application.

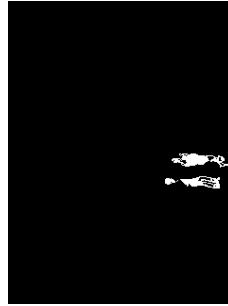
DSG type	Total execution	Improvement	Computation	Improvement
SDSG	22186.749	NA	632.285	NA
The combined CDSG	17947.052	1.24	293.690	2.15



97
 Figure 4.9: Integrated macro-pipelining and fork-join parallelization for multicore BG subtraction.



(c)



(f)



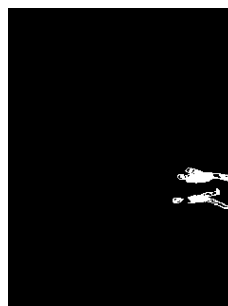
(b)



(e)



(a)



(d)

Figure 4.10: Experimental results for multicore BG subtraction: (a)–(c) show the input frames; (d)–(f) show the corresponding background subtraction results.

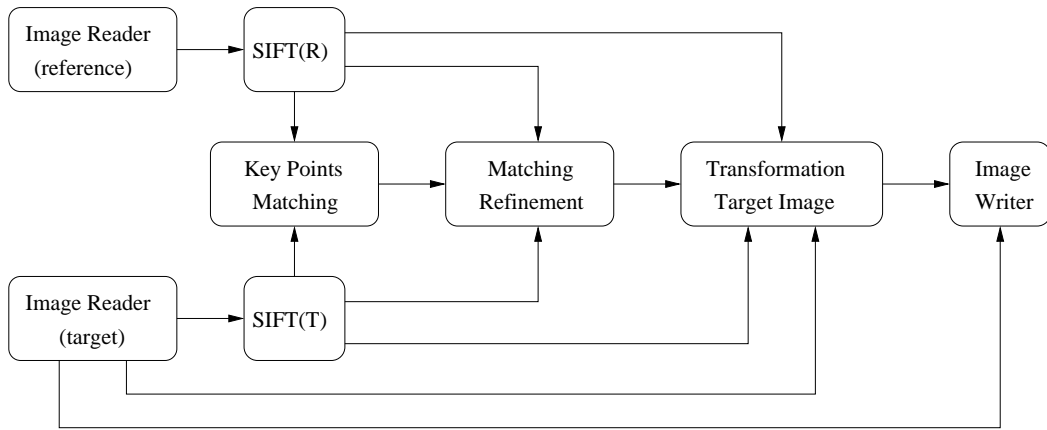


Figure 4.11: A dataflow graph model of SIFT-based image registration.

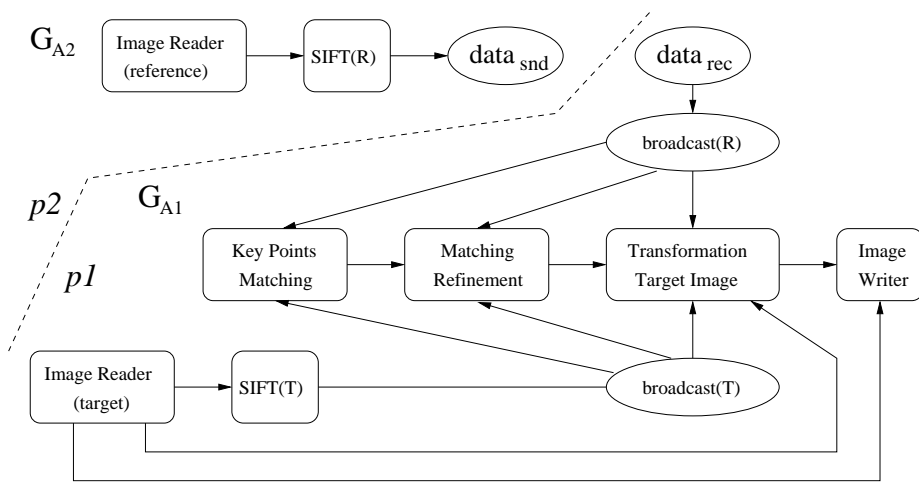


Figure 4.12: Partitioning of Figure 4.11 for mapping onto the target multicore platform.

