In Proceedings of the International Symposium on Rapid System Prototyping, Fairfax, Virginia, June 2010.

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

Hsiang-Huang Wu, Hojin Kee, Nimish Sane, William Plishker, Shuvra S. Bhattacharyya
Department of Electrical & Computer Engineering
University of Maryland, College Park, Maryland, USA
{hhwu, hjkee, nsane, plishker, ssb}@umd.edu

Abstract

Parameterized Synchronous Dataflow (PSDF) has been used previously for abstract scheduling and as a model for architecting embedded software and FPGA implementations. PSDF has been shown to be attractive for these purposes due to its support for flexible dynamic reconfiguration, and efficient quasi-static scheduling. To apply PSDF techniques more deeply into the design flow, support for comprehensive functional simulation and efficient hardware mapping is important. 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 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. Based on this simulation tool, we also present a systematic design methodology for applying PSDF to the design and implementation of digital signal processing systems, with emphasis on FPGA-based systems for signal processing. We demonstrate capabilities for rapid and accurate prototyping offered by our proposed design methodology, along with its novel support for PSDF-based FPGA system implementation.

1 Introduction

Dataflow modeling is widely used in the design and implementation of signal processing systems. 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 simultaneously execution of multiple actors if sufficient input data and sufficient resources are available [1].

As the complexity of digital signal processing (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 [2], 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 are developed in the context of SDF (e.g., see [3]).

Although an important class of useful DSP applications can be modeled efficiently 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 [4], scenario-aware dataflow [5],

and enable-invoke dataflow [6] 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 tradeoffs is provided in [7]. In this paper 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 [8].

PSDF can represent cyclo-static dataflow with the restriction that parameter values assocaited with cyclo-static dataflow rates should be changed periodically. Compared to enable-invoke dataflow, 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* [9]. 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 metamodel 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 [8]. 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 runtime 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 [6]. Functional DIF is based on a general (Turing complete) dataflow model of computation called *enable-invoke dataflow* (EIDF). Functional DIF allows actors whose internals are programmed in Java to be integrated with EIDF-based dataflow graphs that are specified in the *dataflow interchange format* (DIF). DIF is a textual language for specifying dataflow graphs in terms of arbitrary dataflow models of computation [10].

In this paper, 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. 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 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.

2 PSDF Background

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 [8].

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) [2].
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 subinit and body graphs. Such parameter changes are achieved by mapping the associated parameters to appropriate actor output ports in the init graph.

5. The subinit 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 subinit graph executes; new parameter values computed 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 [8, 11].
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 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 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 [8].

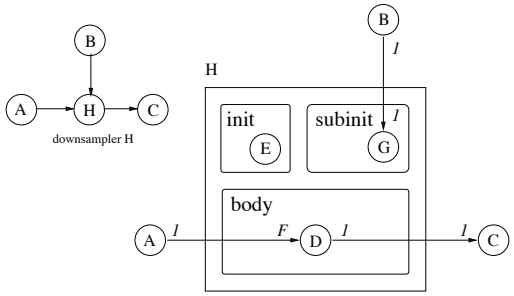

Figure 1. A PSDF Downsampler.

3 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 4.

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. 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 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 over-

ride 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.

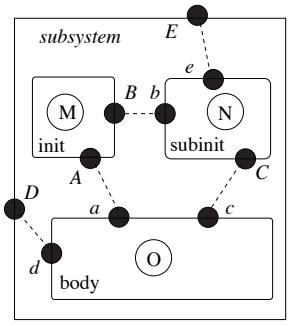

Figure 2. An illustration of parameter propagation in PSDFsim.

4 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 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.

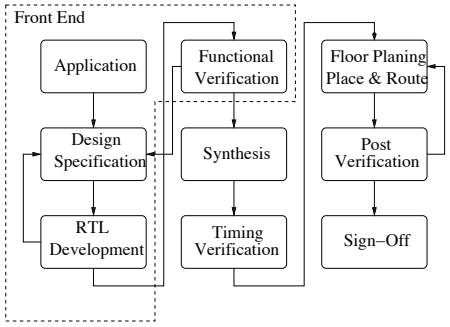

Figure 3. FPGA/ASIC design flow overview.

5 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 [12], SystemC [13], and multidimensional arrayed dataflow [14]. 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 [8]. 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 Section 1), 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. 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 1 summarizes this mapping.

Table 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 streamlining 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 subinit 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 [7]).

Figure 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 5. The dashed lines indicate wires for parameter configuration, and the circuit blocks B and D are parameterized by the init and subinit graph, respectively. The controllers associated with the structures of Figure 4 and Figure 5 are illustrated in Figure 6.

