REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
FEBRUARY 2014	Journal Article	NOV 2010 – NOV 2013

4. TITLE AND SUBTITLE

MAPPING PARAMETERIZED DATAFLOW GRAPHS ONTO FPGA PLATFORMS (PRE-PRINT)

5a. CONTRACT NUMBER FA8750-11-1-0062

5b. GRANT NUMBER N/A

5c. PROGRAM ELEMENT NUMBER 62788F

5d. PROJECT NUMBER T2MC

5e. TASK NUMBER UN

5f. WORK UNIT NUMBER ML

6. AUTHOR(S)

Hsiang-Huang Wu, Chung-Ching Shen, Nimish Sane, William Plishker, Shuvra S. Bhattacharyya (University of Maryland)
Hojin Kee (National Instruments)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Department of Electrical and Computer Engineering,
and Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/Information Directorate
Rome Research Site/RITB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-RI-RS-TP-2014-020

12. DISTRIBUTION AVAILABILITY STATEMENT

Distribution Approved For Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

This to be published in E-Reference Signal Processing Vol. 3, Sec 13, Article #24, Jan 2014. This work was funded in whole or in part by Department of the Air Force contract number FA8750-11-1-0062. The U.S. Government has for itself and others acting on its behalf an unlimited, paid-up, nonexclusive, irrevocable worldwide license to use, modify, reproduce, release, perform, display, or disclose the work by or on behalf of the Government. All other rights are reserved by the copyright owner.

14. ABSTRACT

This paper provides background on relevant methods for application modeling, and platform-based implementation of dynamically reconfigurable signal processing systems. New methods were developed based on an application modeling formalism called parameterized dataflow, along with techniques for mapping parameterized dataflow specifications onto Field Programmable Gate Array architectures. The proposed parameterized dataflow approach for design and implementation of dynamically reconfigurable signal processing systems provides a comprehensive framework that encompasses application modeling, task scheduling, and hardware mapping. These methods were demonstrated using case studies in the domains of wireless communication and computer vision.

15. SUBJECT TERMS

Dataflow graphs, dynamic reconfiguration, FPGA design, model-based design, parameterized dataflow, scheduling.

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
U	U	U	UU	44	STANLEY LIS
					19b. TELEPHONE NUMBER (*Include area code*) N/A

Mapping Parameterized Dataflow Graphs onto FPGA Platforms

Hsiang-Huang Wu[a], Chung-Ching Shen[a], Hojin Kee[b], Nimish Sane[a], William Plishker[a], Shuvra S. Bhattacharyya[a]

[a]*Department of Electrical and Computer Engineering, and*
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland, USA
{hhwu, cchsen, nsane, plishker, ssb}@umd.edu
[b]*National Instruments*
Austin, Texas, USA
hkee@ni.com

Abstract

Dynamic reconfiguration in digital 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.

In this paper, we provide background on relevant methods for application modeling, and platform-based implementation of dynamically reconfigurable signal processing systems. To help bridge the gap between these groups of methods, we develop new methods based on an application modeling formalism called *parameterized dataflow*, along with techniques for mapping parameterized dataflow specifications onto FPGA architectures.

Our proposed parameterized dataflow approach for design and implementation of dynamically reconfigurable signal processing systems provides a comprehensive framework that encompasses application modeling, task scheduling, and hardware mapping. We demonstrate our methods using case studies in the domains of wireless communication and computer vision.

Keywords:

Dataflow graphs, dynamic reconfiguration, FPGA design, model-based design, parameterized dataflow, scheduling.

1. Introduction

As the speed and logic capacity of field programmable gate arrays (FPGAs) have been improving steadily, FPGAs have become increasingly attractive for a wide variety of signal processing systems. FPGAs are increasingly employed in the form of *platform FPGAs*, which are integrated circuits that combine significant amounts of configurable logic fabric along with additional subsystems, such as application-specific accelerators, processor cores, memory blocks, and input/output interfaces, to facilitate FPGA-based, system-on-chip design [54]. FPGA fabric is also integrated into application specific integrated circuits (ASICs) to allow implementations that provide a mix of programmable and custom hardware (e.g., see [60]).

Through support for dynamic reconfiguration, modern FPGAs allow customization of hardware structures both statically and at run-time, 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 [22]).

However, in addition to such potential for improved operation, incorporating dynamic reconfiguration into the digital system design space also brings increased design complexity. Model-based design methodologies have been evolving steadily over the years to help address issues of design complexity in embedded systems [28]. In model-based design, applications are represented and analyzed in terms of formal models of computation, which promote analysis of functionality as well as hardware and software structure at a high level of abstraction. In the domain of signal processing, model-based techniques based on dataflow models of computation are particularly popular, and are employed in a growing variety of design tools [7].

While dataflow techniques allow for high level reasoning about and manipulation of application dynamics, there are important challenges in mapping dataflow models into FPGA platforms in ways that systematically and effectively exploit the dynamic reconfiguration capabilities of the platforms. This paper provides a review of state-of-the-art model-based design techniques and FPGA implemen-

tation techniques for signal processing systems, and explores the challenges involved in effectively mapping high level application models into efficient implementations on dynamically reconfigurable FPGA platforms.

The exploration presented in this paper on mapping models into implementations builds on our earlier work in this area, which was presented in preliminary form in [55]. The reconfiguration-aware mapping techniques presented in this paper (Sections 4-5) go beyond the developments of [55] in a number of ways. Specifically, this extended paper enhances the hardware architecture mapping methodology of [55] and provides two alternative perspectives on scheduling. These two perspectives affect important trade-offs between performance and modularity. An important new aspect integrated into one of these scheduling perspectives involves integration of the recently-developed *dataflow schedule graph* model [57] into processes for FPGA mapping of dynamically reconfigurable signal processing systems.

2. Background

2.1. FPGA Technology

As shown in Figure 1, an FPGA can be viewed as a matrix of cells, which can encapsulate various kinds of hardware structures, such as programmable logic subsystems, memories, special purpose hardware subsystems (e.g., multipliers or higher level signal processing accelerators), and embedded processor cores. A ring of I/O (input/output) modules is placed along the periphery for connection to the outside world, and routing channels, driven through configurable switches, are used for communication among cells.

Although the details of how programmable logic subsystems are constructed varies among different vendors, their basic structure, illustrated in Figure 2(a), comprises M-input look up tables (LUTs) and D flip-flops (DFFs), which are integrated into programmable logic blocks. Collections of such logic blocks can be programmed to implement digital logic functions of arbitrary complexity. Figure 2(b) illustrates, in simplified form, the structure of logic blocks that are employed in FPGA devices made by Xilinx and Altera. These blocks contain substructures for arithmetic and carry logic to support the flexible construction of computational building blocks.

To connect logic blocks, flexible routing architectures are provided to accommodate different kinds of interconnection patterns. The *island-style* routing architecture is widely adopted in commercial FPGA devices. An island-style global routing architecture involves routing channels on all four sides of the logic blocks.

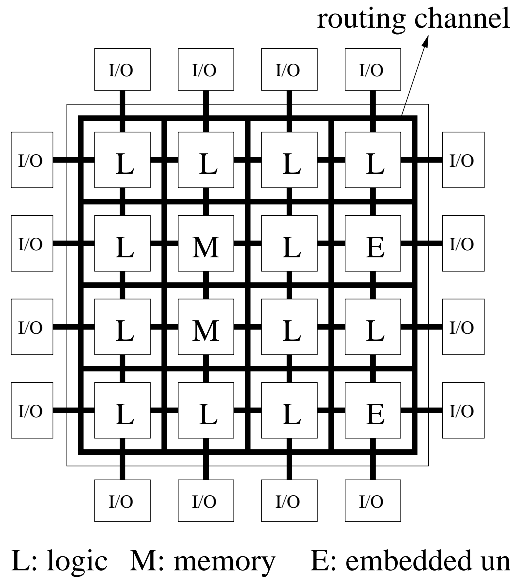


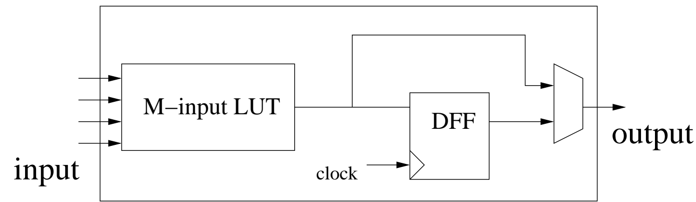
Figure 1: Basic architecture of an FPGA.

The numbers of wires contained in a channel is set to a constant W during fabrication, and is one of the key design decisions made in the FPGA architecture design. Island-style routing architectures generally employ wire segments of different lengths in each channel to allow optimized selection of lengths based on the specific connections that are made. The end points of wire segments are typically staggered so that logic blocks can be connected at the end points of wires that have appropriate lengths.

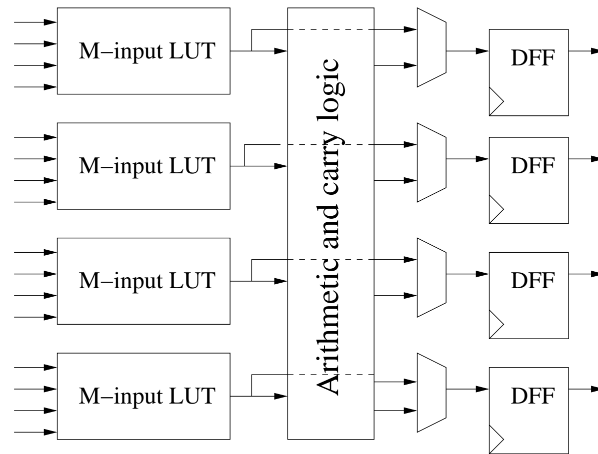
Figure 3 illustrates channels and switches in an island-style FPGA architecture, and also shows a routing example. The figure shows an interconnection that starts from logic block 5, and goes through logic block 4 to logic block 2. Two switch nodes, s1 and s2, are used. To connect logic block 5 to logic block 4, programmable switch D in switch node 2 is configured. Similarly, programmable switch B in switch node 1 is configured for the connection between logic block 4 and logic block 2.