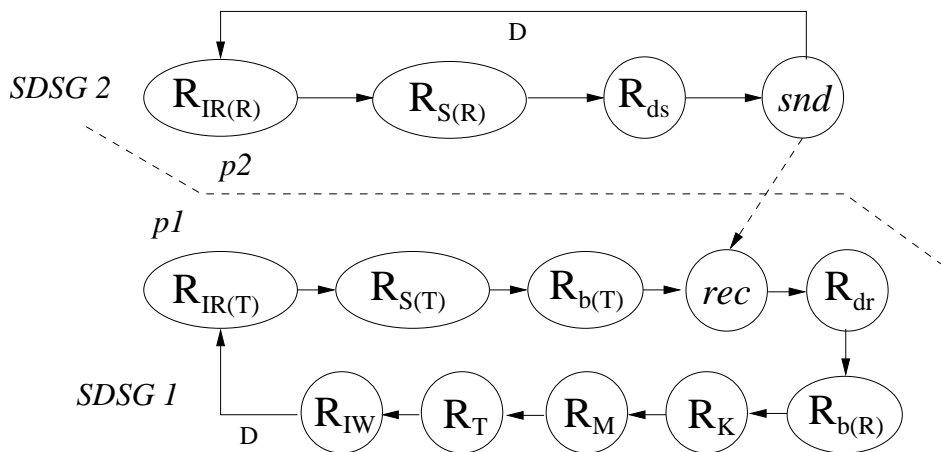


Figure 4.13: The CDSG derived for Figure 4.12.

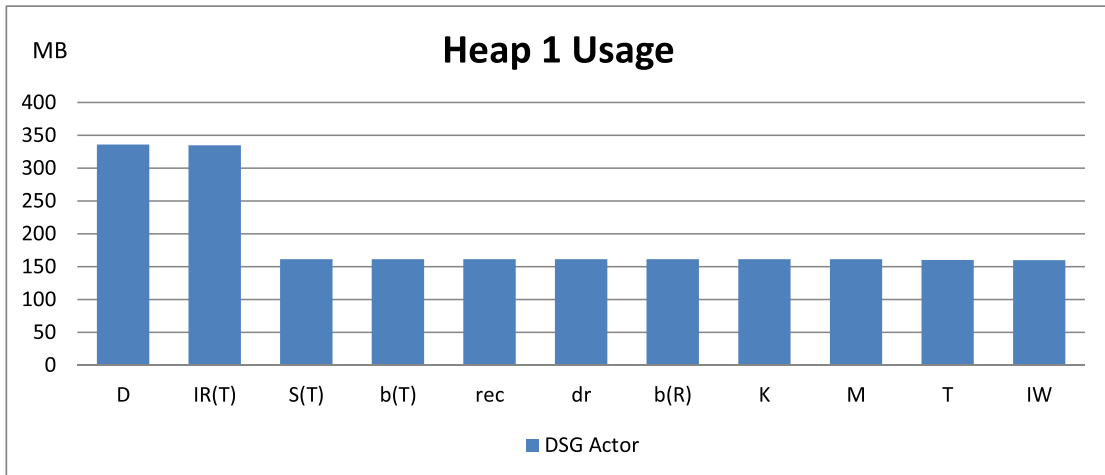


Figure 4.14: Heap usage measurements for SDSG 1.

