In Proceedings of the International Conference on Signal Processing, pages 402-408, Beijing, China, October 2012.

Systematic Integration of Flowgraph- and Module-Level Parallelism in Implementation of DSP Applications on Multiprocessor Systems-on-Chip

Zheng Zhou, Chung-Ching Shen, William Plishker, Hsiang-Huang Wu, Shuvra S. Bhattacharyya
Department of Electrical & Computer Engineering
University of Maryland, College Park
Email: (zhengzho, ccshen, plishker, hhwu, ssb)@umd.edu

***Abstract*—Increasing use of multiprocessor system-on-chip (MPSoC) technology is an important trend in the design and implementation of signal processing systems. However, the design of efficient DSP software for MPSoC platforms involves complex inter-related steps, including data decomposition, memory management, and inter-task and inter-thread synchronization. These design steps are challenging, especially under strict constraints on performance and power consumption, and tight time to market pressures. To facilitate these steps, we have developed a new dataflow based design flow within the targeted dataflow interchange format (TDIF) design tool. Our new MPSoC-oriented design flow, called *TDIF-PPG*, is geared towards analysis and mapping of embedded DSP applications on MPSoCs. An important feature of TDIF-PPG is its capability to integrate *graph level parallelism* for DSP system flowgraphs and *actor level parallelism* for DSP functional modules into the application mapping processing. Here, graph level parallelism is exposed by the dataflow graph application representation in TDIF, and actor level parallelism is modeled by a novel model for multiprocessor dataflow graph implementation that we call the *parallel processing group* (*PPG*) model. We demonstrate our approach through actor and subsystem design for software defined radio.**

I. Introduction

As multicore processor technology evolves, increasing numbers of processors are integrated into system-on-chip (SoC) devices for signal processing system implementation. The trend towards multiprocessor SoCs (MPSoCs) is motivated by the performance gain from efficient parallel execution of programs. This performance gain is determined in part by the amount of parallelism exposed from the program.

Digital signal processing (DSP) applications are often specified in terms of dataflow graphs, which provide high level, model-based views of systems being designed (e.g., see [1]). The parallelism exposed from such high level dataflow representations includes the following three forms.

1) Data parallelism: an actor (dataflow graph functional component) performs the same computation on different units of data.
2) Control/task parallelism: multiple actors execute different tasks on the same or different data.
3) Temporal parallelism (pipeline parallelism): multiple instances of the same actor execute simultaneously, where the instances correspond to different iterations of the enclosing dataflow graph.

These forms of dataflow modeling parallelism can all be viewed as *graph level parallelism*. A significant body of work has been developed to help expose and exploit graph level parallelism from dataflow models [4], [12], [11]. Relatively less attention has been given to exploiting parallelism *within* actors (*actor level parallelism*) within an enclosing dataflow framework.

Actor level parallelism provides optimization opportunities for enhancing performance beyond graph level parallelism. What is more challenging is the effective integration of graph level parallelism and actor level parallelism within an overall system-level optimization framework. However, without suitable models and tools to facilitate this exploration, procedural language compiler techniques, such as data decomposition, memory management, and inter-task and inter-thread synchronization, need to be developed from scratch to effectively exploit both actor level parallelism and graph level parallelism. On the other hand, general purpose parallel programming models, such as OpenMP and MPI, are designed for use across arbitrary application domains. Such generality can enhance convenience, but does not allow designers to thoroughly exploit specialized properties of their targeted application areas, such as the coarse grain dataflow structure (i.e., the signal processing flowgraph structure) of DSP applications [1].

In the DSP domain, dataflow models are widely used to specify, analyze, and simulate DSP applications. A variety of dataflow techniques have been developed for DSP applications to target problems such as buffer size optimization, scheduling, and cross platform porting. In this paper, we present a dataflow-based design flow, called *TDIF-PPG*, for design and implementation of parallel software targeted to MPSoC devices. TDIF-PPG extends the capabilities of the *targeted dataflow interchange format* (*TDIF*) [10] design environment with methods for expressing intra-actor parallelism, and associated capabilities for platform independent design, and early-stage performance evaluation. TDIF-PPG applies and systematically integrates both graph level parallelism and actor level parallelism. As a key component of TDIF-PPG, we propose a novel actor design technique called the *parallel processing group (PPG)*.