There are six possible connections for a programmable switch, as illustrated in Figure 4, which is why six switches (A through F) are depicted in Figure 3.

The island-style architecture facilitates optimization of the physical layout of logic blocks and their surrounding routing channels, and its regularity facilitates efficient delay estimation (e.g., see [33]).



(a) Primitive structures for constructing programmable logic blocks.



(b) A simplified illustration of logic blocks employed in FPGA devices from Xilinx and Altera.

Figure 2: Programmable logic blocks.

2.2. Model-based Design of Signal Processing Systems

As described in Section 1, model-based techniques based on dataflow models of computation are used widely in the design of signal processing systems. Dataflow can be viewed as a special case of Kahn process networks (KPNs), which are composed of processes that communicate through unbounded communication channels [21]. Each communication channel is a first-in first-out (FIFO) queue of *tokens*, where tokens encapsulate data values as they pass between processes. FIFO communication channels in KPNs can be written to and read only from the processes that are connected to them in the enclosing KPN. An important restriction on execution of KPNs is that processes can access data from incident FIFOs only through blocking read operations, which effectively suspend the processes if requested data is not available, and allow the processes to resume only after

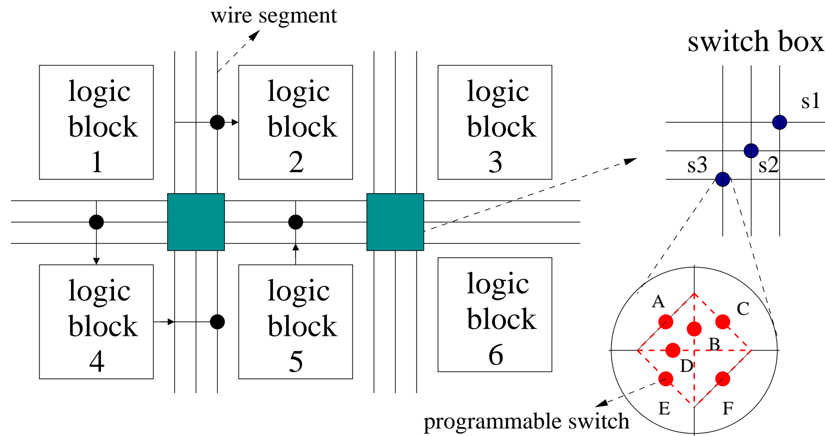


Figure 3: Island-style routing architecture.

the associate read operations are completed. This restriction helps to ensure determinacy of KPN-based representations. Variations on Kahn process networks have been explored in recent years, such as polyhedral process networks and reactive process networks, which provide useful new capabilities for modeling and analysis of process and network execution [52, 36, 12].

In the context of design and implementation of signal processing systems, *dataflow* is a restricted form of KPN in which processes (called *actors* in the context of dataflow representations) execute in terms of well-defined, discrete units of execution, called *firings*, of the associated actors [27]. Such a discrete approach to process firing can be enforced, for example, through *enable-invoke* semantics, where fireability (availability of sufficient input data) is fully separated from process execution (*invoking*) functionality [43].

Model-based design methods based on dataflow models of computation have provided designers of signal processing systems with representations of intuitive correspondence to signal processing block diagrams. In such representations, DSP applications are modeled as directed graphs, where vertices correspond to dataflow actors and represent computational modules for executing (or *firing*) the corresponding tasks, and edges represent inter-actor FIFO channels, as in KPNs. Actors produce and consume tokens from their input and output edges, respectively, as they are fired. In enable-invoke dataflow, an actor firing cannot begin execution until all of the input data required by the firing is present on the relevant input edges, and thus blocking reads are not employed [43].

Scheduling is a critical issue when implementing dataflow representations of

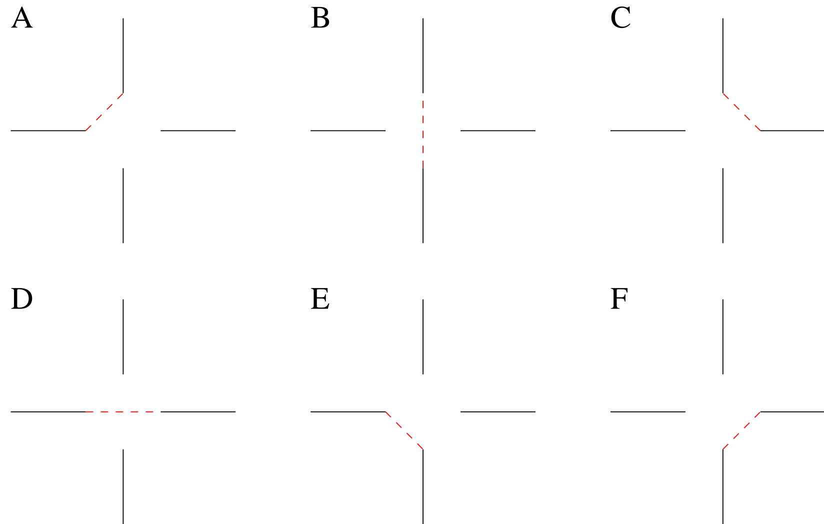


Figure 4: Six possible connections for a programmable switch.

signal processing systems. Scheduling is the process of constructing a *schedule*, which assigns each actor firing to a processing resource, and determines the ordering of firings that share the same resource. In addition to affecting overall system functionality (a schedule that is not constructed properly can cause deviations from expected functionality), scheduling generally has significant effect on performance, resource utilization, and memory requirements.

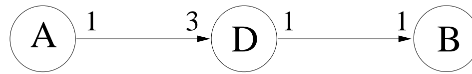
Synchronous dataflow (SDF), proposed in [26], is a restricted form of dataflow in which each actor firing consumes and produces constant amounts of data from each of its input and output edges, respectively. The SDF restriction is orthogonal to enable-invoke dataflow. In an enable-invoke context, an SDF firing can execute only after all of the required input data has arrived at the actor inputs. On the other hand, if enable-invoke semantics is not enforced, then even though the firing consumes constant amounts of data, it can begin execution when only part of its input data is available, and employ blocking reads to read the rest of the data — in a manner that is interleaved with the computations associated with the firing — until the firing is complete. An important advantage of SDF is its support for static scheduling, and a wide variety of scheduling techniques have evolved that exploit this support (e.g., see [26, 14, 13, 39, 20, 49]).

Cyclo-static dataflow (CSDF) [9] is one of the most popular extensions of SDF. CSDF generalizes SDF by allowing the consumption or production rate of an actor port to vary as long as the pattern of variations forms a periodic sequence

that is statically known.

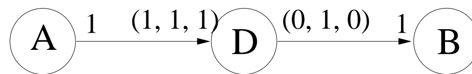
An example of an SDF actor is illustrated in Figure 5(a). Here, actor D represents a downsampler that consumes three tokens generated from actor A and transfers one among them onto the edge that is directed to actor B. An example of a schedule for this graph is $S_1 = (3A)DB$. This schedule is expressed in *looped schedule* notation, where each parenthesized term represents a *schedule loop* that is iterated a number of times specified by a positive integer iteration count that is given as the first item in the term [8]. Thus, the schedule S_1 corresponds to the firing sequence (A, A, A, D, B).

A CSDF version of this downsampler-based example is shown in Figure 5(b). Here, firings of actor D execute based on a periodic sequence of three distinct *phases*, and the inputs and outputs of the actor are annotated with the numbers of tokens consumed and produced in these phases. Thus, D consumes one token on each phase and produces one token on every third phase, starting with the second phase. This CSDF graph can be executed with the schedule $S_2 = ADADBAD$, which differs from S_1 in that less buffer space (one unit of storage versus three) is needed for the edge (A, D). Indeed, the potential for improved memory requirements is a useful feature of the CSDF model. A detailed comparison of SDF and CSDF is developed in [41]. In addition, CSDF can be integrated with parameterized dataflow (this will be discussed in the following section), and the resulting model is called parameterized cyclo-static dataflow (PCSDF). Such integration, explored in [15, 45], provides further flexibility in the modeling of application dynamics compared to PSDF.



schedule: (*3A*) *DB*

(a) SDF specification and a corresponding schedule.



schedule: *ADADBAD*

(b) CSDF specification and a corresponding schedule.

Figure 5: Alternative dataflow models for a simple multirate signal processing example.

2.3. *Parameterized Dataflow*

Parameterized dataflow is a meta-modeling technique that integrates dynamic parameterization into dataflow modeling in a systematic and general (applicable to different dataflow models) manner.

In parameterized dataflow, sets of actor parameters, domains in which such parameters can "reside", and configurations for such sets (bindings to actual values within the respective parameter domains) are important aspects of application models in addition to more conventional forms of dataflow information, such as specifications on token production and consumption rates associated with actor firings.

Settings or updates to parameter configurations can be distinguished as happening statically, pre-execution, or dynamically. Static configurations are determined at compile time — i.e., when the code for an implementation is derived. Pre-execution configuration refers to configuration that occurs after compile time but before a given execution of the application. In dynamic configuration or dynamic reconfiguration, parameter values can be initialized or updated while the application is executing.

For example, in an FPGA implementation, an addition component can be configured by instantiating a look-ahead adder or a ripple carry adder, depending on a trade-off assessment in terms of relevant area and performance constraints, and such an assessment-configuration sequence can in general be carried out statically, pre-execution or dynamically depending on factors such as the overhead of performing the configuration, and the degree to which this overhead is amortized by the benefits of applying the selected configuration.

In this paper, we focus on methods for dynamic reconfiguration, which are increasingly important in the development of signal processing systems that can adapt in response to dynamically changing application constraints, data characteristics, and other operating conditions [7].