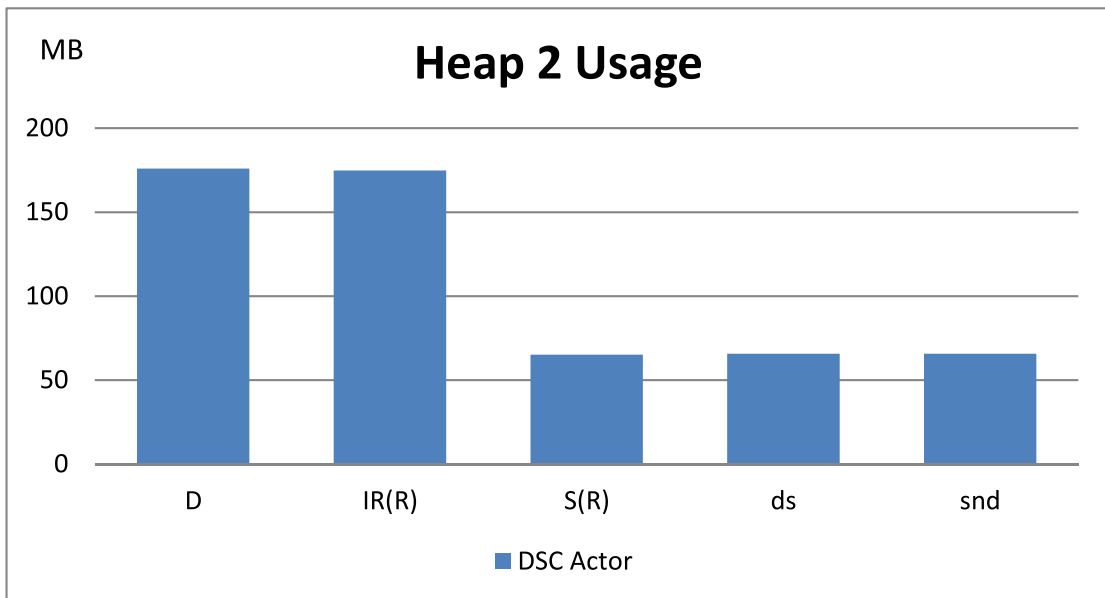


Figure 4.15: Heap usage measurements for SDSG 2.

Table 4.5: Execution time measurements for application actors (ms).

SDSG 1			
Actor	Time	Actor	Time
IR(R)	388.982	<i>data_{rec}</i>	0.762
S(R)	45085.449	<i>snd</i>	0.057
SDSG 2			
Actor	Time	Actor	Time
IR(T)	387.057	b(R)	0.067
S(T)	52235.870	K	11404.301
b(T)	0.067	M	27.849
<i>rec</i>	3764.150	T	1917.787
<i>data_{snd}</i>	0.190	IW	28.38
Execution time of CDSG		69765.718	
Execution time of Seq. code		88285.298	
Improvement		1.27	

Chapter 5

Conclusion and Future Work

In this thesis, we have addressed the importance of the embedded systems, as well as critical problems in the design and implementation of embedded systems, especially in the broad domain of embedded Digital Signal Processing (DSP) systems. We have focused specifically on developing well-integrated models, methods, and tools for handling dataflow-based application modeling, static scheduling, runtime scheduling, memory management and optimized parallel implementation. We have also demonstrated our contributions throughout the thesis on relevant application case studies.

To help in the modeling of dynamically structured DSP flowgraphs, we have demonstrated a design methodology and associated simulation tool, called PSDFsim, for design and implementation of reconfigurable signal processing systems. We have demonstrated the use of PSDFsim and our associated design methods to help streamline the processes of rapid prototyping, design exploration, and implementation. Our experiments show improvements in simulation efficiency and in the quality of synthesized solutions. Furthermore, in contrast to ad-hoc techniques for applying dynamic parameter control to Synchronous Dataflow (SDF) graphs or other kinds of design subsystems, the Parameterized Synchronous Dataflow (PSDF)-based approach that we present provides for well-structured integration of parameter management into the SDF framework. This leads to more efficient and reliable techniques for hardware design and implementation.

We have also introduced the Dataflow Schedule Graph (DSG) as a formal, dataflow-based model for representing and interpreting schedules for dataflow graphs. We have shown that both sequential and parallel schedules can be accommodated in the DSG framework. The DSG is not restricted to any specific dataflow model, and provides for a wide range of static, quasi-static, and dynamic scheduling structures. Furthermore, the model is readily extended with new types of schedule control actors and reference actor subfunctions so that the structures of the represented schedules can be flexibly customized by designers, tool developers, and adaptive scheduling strategies. We have demonstrated the utility of the DSG representation through various examples with emphasis on demonstrating the utility across a heterogeneous variety of computing platforms.

We have applied the DSG model for schedule representation to develop a methodology for design and implementation embedded image processing applications on Multiprocessor Systems-On-Chip (MPSoC) platforms. We refer to this methodology as the DEIPS (DSG-based design and implementation of Embedded Image Processing Systems) methodology. The DEIPS methodology provides DSP system designers with an intuitive approach to specifying and exposing concurrency in DSP applications, while also providing control in how this concurrency is exploited, flexibility in experimenting with alternative scheduling strategies under resource constraints, and expressive power to represent a wide range of DSP applications, including applications that exhibit dynamic dataflow behavior. In the DEIPS methodology, a dataflow model of the application functionality is used to identify the computation-intensive, memory-demanding or I/O intensive parts, while a DSG model of the schedule is used to coordinate execution of the application on a given target platform.