TDIF-PPG provides a flexible design environment without compromising the types of parallelism that can be exploited. TDIF-PPG achieves this by providing a breadth of formal models spanning graph and actor level parallelism. This approach to actor and system design allows the designer to exploit trade-offs among multiple factors, such as the number of utilized cores, buffer usage, throughput, and latency. The features of TDIF-PPG also provide schedulers more opportunities to achieve better system performance.

II. Related Work and Background

A. Core Functional Dataflow

Core functional dataflow (*CFDF*) is a deterministic sub-class of enable-invoke dataflow [8], which is a dynamic dataflow model that can express both static and data-dependent dataflow behaviors. In CFDF, actors are specified as sets of modes, where each mode has a fixed production and consumption rate associated with each actor output and input port, respectively. On each CFDF firing (actor invocation), an actor operates based on a unique *current mode*, which is maintained as part of the actor state. During each firing, in addition to consuming input tokens and producing output tokens, the actor

selects one mode from it set of modes as the *next mode*, which will be applied as the current mode in the next firing of the actor.

B. Related Work

Mapping an application to an MPSoC platform based on conventional methods is an error prone and time consuming process involving multiple steps. These steps include (1) decomposing a program into computational units and dividing these units into balanced threads; (2) managing the resulting inter-task and inter-thread communication and synchronization; and (3) handling resource management (memory, processors, interconnection bandwidth, etc.). A variety of layered models have been proposed to help hide hardware complexity from programmers, and allow advanced automation techniques to take over parts of the burden in the design process.

Jerraya et al. [5] discuss a design methodology based on a hardware/software layered interface and suggest a three-layered interface. This interface includes the parallel programming model, hardware dependent software, and hardware abstraction layer. Ceng [2] proposes a novel compiler technique using the tightly-coupled thread (TCT) model. This approach uses sequential C code as input, and automatically generates a parallel executable with minimal user guidance for a specialized architecture that supports the TCT model. Mignolet et al. [7] develop an MPSoC mapping flow that takes sequential C code as input, and generates parallel code in terms of concurrent C threads. The user is required to designate the segments of code that are to be selected for parallelization, the number of threads that execute these segments, and a high level description of the memory hierarchy on the target platform. The resulting parallel C code is then compiled for the target platform. Kwon [6] introduces the Common Intermediate Code (CIC) layer to interface software and hardware. Software is first partitioned and then written in CIC using generic application programming interfaces (APIs) for inter-task communication and synchronization. Later, the CIC translator translates the CIC code to platform-dependent code with the required hardware information. After that, scheduling code for the different processors is generated.

In contrast to prior work, the primary contribution of this paper is a novel framework for systematically integrating dataflow graph level parallelism and actor level parallelism into the design and implementation process for multiprocessor DSP systems. Key details on the novelty and utility of our contribution are as follows. (1) We provide a clear separation between graph- and actor-level parallelism, which enables the utilization of different forms of DSP parallelism at different levels of abstraction (e.g., parallelism across distinct filters versus parallelism across different taps of the same filter). At the graph level, dependencies between actors is loose and decoupled, while at the actor level, the synchronization and communication is more tightly coupled to exploit features for exploiting fine-grained parallelism in DSP-oriented processors. (2) Our proposed PPG model allows DSP system designers to flexibly explore different combinations of data parallelism, task parallelism and temporal parallelism within individual actors. (3) When both graph-level parallelism and actor-level parallelism are exposed, our TDIF-PPG framework provides comprehensive APIs to implement schedules that efficiently manage the resulting inter-task and inter-thread communication and synchronization on the target platform. (4) Our TDIF-PPG framework provides a unified abstraction layer of the underlying hardware platform. This abstraction layer helps to hide hardware complexity from programmers, and automatically generate code to handle resource management. Our efficient support for such an abstraction layer is especially useful given the diversity of processor families that are relevant for DSP system design.