The organization of the rest of this paper is summarized as follows. We first review hardware methods for dynamic reconfiguration. We then show how parameterized dataflow techniques integrated with the SDF model of computation can be applied as an abstract model for design and implementation of dynamically reconfigurable signal processing systems. This integrated model, called *parameterized synchronous dataflow* (*PSDF*), has been studied in a variety of useful design contexts before (e.g., see [3, 7]); in this paper, we present novel methods for applying this model to the systematic hardware mapping of dynamically reconfigurable signal processing systems.

Next, to represent adaptive schedules efficiently, we apply a schedule representation called the *dataflow schedule graph* (*DSG*), which provides a formal approach for representing dynamic interactions between applications and the architectures on which they execute. We show that using the DSG model, hardware mapping of PSDF can be interpreted naturally, and a structured path to efficient implementation can be achieved. Using our methods based on the parameterized dataflow, SDF and DSG representations, conventional, ad-hoc methods for dynamic reconfiguration can be replaced by formally-rooted, model-based techniques that promote efficiency, reliable integration, and modularity through a systematic, dataflow-based design flow.

3. Dynamic Reconfiguration Techniques in FPGAs

FPGAs are widely employed as signal processing platforms for both rapid prototyping and optimized implementation. Because they allow customization of digital hardware structures with significantly higher flexibility, lower cost, and lower turnaround time, they provide a valuable alternative to ASIC implementations when key application subsystems can be mapped efficiently into FPGA structures.

Hybrid architectures that integrate FPGA fabric within an ASIC have been explored to provide a wider range of trade-offs between efficiency and flexibility (e.g., see [60]). When applying such a hybrid approach, key challenges include the partitioning of designs into ASIC and FPGA parts, and integrating the ASIC and FPGA design flows. For example, the ASIC/FPGA partition determines the size (area) of the required FPGA subsystem, which in turns affects the FPGA placement. Furthermore, timing characteristics between the boundaries of the FPGA and custom logic subsystems complicate timing closure. For more details on such challenges and proposed solutions, we refer the reader to [24].

Using modern FPGAs, designers can exploit high speed *partial reconfiguration* technology (e.g., reconfiguration involving relatively small subsets of the blocks shown in Figure 1) to incorporate dynamic hardware reconfiguration into practical implementations. Our approach to applying dynamic reconfiguration involves careful selection of FPGA components that will be reconfigured at run time. The reconfigured circuits must satisfy the given area and timing constraints. That is, the FPGA logic block and routing channel structures illustrated in Figure 2 and Figure 3 need to be taken into account when determining the circuits that will be loaded in and out during run-time reconfiguration. While FPGA design flows for static configuration perform a full configuration that can program the entire

target FPGA, *partial reconfiguration* preserves the programming in some regions of the FPGA, and makes changes to other regions.

Commercial tools are available that provide support for dynamic reconfiguration. For example, PlanAhead, a tool from Xilinx, allows users to specify regions of a target FPGA that can be configured dynamically [59]. Communication between such regions and statically- or pre-execution-configured regions relies on the insertion of certain kinds of bus macros. In terms of the design hierarchy, regions set up for dynamic reconfiguration must be represented through top-level design modules.

Software programmable reconfiguration is a methodology that provides dynamic reconfiguration capabilities for FPGAs in a more software-oriented way — i.e., through interfacing with a soft-core processor [1]. Such dynamic reconfiguration is achieved, in a manner analogous to software context switching, by changing routing paths at run-time through control of the associated soft-core processor.

JBits SDK, another tool developed by Xilinx, contains a set of software modules and application programming interfaces to create Xilinx Virtex bitstreams from Java code [58]. Two tools for partial and remote reconfiguration that apply JBits SDK are presented in [37]. One is the *circuit customization tool*, which helps to create an interface to configure parameters of a circuit. The other is the *core unifier tool*, which allows designers to manipulate connections between cores using partial reconfiguration. When applying the JBits SDK, there are two types of cores called *controller* and *slave* cores. Controller cores are downloaded statically on to an FPGA, and can communicate with slave cores, which can be loaded dynamically. During execution, controllers can switch between slave cores or replace existing slave cores by downloading new ones.

To facilitate simulation of designs that employ dynamic reconfiguration, simulation techniques developed in [50] have been integrated into traditional FPGA design flow. This allows dynamic reconfiguration capabilities to be integrated into simulations that employ existing simulation tools, such as NCVerilog and ModelSim. In such simulations, circuits that are to be reconfigured (*reconfiguration circuits*) need to be specified and encapsulated up front. Such design capture for reconfiguration circuits helps to specify a *reconfiguration schedule*, which defines the conditions under which each reconfiguration circuit in the system is loaded or replaced. Specialized components, such as isolation switches and schedule control modules, are typically applied for the encapsulation, and reconfiguration, respectively.

Building on platform-based tools for dynamic reconfiguration, various design approaches have been developed for FPGA system design. For example, meth-

ods for real-time and power-aware implementation are developed in [2]. The application of dynamic instruction sets in soft-core microprocessors is explored in [53]. In this approach, if an FPGA cannot accommodate all of the relevant instructions with their associated hardware support, support for selected instructions can be configured dynamically as needed by the application. In [31, 32], a design methodology called *dynamic hardware/software partitioning* is proposed in which FPGAs are employed as coprocessors to accelerate computationally-intensive loops. In [48, 18], an architecture is developed that contains a processor, bit-level reconfigurable part (similar to conventional FPGA fabric), and components (*tiles*) for composing a novel form of reconfigurable subsystem called a *field programmable function array* (*FPFA*). Such tiles can be viewed as word-level, reconfigurable building blocks that are composed of ALUs and lookup tables. An FPFA is constructed from FPFA tiles to accelerate intensive, regularly-structured computations, such as linear interpolation or fast Fourier transforms. A class of heterogeneous processing arrays that integrate signal processors and FPGA subsystems, and are amenable to dataflow-based design and mapping techniques, is explored in [56].

Development of applications on platform FPGAs, FPGAs that employ soft core processors, and host-FPGA combinations often involves hardware/software co-design as a key aspect. Sophisticated algorithms have been developed for optimization of timing, area and energy in HW/SW systems. For example, the work presented in [47] takes as input a library containing general purpose processors, dynamically reconfigurable FPGAs, communication links, and memory modules, and applies an evolutionary algorithm to instantiate hardware resources, and assign tasks and communication events to the resources. A dynamic priority multirate scheduling algorithm determines the times at which the tasks and communication events in the system occur. A framework called *Nimble* is presented in [30]. This framework takes as input application specifications in C, and maps them into implementations on a heterogeneous platform that includes a general-purpose processor, an FPGA, and a memory hierarchy. An overview of various hardware/software co-synthesis approaches for signal processing systems is presented in [6].

Various methods have also been developed to help minimize overhead associated with dynamic reconfiguration. For example, in [16], methods are developed for evaluating the degree of computation-reconfiguration overlap in dynamically reconfigurable systems. These methods are based on modeling of the dynamic reconfiguration process, and identification of functional commonality between reconfiguration circuits. In another approach, an incremental elaboration model is

applied to streamline requests for new reconfiguration operations by using set theoretic techniques to leverage known characteristics of existing (currently active) configurations [11].

In this section, we have provided a brief overview of platform-based technologies and tools for dynamic reconfiguration in FPGAs. For more details on design flows for dynamically reconfigurable FPGA system implementation, we refer the reader to [35] and [44]. Methods for consistency analysis of dynamic reconfiguration functionality in dataflow graphs are discussed in [5, 38]. For comprehensive reviews of FPGA technology and system design methods, we refer the reader to [54, 46].

4. Modeling Dynamic Reconfiguration using PSDF Techniques

In the remainder of this paper, we develop methods for systematic mapping of model-based signal processing application representations into efficient implementations on dynamically reconfigurable hardware.

We apply 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 [4]. Compared to enable-invoke dataflow [43], PSDF has lower expressive power, but is equipped with streamlined scheduling techniques for the subclass of application models that are amenable to PSDF semantics. Compared to scenario-aware dataflow [51], PSDF can be viewed as having a more strict separation between data and parameters, which facilitates symbolic scheduling techniques based on parameterized looped schedules.

As described in Section 2.3, 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 [3]. 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 parameterized synchronous dataflow (PSDF).

Efficient quasi-static scheduling techniques have been demonstrated previously for PSDF specifications [4]. 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.

4.1. Execution of PSDF Graphs

The PSDF model allows the behavior of subsystems to be controlled by sets of parameters that can be configured dynamically. Such parameter control is modeled by mapping selected dataflow graph outputs of certain graphs, which are dedicated to computing parameter updates, to parameters of the graphs that they control. This coordination between parameter update computations and parameter reconfiguration operates under a carefully structured framework that promotes predictability and efficiency. Basic concepts associated with PSDF modeling and execution are outlined as follows.

- A *PSDF subsystem* consists three distinct PSDF graphs, called the *init*, *subinit* and *body* graphs.
- The interface dataflow behavior of a PSDF subsystem (i.e., the rates of token production and consumption at the subsystem inputs and outputs) can only be changed only by the init graph.
- The init graph can configure both the subinit and body graphs.
- The subinit graph is allowed to configure the body graph but not allowed to change the interface dataflow behavior of its enclosing subsystem.
- The body graph is executed immediately after the execution of the subinit graph.
- A hierarchical PSDF actor encapsulates a PSDF subsystem; such nesting in terms of PSDF *semantic hierarchy* an be arbitrarily deep based on how a design is constructed.

We use the downsampler shown in Figure 6 to illustrate these concepts. Here, node H represents a PSDF downsampler (i.e., a subsystem), which has two parameters, the *factor* and *phase*. These parameters represent the consumption rate F and the index P of the token that is selected (for transfer to the output port) among the F tokens that are consumed in a single execution of the downsampler. Thus, for example, if $F = 5$ and $P = 2$, then downsampling by a factor of 5 is performed, and on each execution, the downsampler outputs the second token from among the window of 5 tokens that it consumes.

The consumption or production rate associated with a subsystem input or output port is viewed as interface dataflow behavior, and can only be configured by the init graph (or kept fixed at a statically configured value), as described above. Thus, the *factor* F is configured by the init graph only whereas the phase P can be configured either by the init or subinit graph since it does not change the interface dataflow behavior.