We have demonstrated the DEIPS methodology using an image processing case study involving background subtraction. We have demonstrated how different forms of parallelism can be applied and experimented with for this application using the DEIPS methodology. Considering I/O interfacing and memory usage, we also demonstrated implementation details associated with this case study, with emphasis on those associated with I/O and memory management. Our experimental results on this case study showed up to 2.15 times improvement in computation time using our DEIPS methodology. We demonstrated the versatility of the DEIPS methodology by applying it to a second image processing case study, which involved image registration.

Dataflow models for DSP system design have been studied extensively in various contexts, including simulation, hardware implementation, and software implementation. However, as embedded DSP applications and platforms incorporate increasing levels of dynamics, it becomes important to revisit and augment the library of existing dataflow methods with new methods that can reliably and efficiently handle complex dynamic behaviors. Our work on the DSG provides a formal foundation for methodically incorporating such dynamics throughout the processes of design and implementation. In this thesis, we have demonstrated the efficacy and utility of the DSG model. This foundation can be applied to develop new techniques for mapping dataflow graphs into different classes of platforms in ways that rigorously integrate application- and schedule-level modeling. Useful directions for future work along these lines include the application of DSG as a substrate for optimized integration of hybrid scheduling techniques.

Our DEIPS methodology incorporates the Sequential Dataflow Schedule Graph (SDSG) and Concurrent Dataflow Schedule Graph (CDSG) representations as core com-

ponents of the design process. Developing back ends for optimized integration of the DEIPS methodology, and associated CDSG and SDSG transformations with relevant classes of platforms — including programmable digital signal processors, graphics programming units, field programmable gate arrays, and high performance, general purpose multi-core platforms (e.g., for DSP system simulation) — is an important area for future work.

Bibliography

- [1] OpenCV for Android website. <http://opencv.org/platforms/android.html>, visited on January 16, 2013.
- [2] M. Ade, R. Lauwereins, and J.A. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *Proceedings of the Design Automation Conference*, pages 64–69, June 1997.
- [3] C. Baloukas et al. Mapping embedded applications on MPSoCs: The MNEMEE approach. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 512–517, 2010.
- [4] S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman. Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. *IEEE Transactions on Signal Processing*, 43(6):1468–1484, June 1995.
- [5] B. Bhattacharya and S. S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 84–89, Paris, France, June 2000.
- [6] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001.
- [7] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, 2010.
- [8] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for DSP. *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, 47(9):849–875, September 2000.
- [9] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.
- [10] 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.
- [11] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.
- [12] J. Ceng et al. MAPS: An integrated framework for MPSoC application parallelization. In *Proceedings of the Design Automation Conference*, pages 754–759, 2008.
- [13] T. P. Chen, H. Haussecker, A. Bovyryn, R. Belenov, K. Rodyushkin, A. Kuranoc, and V. Eruhimov. Computer vision workload analysis: Case study of video surveillance systems. *Intel Technology Journal*, 9(2):109–118, 2005.

- [14] M. I. Cole. *Algorithmic Skeletons*. MIT Press, 1989.
- [15] J. Dean and S. Ghemawate. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [16] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.
- [17] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, pages 686–701, June 1993.
- [18] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput analysis of synchronous data flow graphs. In *Proceedings of the International Conference on Application of Concurrency to System Design*, June 2006.
- [19] B. Goldberg. Functional programming languages. *ACM Computing Surveys*, 28(1):249–251, 1996.
- [20] S. Ha and E. A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7), July 1997.
- [21] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubhr, A. Deyhle, A. Hadert, and J. Teich. A SystemC-based design methodology for digital signal processing systems. *EURASIP Journal on Embedded Systems*, 2007:Article ID 47580, 22 pages, 2007.
- [22] 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*, pages 37–49, Dallas, Texas, September 2005.
- [23] C. Hsu, J. Pino, and S. S. Bhattacharyya. Multithreaded simulation for synchronous dataflow graphs. *ACM Transactions on Design Automation of Electronic Systems*, 16(3):25–1–25–23, June 2011.
- [24] T. Isshiki, D. Li, and H. Kunieda. Multiprocessor SoC design framework on Tightly-Coupled Thread model. In *Proceedings of the International SoC Design Conference*, pages I–56–I–61, 20082.
- [25] C. Kao. Benefits of partial reconfiguration. *Xcell Journal*, 55:65–67, 2005.
- [26] J. Keinert, C. Haubelt, and J. Teich. Windowed synchronous data flow. Technical report, University of Erlangen-Nuremberg, 2005.
- [27] V. Kianzad and S. S. Bhattacharyya. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):667–680, July 2006.