III. TDIF-PPG Design Flow

The PPG plug-in adds two extra layers to the previous two layer TDIF design flow, as shown in the four-layer design flow demonstrated in Fig. 1.

In layer 1 — the *system layer* — the given DSP application is modeled as a CFDF graph using the DIF language. The DIF parser takes the CFDF graph as input, and constructs a corresponding model in the DIF intermediate representation, which helps to expose graph level parallelism.

In layer 2 — *the actor interface layer* — actor interface specifications, including information about input and output ports, actor parameters, and CFDF modes, for individual actors are provided using the TDIF language. Then the TDIF compiler parses the TDIF specifications for the actors, and generates equivalent actor API code in the targeted actor implementation language (C with optional extensions in the current version of TDIF-PPG). The generated APIs provide prototypes for actor interface functions, including functions for accessing the ports, modes, and parameters of each actor, as well as invoking and testing firability of the actors.

In layer 3 — *the platform-independent mode specification* (*PIMS*) layer — the PPG model guides the programmer in exposing actor level parallelism within the functional specification for each mode of an actor. The programmer creates parallel threads for actor modes, and describes the corresponding inter-thread communication and synchronization using generic PPG APIs. This provides a generic implementation of the associated actor that is not tied to any specific parallel platform, and thus, facilitates portability across platforms.

In layer 4 — *the actor implementation layer* — generic actor specifications from the PIMS layer are integrated with optimized platform-specific PPG API and run-time implementations. This integration is performed automatically by the *TDIF-PPG Code Synthesis Engine*. Third-party profiling tools are integrated in our design flow with layer 4 experimentation to provide measurements on final actor implementations, and associated feedback to help refine higher levels of the overall design flow. Note that multiple feedback loops (the dashed lines in Fig. 1) are provided in the design flow to allow for feedback at different levels with trade-offs between exploration speed and accuracy. The feedback loop in layer 3 is much faster but less accurate than the feedback loop in layer 4.

To produce a complete system implementation, the TDIFSyn package takes the DIF intermediate representation as input and generates the top-level C language implementation file and associated APIs for actor scheduling [10]. The automatically generated top-level C file initializes the operational contexts of actors and FIFOs, configures actor parameters, lays out the graph topology by instantiating connections between actor ports and their incident FIFOs and calls a user-defined scheduler that utilizes the APIs for actor scheduling and PPG scheduling. It is the responsibility of this user-defined scheduler, which can be provided by a programmer or constructed by a tool, to utilize the PPG scheduling APIs correctly, and ensure that the PPGs inside an actor are scheduled correctly. In our present implementation of the TDIF-PPG design flow, we develop the user-defined scheduler modules by hand (i.e., they are provided by a programmer). Integrating tools into TDIF-PPG for automated schedule construction is a useful direction for future work.

The given user-defined schedule along with the TDIF-PPG run-time library and the actor implementations are integrated automatically through glue code that is synthesized by the *TDIF Software Synthesis Engine*. System profiling tools can then be applied on