In this example, P is configured by the subinit graph to allow a finer granularity of (more frequent) control compared to configuration by the init graph. Initially, actor E configures F to the value 3, which yields an SDF graph that maintains its given SDF properties while the parameter F remains fixed at this value. To execute the graph in this *SDF configuration*, we can apply any valid SDF schedule for the configuration — one such schedule is $(3A)BHC$. This schedule is repeated some number of times before the downsampler value is changed. In particular, changes must occur between iterations of this schedule, as governed by PSDF semantics so that within any given iteration the graph operates as an SDF graph, while the SDF graph configuration can be changed between iterations.

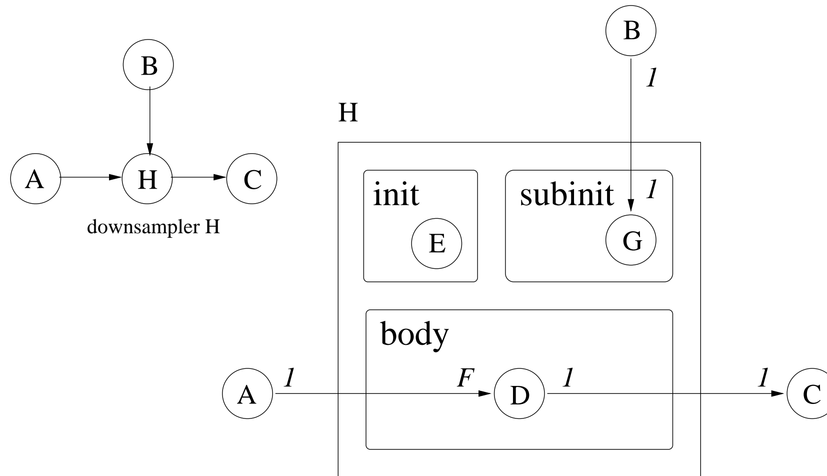


Figure 6: A PSDF-based downsampler example.

PSDF-based design and implementation is supported by a Java-based PSDF simulator, called *PSDFsim* [55], which provides modeling and functional simulation capabilities for PSDF specifications as part of the *dataflow interchange format* (*DIF*) environment [19].

4.2. PSDF Design Methodology

PSDF-semantics can be applied for model-based design at the front end of the FPGA/ASIC design flow shown in Figure 7. Such an approach provides a structured framework for modeling adaptive behaviors and dynamic reconfiguration, and deriving corresponding adaptations to scheduling strategies and resource allocations (e.g., see [4, 55]).

Our PSDF-based approach for FPGA system design involves two key phases — high level modeling and validation (*modeling*) and hardware architecture mapping (*mapping*). These two phases can in general be applied iteratively to implement and experimentally refine 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 code 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. Use of the formal dataflow methodology to encapsulate design components (actors) facilitates this reuse of testbenches. Similarly, edges in the DIF-based application 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.

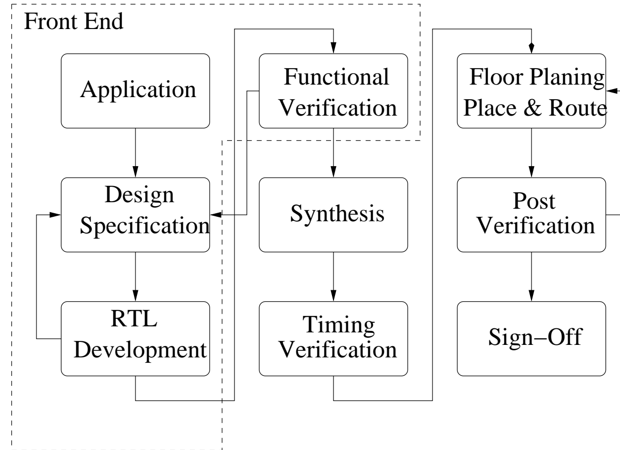


Figure 7: FPGA/ASIC design flow overview.

The simulation and implementation tools discussed in Section 3 focus on mapping hardware description language (HDL) programs into FPGA implementations. In contrast, our proposed methods show how to map higher level, model based specifications into monolithic, FPGA-targeted, HDL programs, which can then be further processed by tools such as those discussed in Section 3. In our experiments, we have not integrated the tools discussed in Section 3 in this way (i.e., as a back end to our proposed methods); this is a useful direction for further study.

5. Hardware Mapping

In this section, we present two hardware architecture mapping methods that apply PSDF modeling, and support dynamic reconfiguration with two useful objectives — modularity and performance optimization. Both of these methods exploit the modular design representation format facilitated by PSDF, which is discussed in Section 5.1. We present a novel form of co-design between PSDF application *modeling* and *scheduling* in Section 5.2, and we demonstrate the utility of this approach in deriving efficient, model-based implementations of dynamically reconfigurable signal processing systems.

5.1. Modular Mapping

We develop a systematic approach for mapping PSDF specifications into hardware implementations. Because of natural correspondences that are used between dataflow design objects (actors and edges) and corresponding hardware structures, as well as between specialized PSDF modeling features and their implementations, the approach provides a high degree of modularity. In this modular approach, implementations are composed in terms of smaller building blocks that can be tested independently and integrated precisely through our mapping of PSDF semantics into hardware control.

Previous work on mapping dataflow structures into hardware include the work on VLSI dataflow arrays [23], multidimensional arrayed dataflow [34], and SystemC [17]. The methods developed in this paper 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 [4, 3]. Due to the potential for applying parameterized dataflow semantics with arbitrary dataflow models of computation (subject to suitable definitions of graph iterations), the integration of the techniques presented in this paper 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. Table 1 summarizes our methodology for deriving such mappings.

Table 1: Mapping PSDF constructs to hardware.

PSDF Modeling Components	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 control for scheduling and parameter management for the blocks.

This allows for significant reuse of parameterized HDL “glue code”, as well as corresponding streamlining of verification effort.

We employ self-timed scheduling and control of dataflow actors within a PSDF context. In a such a self-timed approach, actors can fire as soon as they have sufficient data on their input ports, have access to sufficient empty buffer slots on their output ports, and have their current parameter values available, as determined by the associated subinit and init graphs. Such self-timed hardware mapping is natural for signal processing oriented dataflow models of computation (e.g., see [7]). The modularity of the approach is enhanced because the controllers for individual actors are structured independently of any global scheduling control, which allows scheduling strategies to be changed efficiently, conveniently, and reliably. In this approach, only loop counts associated with actor control vary with changes in the schedule control, and such adaptation of loop counts can be carried out naturally as actor parameter updates through the overall framework of parameterized dataflow.

Figure 8 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 *prod 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 9. The controllers associated with the structures of Figure 8 and Figure 9 are illustrated in Figure 10. In comparison with the circuit block, the other hardware components are relatively less complicated, and to provide flexibility, we do not constrain the implementations of these components to any particular styles. For example, FIFOs can be implemented using D Flip-Flops or SRAMs. The subsystem controllers and graph controllers (i.e., the controllers for the init, subinit and body graphs) follow PSDF operational semantics and the generated schedule to guide execution of the graph controllers and the circuit blocks, respectively. These two types of controllers are finite state machines in which control remains in a given state while there is no triggering input. For instance, a graph controller remains in the *EXE* state until the corresponding circuit block completes execution.

The circuit block control, illustrated in Figure 10(a), is a key part of our proposed method for self-timed, PSDF hardware implementation. Such a circuit block control structure provides control for an individual PSDF actor. At the beginning of a control iteration (the state labeled *PARAM*), the circuit block con-

figures any dynamically managed parameters based on the current settings and attempts to consume data from the actor input port. 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 encapsulated functional unit to process the input data and generate any output values. When the output data is ready, the *prod circuit* pushes the output data onto the corresponding output edges in the *PROD* 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 transitions back to the PARAM state and waits for another circuit block iteration before proceeding; otherwise, the controller transitions to the CONS state to consume tokens for the next firing.

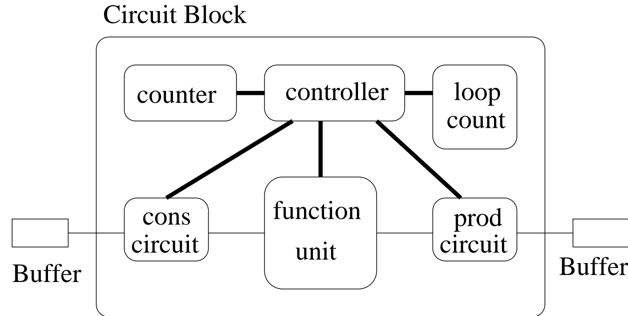


Figure 8: Interface and control architecture for a circuit block.

Based on the modularity of our hardware mapping approach, described earlier, the control structures for actors need to be configured only by setting their respective loop counts. Such loop counts can be derived and validated efficiently at the functional prototyping stage, using the PSDFsim tool in conjunction with techniques to determine *repetitions vectors* of specific SDF configurations [55]. The repetitions vector of an SDF graph gives the number of times each actor needs to be fired in a periodic schedule for the graph [26].

Thus, the overall design flow involves applying PSDFsim for functional prototyping, performing systematic hardware mapping using the approach described in this section, and then synthesizing and deploying the resulting self-timed implementation on the target FPGA.