The circuit block control, illustrated in Figure 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.

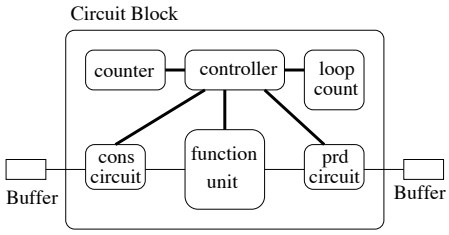

Figure 4. Interface and control architecture for a circuit block.

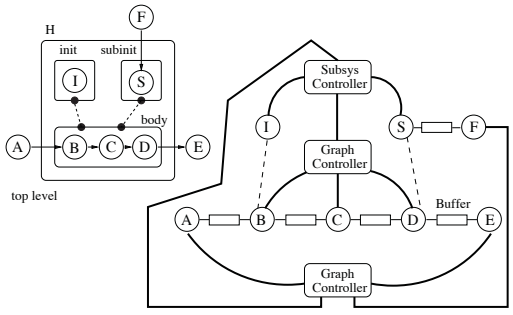

Figure 5. An illustration of subsystem-level hardware mapping.

6 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

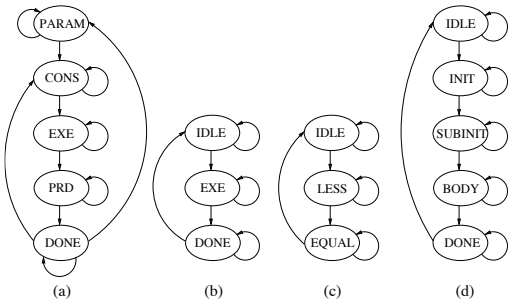

Figure 6. Finite state machines for (a) a circuit block, (b) a graph controller, (c) consumption and production circuits, and (d) a subsystem controller.

(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 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 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 multiplications are required for each input sample.

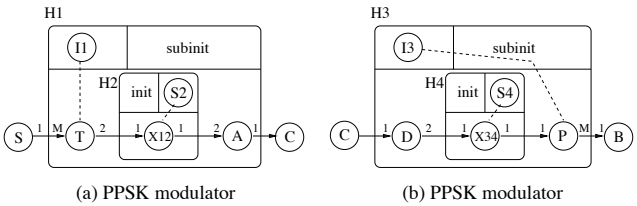

Figure 7. PSDF-based model of PSK modulator and demodulator.

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 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.

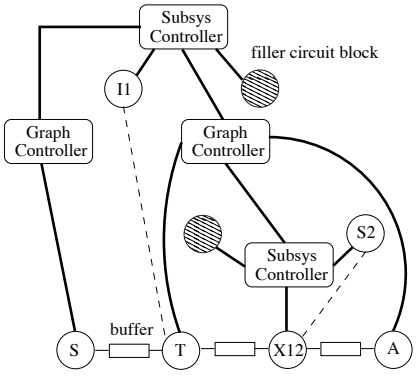

Figure 8. Hardware mapping for modulator.

A comparison of the simulation time for the PSK modulator between PSDFsim and ModelSim SE 6.5 is shown in Table 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 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. 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. 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)

7 Conclusion

We have demonstrated a design methodology and associated simulation tool, PSDFsim, for design and implementation of reconfigurable signal processing systems. We have demonstrate the use of these 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 SDF graphs or other kinds of design subsystems, the 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 plan to release PSDFsim as part of the DIF project [10, 15], which provides open-access tools — primarily in the form of Java-based `jar` files — that can be used for compiling, analyzing, and extending specifications in the dataflow interchange format (the DIF language).

8 Acknowledgments

This research was sponsored by the U. S. National Science Foundation, Award Number 0720596, and the Laboratory for Telecommunications Science.

References

[1] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model", *Parallel Computing*, pp. 1907–1929, 1999.

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

[3] 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*, vol. 47, no. 9, pp. 849–875, September 2000.

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

[5] 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.

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

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

[8] 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*, Paris, France, June 2000, pp. 84–89.

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

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

[11] 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.

[12] S. Y. Kung, P. S. Lewis, and S. C. Lo, "Performance analysis and optimization of VLSI dataflow arrays", *Journal of Parallel and Distributed Computing*, pp. 592–618, 1987.

[13] 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*, vol. 2007, pp. Article ID 47580, 22 pages, 2007.

[14] 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*, vol. 43, no. 2–3, June 2006.

[15] "DIF project website", `http://www.ece.umd.edu/DSPCAD/dif/index.htm`.