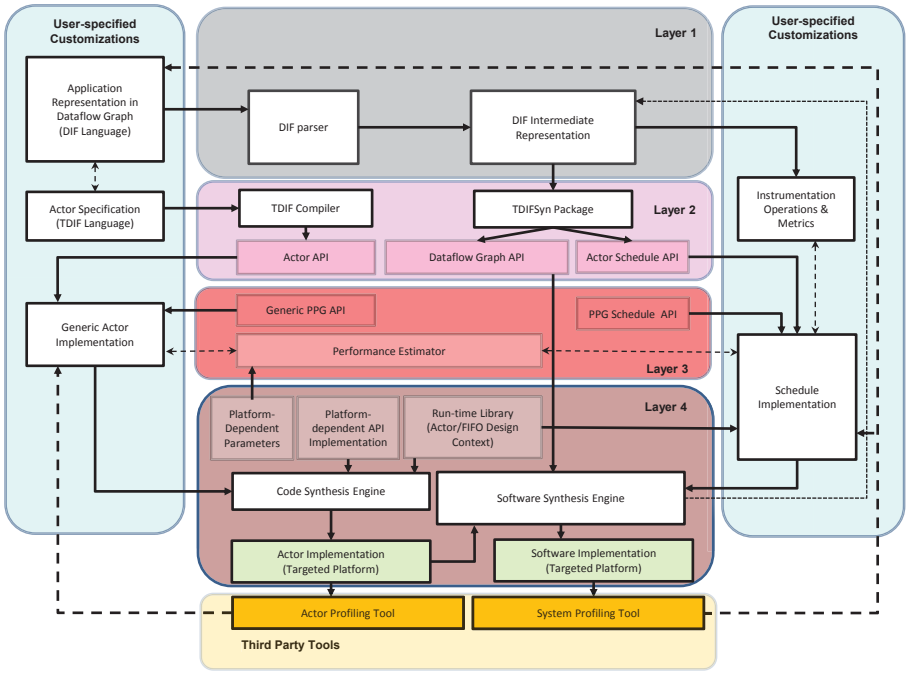


Fig. 1. TDIF-PPG design flow.

the generated implementations to validate whether or not system constraints are satisfied, and provide feedback for tuning of the schedule or higher levels of the design hierarchy, all the way up to the application model provided in layer 1.

In summary, the TDIF-PPG design flow enhances the retargetability of designs across different platforms by allowing designers to provide platform independent actor specifications, and automatically generating optimized implementations for different parallel platforms. Such retargetability is useful in efficiently exploring design options, and porting designs across platforms — e.g., to upgrade to newer hardware generations or provide alternative design versions that are targeted to different types of hardware, such as alternative versions for low cost, and high performance. Also, the provisions in the TDIF-PPG design flow for quick feedback across different design layers helps to reduce total software development time. Furthermore, the TDIF-PPG design flow uniquely takes both graph level parallelism and actor level parallelism into account for system optimization.

IV. Parallel Processing Group

A. Model Description

In an abstract sense, a PPG is a set of threads associated with computations within a dataflow graph actor. A PPG either contains a single thread (*single thread PPG*) or contains multiple threads that can be executed in parallel. Each thread in a PPG contains a set of data objects and a single code segment (a task or function that implements the thread). In a PPG, inter-thread synchronization and communication are restricted to three basic methods: *broadcast*, *barrier* and *point to point* (*P2P*), which are discussed in Section IV-B. We plan to extend support to other synchronization/communication methods in our future work.

An actor can contain any non-negative number of PPGs. The PPGs inside an actor are connected (in a logical sense) with FIFOs. SIMD and MIMD execution styles are both supported by our concept of PPGs. In addition to the *thread information* (the data objects and code segments), a PPG includes a *PPG execution context* for managing execution of the contained threads. APIs for PPG-based actor design are introduced in Section IV-B, and thread execution in PPGs is demonstrated in Section IV-C. In the remainder of this section, we elaborate on the components that make up a PPG.

Thread information

- **Readonly Shared Data Object:** A PPG *Data Object* is an abstract data type that specifies the start address of a data block and the byte length of the data block. Each thread in a PPG can have a separate data object, or subsets of multiple threads can share common data objects. When implementing PPG threads that share data objects, it may be desirable (for enhanced performance) for each thread to maintain a local copy of the object. Such optimization is supported in our proposed PPG-based design flow.
- **Input Data Object:** Each thread in a PPG can have one or more input data objects. Threads do not share input data objects. Input data objects are used to access any data arriving from input ports of the enclosing dataflow actor, as well as any data arriving from other PPGs associated with the same actor.
- **Output Data Object:** (analogous to an input data object) Each thread in a PPG can have one or more output data objects. Threads do not share output data objects. Output data objects are used to access any data that is sent to output ports of the enclosing dataflow actor, as well as any data that is sent to other PPGs associated with the same actor.
- **Thread Function/Task:** Each thread in a PPG has an associated reference to a computational task (e.g., a function pointer in C-based actor implementation), which provides the program code associated with the thread.