5.2. Schedule-Based Mapping

Effective scheduling is important in deriving efficient implementations of dynamically reconfigurable signal processing systems. However, scheduling in the

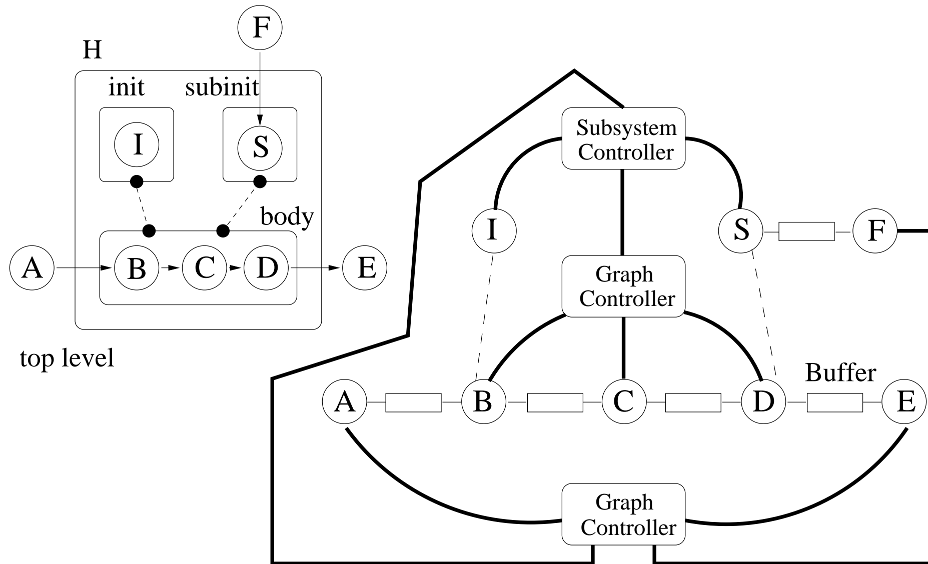


Figure 9: An illustration of subsystem-level hardware mapping.

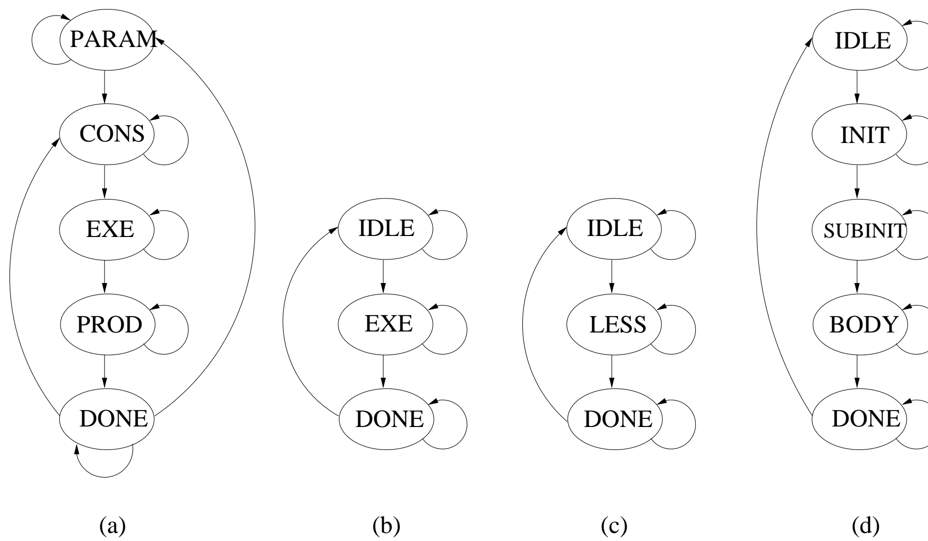


Figure 10: Finite state machines for (a) a circuit block, (b) a graph controller, (c) consumption and production circuits, and (d) a subsystem controller.

presence of dynamic reconfiguration is challenging because of the increased dynamics in the scheduling process, as well as the increased difficulty in modeling and manipulating schedules whose structures change during execution.

The *dataflow schedule graph* (*DSG*), proposed in [57], is a dataflow based schedule representation that helps to address these challenges. The DSG can be viewed as a model for representing schedules of dataflow graphs that is itself rooted in dataflow semantics. This allows an integrated modeling approach among applications, schedules, and their interactions.

DSG-based modeling of actors centers around two special kinds of actors, called *schedule control actors* (*SCAs*) and *reference actors* (*RAs*). A DSG for a given processor (i.e., the DSG-based schedule model for a given processor) can have at most one token at any given time within the graph. This token serves to enable the next actor to be executed on the processor. The restriction that the total token count be bounded by 1 enforces the constraint that a processor can execute at most one task at any given time. Here, a "processor" can represent any computational resource that executes actors in a dedicated (exactly one actor assigned to the resource) or time-multiplexed manner.

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. A basic form of RA is one that simply performs a guarded execution of the actor that it points to. A *guarded execution* of an actor does nothing if the actor does not have sufficient data on its inputs to complete its next firing; if sufficient input data is available, a guarded execution executes a single firing of the actor. However, more capabilities — beyond just performing guarded executions — can be incorporated into RAs using the optional auxiliary computations mentioned above.

Table 2 gives examples of several types of SCAs and 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 token in the enclosing DSG, 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, $case$ and $endcase$ provide DSGs with the capability of selecting computations conditionally. The number of outputs for a given $case$ actor must match the number of inputs to the corresponding $endcase$ actor to provide conditional selection of the computations that are enclosed by the matching $case$ and $endcase$ pair.

Table 2: Examples of SCAs.

SCA	# of inputs	# of outputs
$loop$	2	2
$case$	1	≥ 2
$endcase$	≥ 2	1

As described earlier, 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.

The execution of a PSDF specification involves careful coordination, based on details of PSDF semantics, among the init, subinit, and body graphs within the reconfigurable subsystems that are enclosed by the specification. By modeling this PSDF-driven coordination in terms of DSGs, we can precisely represent PSDF execution (i.e., the operational semantics of PSDF) in terms of pure dataflow concepts, thereby enabling analysis and manipulation of schedules based on dataflow techniques rather than having to rely on specialized PSDF-based methods. Moreover, such a PSDF to DSG transformation allows PSDF graphs to be implemented through reuse of dataflow techniques rather than through specialized implementation structures that are derived for PSDF. Such a transformation thus combines the high level modeling flexibility and analysis potential offered by PSDF with streamlined paths to implementation offered through the use of the DSG as an intermediate representation.

A general DSG model for PSDF execution is illustrated in Figure 11. Parameter reconfiguration is achieved through communication between RAs and application actors through DSG tokens that encapsulate updated parameter values. For example, if the init graph changes the value of a parameter associated with a body graph actor A, the DSG token can "carry" this value (e.g., within a list of pending parameter updates) to A for reconfiguration. Once the body graph is executed, and the DSG token "reaches" the RA that encapsulates A, the parameter update

can be “unpacked” from the DSG token and applied to A before A executes. A similar approach can be used to achieve parameter control of the subinit graph by the init graph, and of the body graph by the subinit graph.

Thus, using the DSG model, processes associated with dynamic reconfiguration can be abstracted in a way that precisely and flexibly represents the relevant functionality while hiding platform-specific implementation details about how the reconfiguration is achieved. For example, whether the init graph stores parameter values in registers or memory, and how the DSG token “points to” such storage locations to reference the associated configuration settings, are left as implementation details that can be refined from the DSG-based model.

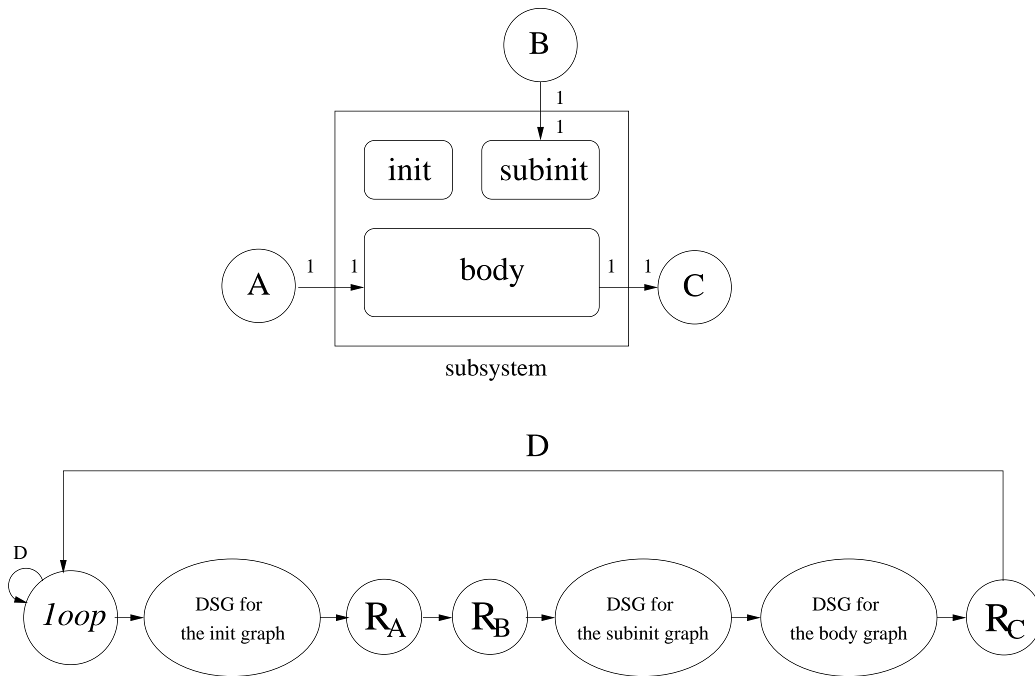


Figure 11: Modeling PSDF execution using DSGs.

DSGs can be applied not only as a target for automated schedule generation techniques, but also as a model in which designers specify, experiment with, and iteratively refine schedules. Because they are rooted in familiar dataflow principles, rather than specialized or esoteric scheduling notations, designers can work with DSG representations using well understood modeling concepts. In such a way, designers can experiment flexibly with schedules that interact with application actors, and apply platform-based features for dynamic reconfiguration. By

providing such flexibility and facilitating such experimentation, DSG-based hardware mapping of PSDF graphs can help designers explore complex design spaces associated with dynamically reconfigurable signal processing systems, and tailor implementations based on the specific implementation constraints for a given application.

In the following section, we explore this integrated PSDF- and DSG-driven design methodology using two case studies that involve relevant signal processing applications.