- [28] B. Kienhuis and E. F. Deprettere. Modeling stream-based applications using the SBF model of computation. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 385–394, September 2001.
- [29] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere. Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. *IEEE Transactions on Signal Processing*, 55(6):3126–3138, June 2007.
- [30] P. Kuacharoen. *Embedded Software Streaming via Block Streaming*. PhD thesis, Georgia Institute of Technology, 2004.
- [31] S. Y. Kung, P. S. Lewis, and S. C. Lo. Performance analysis and optimization of VLSI dataflow arrays. *Journal of Parallel and Distributed Computing*, pages 592–618, 1987.
- [32] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *Journal of the Association for Computing Machinery*, 31(4):406–471, December 1999.
- [33] S. Le Beux, G. Nicolescu, G. Bois, Y. Bouchebaba, M. Langevin, and P. Paulin. Optimizing configuration and application mapping for MPSoC architectures. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, pages 474–481, 2009.
- [34] E. A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, January 2006.
- [35] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.
- [36] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.
- [37] J. Mcallister, R. Woods, R. Walke, and D. Reilly. Multidimensional DSP core synthesis for FPGA. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 43(2–3), June 2006.
- [38] M. D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the USENIX conference on Hot Topics in Parallelism*, pages 5–5, 2010.
- [39] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, August 2002.
- [40] W. A. Najjar, E. A. Lee, and G. R. Gao. Advances in the dataflow computational model. *Parallel Computing*, pages 1907–1929, 1999.

- [41] S. Neuendorffer and E. Lee. Hierarchical reconfiguration of dataflow models. In *Proceedings of the International Conference on Formal Methods and Models for Codesign*, June 2004.
- [42] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zisulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *Proceedings of the Design Automation Conference*, pages 574–579, 2008.
- [43] H. Oh, N. Dutt, and S. Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Proceedings of the Asia South Pacific Design Automation Conference*, pages 497–502, 2006.
- [44] P. R. Panda and N. D. Dutt. Memory architectures for embedded systems-on-chip. In *Proceedings of the International Conference on High Performance Computing*, pages 647–662, 2002.
- [45] M. Piccardi. Background subtraction techniques: a review. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 3099–3104, 2004.
- [46] 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*, pages 17–23, Monterey, California, June 2008.
- [47] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 157–166, Samos, Greece, July 2008.
- [48] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pages 640–645, Washington DC, USA, November 2010.
- [49] C. Shen, L. Wang, I. Cho, S. Kim, S. Won, W. Plishker, and S. S. Bhattacharyya. The DSPCAD lightweight dataflow environment: Introduction to LIDE version 0.1. Technical Report UMIACS-TR-2011-17, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011. <http://hdl.handle.net/1903/12147>.
- [50] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li. Heterogeneous computing. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1996.
- [51] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition, 2009.
- [52] C. Stauffer and W. E. L. Grimson. Adaptive background mixture models for real-time tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1999.

- [53] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer Magazine*, 23(6):12–24, 1990.
- [54] Texas Instruments, Inc. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor Data Manual*, February 2012.
- [55] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proceedings of the International Conference on Formal Methods and Models for Codesign*, July 2006.
- [56] P. van Stralen and A. Pimentel. Scenario-based design space exploration of MP-SoCs. In *Proceedings of the International Conference on Computer Design*, pages 305–312, 2010.
- [57] L. Wang, H. J. Siegel, and V. Roychowdhury. A genetic-algorithm-based approach for task matching and scheduling in heterogeneous environments. In *Proceedings of the Heterogeneous Computing Workshop*, pages 72–85, April 1996.
- [58] W. Wolf. *FPGA-Based System Design*. Prentice Hall, 2004.
- [59] H. Wu, H. Kee, N. Sane, W. Plishker, and S. S. Bhattacharyya. Rapid prototyping for digital signal processing systems using parameterized synchronous dataflow graphs. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 1–7, Fairfax, Virginia, June 2010.