PPG Execution Context

- **Group Input Manager:** Each PPG has a group input manager. If group B reads data from the output FIFO of group A, group A is called a *predecessor group* of group B, while group B is called a *successor group* of group A. A group input manager handles reading of data from any actor input FIFOs, as well as any output FIFOs from predecessor groups that are referenced during group execution. The group input manager performs data transfers to ensure that such "group input data" is transferred into local buffers associated with the group before such data is operated on.
- **Group Output Manager:** Similarly, each PPG has a group output manager, which handles the writing of processed results from the group's local buffer to actor output FIFOs, and input FIFOs of successor groups.
- **Group Member:** Each thread in a PPG has an associated group member, which is the identifier (ID) for the processor that is to execute the thread. The group member for a thread can in general be set and changed dynamically.
- **Group Owner:** Each PPG has a group owner (also a processor ID), which is invoked when the PPG is to be executed. Upon invocation, a group owner broadcasts the PPG to all of its associated group members, and then waits for the completion of the PPG. The group owner for a PPG can in general be set and changed dynamically.

B. Application Programming Interfaces

In our proposed design methodology, the PPG is the basic unit of functionality in an actor. Our development of the the PPG model includes interface APIs for various classes of key operations that are important for working with PPGs:

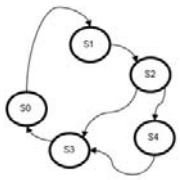
- Standardized interaction between PPGs in the actor and FIFOs of the actor;
- Standardized interaction between PPG threads and threads associated with data movement;
- Construction and configuration of new PPGs;
- Scheduling PPG threads onto processors.

These APIs are "abstract" in the sense that they are developed independently of any specific hardware platform, and can be retargeted across a variety of relevant platforms. Our design of these APIs helps to free the actor designer from tedious platform-specific details, and provides useful utilities for quick adoption of PPGs into his or her actor designs. The APIs also provide a consistent interface with which the TDIF-PPG design flow can be ported, fine tuned, and maintained on different platforms.

C. PPG Static Execution Flow

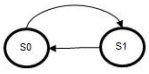
We describe a *static execution flow* for PPGs in this section. In such an execution flow, the thread information and execution context for a PPG are specified when the PPG is created. As demonstrated in the group owner finite state machine (FSM), illustrated in Fig. 2, the group owner reads the PPG first and then reads the set of group members from the PPG. After that, it calls the `broadcast` API to notify the group members about the PPG. If the group owner is also a group member (for the same group), the group owner calls the `execute` API and the `barrier` API, and then waits for all of the PPG threads to complete; otherwise, it simply remains idle and waits for the other threads to complete.

Concurrently, when a group member receives the associated PPG handler, as depicted in Fig. 3, it calls the `execute` function and then uses the `barrier` function to synchronize with the other group members and with the group owner. If inter-thread communication is necessary, it is specified by the `P2P` function. The `barrier` function can be implemented by incrementing or decrementing the value of a shared variable called a *synchronizer variable*, and checking the value of this synchronizer variable upon completion of relevant threads. If the value of this variable is equal to an appropriate predefined value, then the group member interrupts the group owner to report that all of the group members have completed their tasks under the current PPG invocation. When the group owner receives the interrupt, it reorganizes the results (data reformation, data relocation, and data merging) for any successor groups as needed.



State	Description	Transition state: condition
S0	Read the PPG	S1: PPG is read
S1	Read Group Member from PPG	S2: Group Member is read
S2	**Broadcast** the PPG to the group member	S3: The group owner is not the group member S4: The group owner is the group member
S3	Waite for the signal of completion of all threads	S0: The signal is received
S4	Invokes **Execute** and then **Barrier**	S3: Finishes **Barrier**

Fig. 2. Group owner FSM and state description table for static execution flow.