6. Case Studies

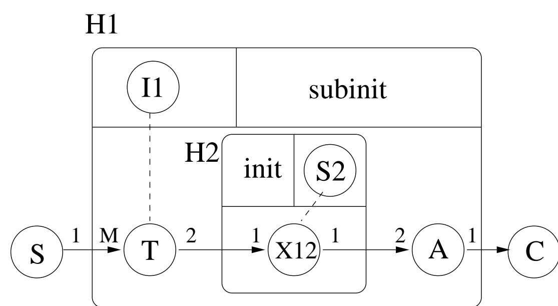
In this section, we present two case studies with which we concretely demonstrate our proposed methods for model-based implementation of dynamically reconfigurable signal processing systems.

6.1. Reconfigurable Phase-shift Keying

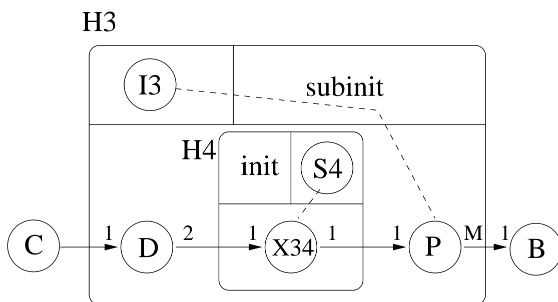
First, we demonstrate our PSDF-based design methodology and modular hardware mapping techniques using a reconfigurable *phase-shift keying* (*PSK*) application that can be configured as binary PSK (BPSK), quadrature PSK (QPSK) or 8PSK. We construct PSDF models of the modulator and demodulator for this system, and develop Java-based functional DIF code to specify the internal functionality of each actor. The resulting PSDF program is 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 derived hardware.

Figure 12 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 multiplications are required for each input sample.

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



(a) PSK modulator.



(b) PSK demodulator.

Figure 12: PSDF-based models of PSK modulator and demodulator.

8PSK, respectively, is effectively activated. After the system model is constructed, we use 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 13. 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 $S2$.

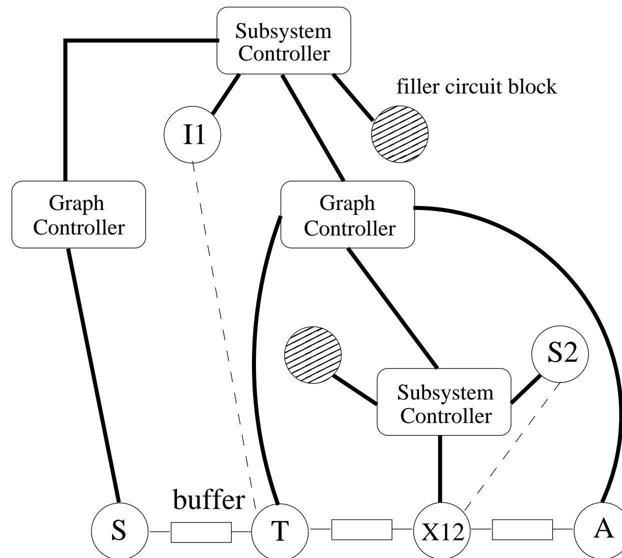


Figure 13: Hardware mapping for modulator.

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, which is derived using PSDFsim and our proposed design methodology. Synthesis re-

sults generated by the Cadence Encounter RTL Compiler are shown in Table 3. 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 3: Comparisons for PSK modulator system.

Area of PSDF design and SDF design (modular hardware mapping)		
PSDF (cell)	SDF (cell)	Reduction
20004	33602	40.47% (1.68X)

This modular hardware mapping approach is readily applied due to its generality, and is also useful as it provides a standard method to realize hardware implementations of PSDF graphs. Our schedule-based hardware architecture mapping approach using DSGs 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.

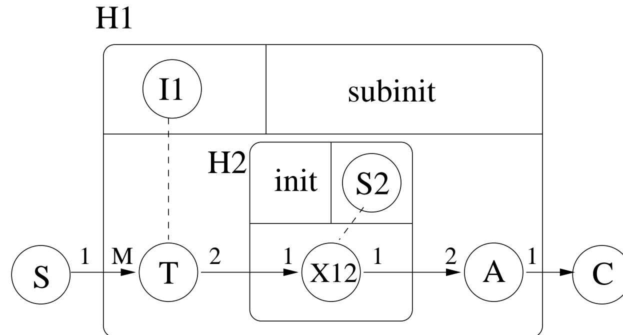


Figure 14: Reconfigurable PSK modulator.

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 15(b) illustrates a DSG representation for the reconfigurable PSK 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 schedule-based 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 4 shows the improvement in area that is achieved by the streamlined DSG representation compared to the modular PSDF-to-hardware mapping approach of Section 5.1. 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 4: Area comparison for reconfigurable PSK modulator under constant speed (100 MHz).

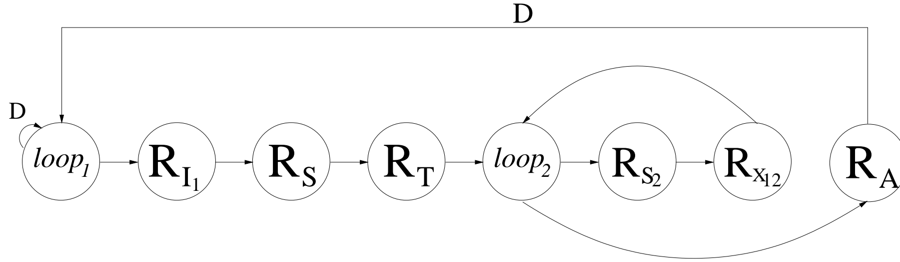
Hardware mapping			
	schedule-based	modular	Reduction
Area (cell)	18949	20004	5.27%

6.2. Foreground/Background Extraction

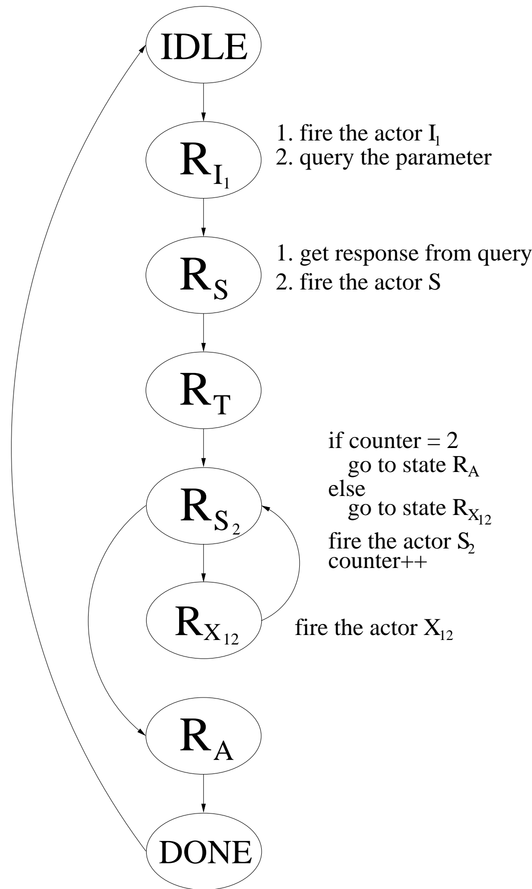
Video surveillance is widely used for security enhancement and environmental monitoring. As the demand for video surveillance increases, the volume of data that must be analyzed for surveillance applications increases dramatically as well. Pattern recognition helps to incorporate automation in this analysis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

To be effective, pattern recognition techniques are often task specific with significant fine tuning of system configurations and algorithm parameters in terms of the kinds of data being analyzed and the objectives of the analysis. At the same time, the vast amount of data that needs to be processed in typical applications can make software-based implementation (e.g., using MATLAB and C) impractical.

Considering these two issues — the need for task-specific tuning and high performance — PSDF mapping to FPGAs provides a potential solution method, where both parameter adaptation and high performance hardware mapping can be supported and optimized through an integrated design process.



(a) A DSG for the reconfiguration PSK modulator of Figure 14.



(b) An FSM for the DSG in Figure 15(a).

Figure 15: Hardware architecture mapping for a DSG.

In a workload analysis study of video surveillance systems, it has been shown that the most expensive computation is foreground/background (FG/BG) extraction [10]. We demonstrate a general PSDF model of FG/BG extraction that can accommodate a variety of FG/BG extraction algorithms. This model is shown in Figure 16.

The FG/BG extraction algorithms represented by this model 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 [42].

Our PSDF model shown in Figure 16 contains two subsystems, which are used to specify algorithms for video preprocessing and FG/BG extraction. The video preprocessing subsystem here can be viewed as an auxiliary subsystem, which processes raw data, and transforms it into a form that is appropriate for the extraction algorithms and the underlying processing platforms. In subsystem 2, actor `switch` passes video frames to actor `BG_model` and `current_frame` in the training phase. At that point, actor `FG_extractor` produces no foreground. In the differentiating phase, actor `switch` stops sending frames to actor `BG_model`, and continues to send frames to actor `current_frame`. If a pixel of the current frame exceeds the corresponding threshold value, actor `FG_extractor` indicates that the pixel is part of the foreground.

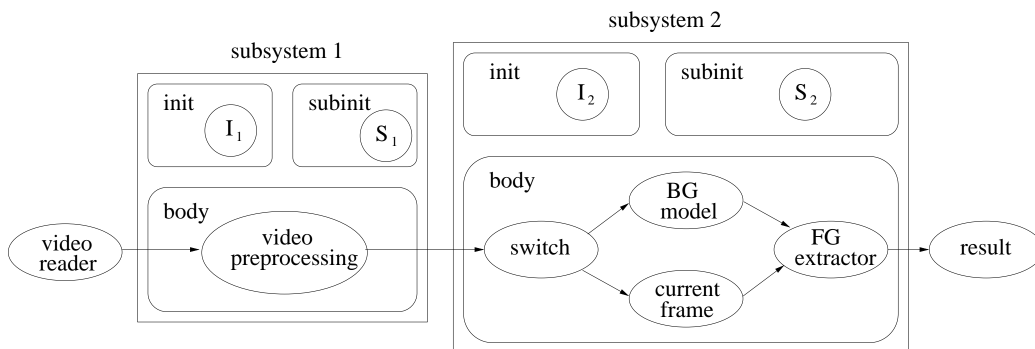


Figure 16: A general PSDF model for FG/BG extraction.