State	Description	Transition state: condition
S0	IDLE	S1: PPG is received
S1	Invokes **Execute** and **Barrier**	S0: Finishes **Barrier**

Fig. 3. Group member FSM and state description table for static execution flow.

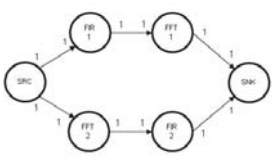
D. Examples

In this section, we demonstrate the utility of our PPG model in expressing actor level parallelism with two important examples of signal processing actors. As with graph level parallelism, data parallelism (DP), control parallelism (CP), and temporal parallelism (TP) can all be relevant to actor level parallelism. The difference in our proposed design methodology is that at the actor level, all parallelism is described in terms of relationships among threads. Since CP and DP are similar, we focus only on DP and TP in the examples of this section. In particular, DP is utilized in designing a Finite Impulse Response (FIR) filter, and a combination of DP and TP is expressed in the design of a fast Fourier transform (FFT) actor.

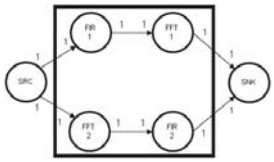
1) FIR Filter: The operation of the FIR filter is described by the following equation, which defines the output sequence $y[n]$ in terms of its input sequence $x[n]$:

$$\begin{aligned} y[n] &= b_0 x[n] + b_1 x[n-1] + \ldots + b_M x[n-M] \\ &= \sum_{i=0}^{M} b_i x[n-i] \end{aligned} \quad (1)$$

In our example of PPG-based FIR actor design, the input signal is first buffered so that blocks of samples (dataflow tokens) can be processed together. We assume that N input samples are buffered and then sent to an Mth order FIR filter to produce $(N-M)$ output samples. This example provides a significant amount of DP. For demonstration purposes, we map the DP onto two processor cores.



(a) CFDF graph for mp-sched



(b) Core computations in mp-sched

Fig. 4. The mp-sched benchmark for SDR.

However, the retargetable TDIF-PPG design flow can easily adapt the implementation of this actor to utilize more cores if available.

In order to exploit DP using a PPG, the N input samples are divided into two *input data objects* for two threads. One of these objects is derived from the samples $[x[0], x[1], \ldots, x[N/2+M-1]]$, and the other object is derived from the samples $[x[N/2], x[N/2+1], \ldots, x[N-1]]$. Similarly, the output samples are divided into two *output data objects*. The coefficient vector B is shared by two threads and placed into a *readonly shared data object*. Each thread executes the FIR calculation independently. This calculation is encapsulated in a function called `fir`, and the associated function address is set as the PPG function for each thread. This provides a PPG-based implementation of the FIR filter with two threads. Performance results from this implementation are examined in Section V.

2) FFT: The radix-2 decimation-in-time (DIT) FFT is the simplest and most common form of the Cooley-Tukey algorithm [3]. The radix-2 DIT FFT computes the DFTs of the even-indexed inputs x_{2m} $(x_0, x_2, \ldots, x_{N-2})$, and of the odd-indexed inputs x_{2m+1} $(x_1, x_3, \ldots, x_{N-1})$, and then combines the two results to produce the DFT of the whole sequence. Then the same procedure is performed recursively to reduce the overall runtime to $O(N \log N)$.

Both DP and TP from the recursive tree are utilized in the derived PPGs. With DP, each thread within $G0$ calculates half of the overall $DFT(N)$ (N point DFT), which is a $DFT(N/2)$. This calculation is performed using the Cooley-Turkey algorithm. Then a single thread in $G1$ merges the two results together to form a $DFT(N)$.