We apply a specific FG/BG extraction algorithm — the running average algorithm — using the general PSDF model of Figure 16. The running average algorithm averages the values of the pixels in the training frames to create the BG model. The target implementation platform for our experiments with the running average algorithm is the Xilinx Spartan 3E Starter (XC3S500E). RS232 and VGA ports are selected as the input and output interfaces, respectively.

Since the total capacity of block RAM (BRAM) on the target platform is only 360K bits, only one monochrome 640X480 (307,200 bits) frame can be accommodated. Here, every bit represents one pixel and the difference between the same pixel of two frames is either 0 or 1. For natural mapping from our general PSDF model of FG/BG extraction, we need one storage subsystem for the BG model and another other one for the current frame.

In this implementation of the model in Figure 16, actors `BG_model` and `current_frame` "own" their associated image storage and are controlled by actor `switch`. Actor `FG_extractor` takes two frames from actor `BG_model` and `current_frame` for differentiation and determines which parts of the frame are foreground. A non-uniform array of threshold values, shown in Figure 17, is derived from the training processes. The actual threshold values represented in Figure 17 are regarded as parameters, which can be adapted based on video stream characteristics.

In this thresholding approach, multiple pixels are grouped into individual blocks, and the sum (number of 1-valued pixels) in a block is computed to characterize the block and compare it with the corresponding block-based threshold. These block-based thresholds characterize entire blocks with a single operations — that is, the entire block is characterized as foreground and background based on the associated threshold comparison. For example, if the pixel sum associated with `block 2` is larger than t_2, the block is classified as being part of the foreground. For more detailed background on this thresholding approach, we refer the reader to [25].

In our experiments, the threshold values are derived from the process of BG model training and stored in actor `BG_model`. One block is composed of eight pixels. The parameters of subsystem 2 are summarized as follows.

- Init graph:
    1. baud rate of RS232 receiver,
    2. number of frames for `BG_model` training;
- subinit graph:

1. switching on/off the path from actor `switch` to `BG_model`,
2. threshold value of discrimination between FG and BG.

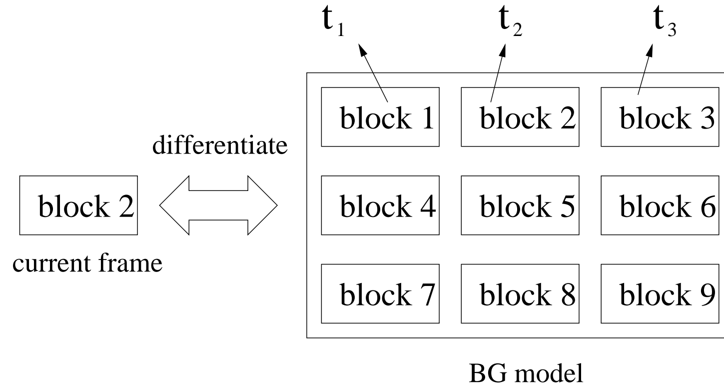


Figure 17: A non-uniform array of threshold values.

The overall implementation involves heterogeneous design languages and platforms — subsystem 1 is implemented in MATLAB and executes on a host PC, subsystem 2 is implemented in Verilog and executes on the targeted FPGA, and the dataflow edge between the `video_preprocessing` and `switch` components represents the RS232 channel, where the transmitter and receiver are in the `video_preprocessing` and `switch` components, respectively. The baud rates of the transmitter and receiver should be consistent. The `video_preprocessing` component selects frames and converts them into monochrome format based on luminance levels. The modular hardware mapping process developed in Section 5 is adopted throughout the implementation process. The parameters of subsystem 1 are summarized as follows.

- Init graph:
  1. baud rate of RS232 transmitter,
  2. luminance level;
- subinit graph:
  1. frame selection.

Figure 18 shows our experimental setup. We use MATLAB to read the video from files on the host platform, and divide the video data into frames. To reduce

the level of serial communication, we select every tenth frame and convert it to monochrome format based on a luminance level threshold of 0.3. The threshold value is set to 4, which was the value that we obtained from the training process described in Section 6.

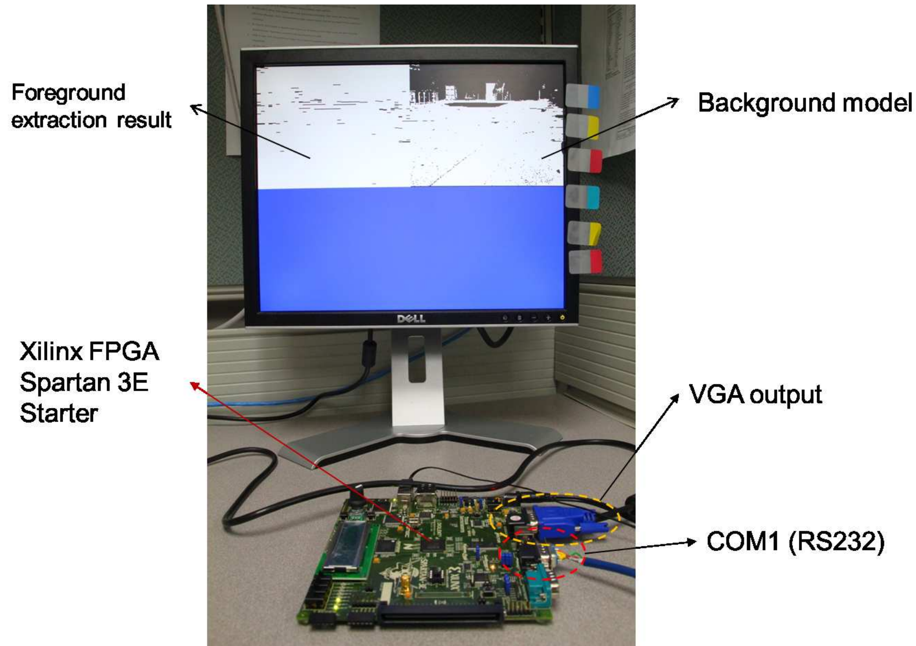


Figure 18: Foreground/background extraction on FPGA, and associated experimental setup.

Results from our experimentation are illustrated in Figure 19 and Figure 20. Figure 19 shows six frames from a video sequence. Figure 19 (a) shows the common background scene for the sequence, while Figures 19 (b-f) show frames in which a man runs from left to right. These six frames, after processing by our implementation of foreground/background extraction on the FPGA, are shown in Figures 20 (a-f), respectively. Here, the red rectangle indicates the subtracted foreground. The existence of black pixels outside of the subtracted foreground is due to the perturbation of trees caused from ambient breeze. Distortion from this phenomenon can be reduced by a sophisticated algorithm [29]. Our results shown in Figures 20 (a-f) demonstrate that the image of the running man can be extracted correctly as foreground.

Table 5 summarizes synthesis results obtained when deriving the FPGA implementation. The maximum frequency is 84.062MHz. By the maximum frequency, we mean the clock frequency that the FPGA logic can execute at without violat-

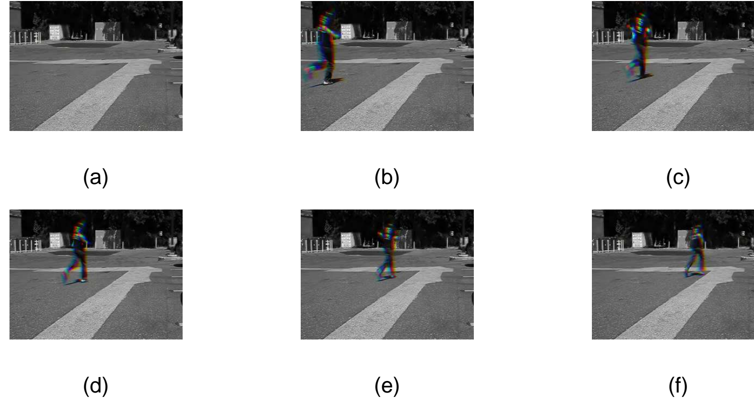


Figure 19: Selected frames from a video sequence.

ing the timing constraints. The video frame rate is set in our experiments to 30 frames/second. The utilization of block RAMS (BRAMs) is high since they are used to store the video frames, whereas the utilization of FPGA slices is relatively low because the running average algorithm does not require complex computation.

Table 5: Device utilization summary.

Selected Device: 3s500efg320-4				
Number of slices:	133	out of	4656	2%
Number of Slice Flip Flops:	90	out of	9312	0%
Number of 4 input LUTs:	245	out of	9312	2%
Number of IOs:	16			
Number of bonded IOBs:	11	out of	232	4%
Number of BRAMs:	16	out of	20	80%
Number of MULT18X18SIOs:	1	out of	20	5%
Number of GCLKs:	1	out of	24	4%

In summary our experiments involving foreground/background extraction on an FPGA demonstrate the correctness and completeness of our proposed PSDF-based approach for FPGA mapping on a practical video processing system. Tuning application parameters at run-time is an important feature for advanced image processing applications, which we seek to support in this work. However, conven-

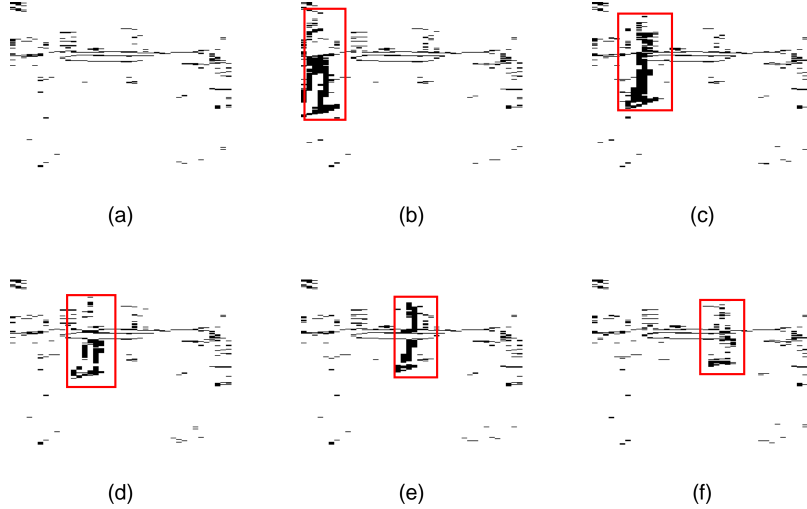


Figure 20: Results for Figure 19 after processing by our FPGA-based implementation of foreground/background extraction.

tional dataflow approaches, including SDF, do not allow such run-time parameter tuning. For this reason, we have focused in our experiments on the PSDF model, and novel application of this model to dynamically parameterized image processing on FPGAs. The experiments demonstrate the capability of the PSDF model to express the behavior of this application, and show that the abstract properties of PSDF, which provide formal, model-based manipulation of scheduling and dynamic reconfiguration, can be integrated with platform-specific details required to achieve a fully operational implementation.