Two different implementations are provided here. In the first, $G0$ is assigned to two cores, and $G1$ executes on one of these two cores after $G0$ completes. In the second implementation, which exploits both DP and TP, $G0$ is assigned to two cores, and $G1$ is assigned to another (third) core. The mapping is coordinated in such a way that $G0$ and $G1$ execute in a software-pipelined fashion to exploit TP. Experimental results from these two implementations are discussed in Section V.

V. Experiments

In this section, we present experiments using TDIF-PPG. Our experiments involve the two actor design examples presented in Section IV-D (FIR filtering and FFT computation), and a synthetic benchmark, called *mp-sched* (which stands for "multiprocessor scheduling"). The mp-sched benchmark is composed from FIR filtering and FFT computations, and is representative of a class of subsystems of software defined radio (SDR) applications [9]. In our experiments, the mp-sched benchmark is modeled using CFDF semantics. The core functionality for each actor in the CFDF representation is encapsulated in a specific CFDF mode, called the *process* mode. In the process modes of the actors, generic PPGs are used to express actor level parallelism.

In our experiments, we employ a Texas Instruments (TI) 6678L multicore programmable digital signal processor (PDSP) as the target platform. The TI 6678L has 8 PDSP cores, where each core runs at 1.25 Ghz, and has 32KB L1 data cache, 32 KB L1 instruction cache, and 512KB L2 cache. The TI 6678L also has 4MB of shared SDRAM and 512MB of DDR3 DRAM. Using TI's Code Composer Studio IDE, we used the TI SYS/BIOS real-time operating system API and the TI Inter-Process Communication library API to implement the platform-dependent PPG APIs for these experiments. These APIs along with the generic, PPG-based actor implementations are fed into the TDIF-PPG Code Synthesis Engine to generate a complete parallel actor implementation.

TABLE I
Execution time comparison for sequential FIR filter and parallel FIR filter implementation on different input sizes.

Input Size	1079	10079	100079	1000079
Seq-FIR (s)	0.0036	0.0336	0.334	3.34
Par-FIR (s)	0.0017	0.015	0.147	1.47
Speedup	2.11	2.24	2.27	2.27

To demonstrate the performance gain from PPG-based implementation, the actual execution time of a sequential $79^{th} - order$ FIR filter (Seq-FIR) implementation is compared to that of our PPG-based parallel FIR filter (Par-FIR) implementation using the same inputs on the targeted TI platform. The results for multiple input sizes are demonstrated in Table I. The input size is in terms of the number of signal samples. The execution time is the processing time. This reported execution time excludes the time required for reading from the input FIFO and writing to the output FIFO. The FIFO reading and writing operations involve only pointer manipulations and no actual data movement, and thus have negligible impact on actor performance. The speedup is defined by the ratio between the execution time of Par-FIR and that of Seq-FIR.

The superlinear speedup is due to the VLIW feature of the employed PDSP cores. As the amount of data to process increases, so does the amount of available instruction level parallelism (ILP).

For the FFT, the execution time of the sequential FFT (Seq-FFT) implementation is compared to that of two different parallel FFT implementations. One of these parallel implementations (Par-FFT2) utilizes only data parallelism using 2 cores, while the other parallel implementation (Par-FFT3) utilizes both data parallelism and temporal parallelism using 3 cores. The execution time of Par-FFT3 is the time required to execute its longest pipeline stage. The speedup is defined as the ratio between the execution time of the parallel FFT implementation (Par-FFT2 or Par-FFT3) and that of Seq-FFT. The latency is another important figure of merit. Here, the latency is defined by the elapsed time between when the FFT actor reads the first token from its input FIFO and when it writes the first result token to its output FIFO. The results are shown in Table II. For Seq-FFT and Par-FFT2, the latency is equal to the execution time, so the latency is not shown separately.

From the results, we see that Par-FFT3 achieves more speedup than Par-FFT2 by introducing more latency.