7. Conclusions

In this paper, we have motivated the use of dynamically reconfigurable hardware platforms for signal processing systems, and have presented background on hardware methods for dynamic reconfiguration. We have then motivated how parameterized dataflow techniques integrated with the synchronous dataflow model of computation, which results in the parameterized synchronous dataflow (PSDF) modeling approach, can be applied as an abstract model for design and implementation of dynamically reconfigurable signal processing systems.

We have demonstrated a PSDF-based design methodology and associated simulation tool, called PSDFsim, for design and implementation of signal processing systems on dynamically reconfigurable platforms. We have also demonstrated the use of dataflow schedule graphs as a formal model for representing and manipulating hardware mappings of PSDF graphs throughout the design process. We have discussed the use of these methods to help streamline the processes of rapid prototyping, heterogeneous system design, hardware mapping, 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 SDF graphs or other kinds of design subsystems, the PSDF-based approach that we have presented provides for well-structured integration of parameter management into the SDF framework. This leads to more efficient and reliable techniques for application of dynamically reconfigurable platforms.

Important directions for further work include exploration of hardware mapping techniques for more general forms of parameterized dataflow, such as parameterized cyclo-static dataflow and parameterized fractional rate dataflow [40, 9, 45], and techniques for mapping parameterized dataflow graphs into platform FPGAs considering more thoroughly the available sets of heterogeneous resource groups (e.g., hard and soft core processors and application-specific accelerators).

References

[1] Altera. FPGA run-time reconfiguration: Two approaches. White paper, Altera, March 2008.

[2] J. Becker, M. Hübner, and M. Ullmann. Run-time FPGA reconfiguration for power-/cost-optimized real-time systems. *IFIP International Federation for Information Processing*, 200:119–132, 2006.

[3] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of DSP systems. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1948–1951, Istanbul, Turkey, June 2000.

[4] 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.

[5] B. Bhattacharya and S. S. Bhattacharyya. Consistency analysis of reconfigurable dataflow specifications. In E. F. Deprettere, J. Teich, and S. Vassiliadis, editors, *Embedded Processor Design Challenges*, Lecture Notes in Computer Science, pages 1–17. Springer, 2002.

[6] S. S. Bhattacharyya. Hardware/software co-synthesis of DSP systems. In Y. H. Hu, editor, *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, pages 333–378. Marcel Dekker, Inc., 2002.

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

[8] S. S. Bhattacharyya and E. A. Lee. Memory management for dataflow programming of multirate signal processing algorithms. *IEEE Transactions on Signal Processing*, 42(5):1190–1201, May 1994.

[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] T. P. Chen, H. Haussecker, A. Bovyrin, R. Belenov, K. Rodyushkin, A. Kuranov, and V. Eruhimov. Computer vision workload analysis: Case study of video surveillance systems. *Intel Technology Journal*, 9, 2005.

[11] A. Derbyshire, T. Becker, and W. Luk. Incremental elaboration for runtime reconfigurable hardware designs. In *Proceedings of the International Conference on Compilers, architecture and synthesis for embedded systems*, pages 93–102, New York, NY, USA, 2006. ACM.

[12] M. Geilen and T. Basten. Reactive process networks. In *Proceedings of the International Workshop on Embedded Software*, pages 137–146, September 2004.

[13] 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.

[14] R. Govindarajan, G. R. Gao, and P. Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing*, 31(3):207–229, July 2002.

[15] F. Haim, M. Sen, D. Ko, S. S. Bhattacharyya, and W. Wolf. Mapping multimedia applications onto configurable hardware with parameterized cyclo-static dataflow graphs. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages III–1052–III–1055, May 2006.

[16] J. Harkin, T. M. McGinnity, and L. P. Maguire. Modeling and optimizing run-time reconfiguration using evolutionary computation. *ACM Trans. Embed. Comput. Syst.*, 3:661–685, November 2004.

[17] 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.

[18] P. M. Heysters, J. Smit, G. J. M. Smit, and P. J. M. Havinga. Mapping of DSP Algorithms on Field Programmable Function Arrays. In *Proceedings of the International Workshop on Field-Programmable Logic and Applications*, pages 400–411, London, UK, 2000. Springer-Verlag.

[19] 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.

[20] C. Hsu, M. Ko, S. S. Bhattacharyya, S. Ramasubbu, and J. L. Pino. Efficient simulation of critical synchronous dataflow graphs. *ACM Transactions on Design Automation of Electronic Systems*, 12(3):21, 2007. 28 pages.

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

[22] C. Kao. Benefits of partial reconfiguration. *Xilinx Xcell*, 55:65–67, 2005.

[23] 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.

[24] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, 2006.

[25] R. Laganiere. *OpenCV 2 Computer Vision Application Programming Cookbook*. Packt Publishing, 2011.

[26] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[27] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, pages 773–799, May 1995.

[28] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. 2011. `http://LeeSeshia.org`, ISBN 978-0-557-70857-4.

[29] L. Li, W. Huang, I. Y. H. Gu, and Q. Tian. Foreground object detection from videos containing complex background. In *Proceedings of the ACM International Conference on Multimedia*, pages 2–10, New York, NY, USA, 2003. ACM.

[30] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the Annual Design Automation Conference*, pages 507–512, New York, NY, USA, 2000. ACM.

[31] R. Lysecky and F. Vahid. A configurable logic architecture for dynamic hardware/software partitioning. In *Proceedings of the Conference on Design, Automation and Test in Europe*, page 10480, Washington, DC, USA, 2004. IEEE Computer Society.

[32] R. Lysecky and F. Vahid. A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 18–23, Washington, DC, USA, 2005. IEEE Computer Society.

[33] P. Maidee, C. Ababei, and K. Bazargan. Timing-driven partitioning-based placement for island style FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):395–406, March 2005.

[34] 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.

[35] E.J. McDonald. Runtime FPGA Partial Reconfiguration. In *Aerospace Conference, 2008 IEEE*, pages 1 –7, March 2008.

[36] S. Meijer, H. Nikolov, and T. Stefanov. Throughput modeling to evaluate process merging transformations in polyhedral process networks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 747–752, 3001 Leuven, Belgium, 2010. European Design and Automation Association.

[37] D. Mesquita, O Moraes, J. Palma, R. Mőller, and N. Calazans. Remote and partial reconfiguration of FPGAs: tools and trends. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, pages 22–26, 2003.

[38] 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.

[39] H. Oh, N. Dutt, and S. Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Proceedings of the International Conference on Asia and South Pacific Design Automation*, pages 497–502. IEEE Press, 2006.

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

[41] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cyclo-static dataflow. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1995.

[42] M. Piccardi. Background subtraction techniques: a review. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, volume 4, pages 3099 – 3104, oct. 2004.

[43] 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.

[44] D. Rupe. An FPGA framework supporting software programmable reconfiguration and rapid development of SDR applications. SDR Forum Technical Conference, November 2007.

[45] S. Saha, S. Puthenpurayil, and S. S. Bhattacharyya. Dataflow transformations in high-level DSP system design. In *Proceedings of the International Symposium on System-on-Chip*, pages 131–136, Tampere, Finland, November 2006. Invited paper.

[46] R. Sass and A. G. Schmidt. *Embedded Systems Design with Platform FPGAs: Principles and Practices*. Morgan Kaufmann Publishers Inc., 2010.

[47] L. Shang, R. P. Dick, and N. K. Jha. SLOPES: Hardware/software cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(3):508 –526, March 2007.

[48] G. J. M. Smit, P. J. M. Havinga, L. T. Smit, P. M. Heysters, and M. A. J. Rosien. Dynamic reconfiguration in mobile systems. In *Proceedings of the Conference on Field Programmable Logic and Applications*, pages 162–170. Springer Verlag, 2002.

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

[50] J. Stockwood and P. Lysaght. A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays. *IEEE Transactions on VLSI Systems*, 4:381–390, 1995.

[51] 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.

[52] S. Verdoolaege. *Handbook of Signal Processing Systems*, chapter 4. Springer, first edition, 2010.

[53] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 99–107, 1995.

[54] W. Wolf. *FPGA-Based System Design*. Prentice Hall, 2004.

[55] 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.

[56] H. Wu, C. Shen, S. S. Bhattacharyya, K. Compton, M. Schulte, M. Wolf, and T. Zhang. Design and implementation of real-time signal processing applications on heterogeneous multiprocessor arrays. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 2121–2125, Pacific Grove, California, November 2010. Invited paper.

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

[58] Xilinx. JBits SDK. http://www.xilinx.com/products/jbits/, 2004.

[59] Xilinx. *PlanAhead User Guide, v 11.4*. Xilinx, Inc., 2009.

[60] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel. A hybrid ASIC and FPGA architecture. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 187–194, New York, NY, USA, 2002. ACM.