Fig. 4(a) shows the CFDF graph for the mp-sched benchmark. In this graph, there are two paths from actor SRC to actor SNK, which represent two different signal processing procedures on the incoming signal. In the upper path from SRC to SNK, the signal is first filtered in the time-domain and then transformed to the frequency-domain. In the lower path, the signal is first transformed to the frequency-domain and then filtered in the frequency-domain. There are both graph level parallelism and actor level parallelism in this application. We derive and experiment with four different schedules to demonstrate the performance gain and trade-offs associated with the two different

TABLE II
EXECUTION TIME AND LATENCY COMPARISON AMONG SEQUENTIAL FFT, PARALLEL FFT USING 2 CORES AND PARALLEL FFT USING 3 CORES. THE RESULTS ARE COMPARED FOR DIFFERENT INPUT SIZES.

Input Size	64	256	1024	4096
Seq-FFT (s)	0.00028	0.0016	0.0086	0.045
Par-FFT2 (s)	0.00021	0.001	0.005	0.255
Speedup	1.3	1.61	1.72	1.788
Par-FFT3 (s)	0.00023	0.00073	0.0039	0.021
Speedup	1.21	2.19	2.20	2.14
Latency (s)	0.00038	0.00118	0.00517	0.0257

forms of available parallelism.

The eight PDSP cores employed in these experiments are labeled as DSP0, DSP1, ..., DSP7. In the first schedule, all four actors are assigned to DSP0 and executed sequentially by the actor sequence FIR1, FFT2, FFT1, FIR2. Note that the two paths in the graph are independent so that control parallelism can be used. Additionally, actors in each path can be pipelined to make use of temporal parallelism in the graph. In the second schedule, we again use sequential implementations for the individual actors. The actors are manually scheduled onto 4 PDSP cores to take advantage of graph level parallelism. FIR1 is assigned to DSP0; FFT1 is assigned to DSP1; FFT2 is assigned to DSP2; and FIR2 is assigned to DSP3. FIR1 and FFT2 are fired simultaneously as pipeline stage 1, and FFT1 and FIR2 are fired simultaneously after execution of pipeline stage 1 completes. We could also schedule multiple graph iterations together to explore more temporal parallelism at the graph level. However, this approach is not chosen in our experiments because it increases the required buffer sizes.

By replacing the actor implementations with parallel versions, we derive two additional schedules, which provide the third and fourth schedules for our experiments. These schedules use two different combinations of actor level parallelism and graph level parallelism based on the two different parallel FFT actor implementations discussed in Section IV-D2. In the third schedule, FIR1 is assigned to DSP0 and DSP1 using a PPG. In similar ways, FFT1 is assigned to DSP2 and DSP3; FFT2 is assigned to DSP4 and DSP5; and FIR2 is assigned to DSP6 and DSP7. FIR1 and FFT2 are fired simultaneously as pipeline stage 1, and FFT1 and FIR2 are fired simultaneously after pipeline stage 1 completes execution.

In the fourth schedule, FIR1 is assigned to DSP0 using a PPG, and similarly, FFT1 is assigned to DSP1, DSP2, and DSP3; FFT2 is assigned to DSP4, DSP5, and DSP6; and FIR2 is assigned to DSP7. FIR1 and PPG G0 of FFT2 (see Section IV-D2) are fired simultaneously as pipeline stage 1, PPG G1 of FFT2 and PPG G0 of FFT1 are fired simultaneously as pipeline stage 2, and PPG G1 of FFT1 and FIR2 are fired simultaneously as pipeline stage 3. The different schedules with the associated actor implementations are fed into the TDIF-PPG Software Synthesis Engine to generate the corresponding complete software implementations for targeted TI platform.

Using an input size of 1024, we experiment with the four different mp-sched implementations described above. The execution time, speedup and latency values for these implementations are compared on the core computation shown in Fig. 4(b). The remaining actors (SRC and SNK) take only approximately 1.2% of the computation time for sequential execution and thus do not have a significant impact on overall performance. The execution time is taken to be the processing time in the core computation region defined above. If pipelining is used, then the execution time is the time for the longest pipeline stage. The latency is defined as the elapsed time during the first iteration of graph execution between when the first input token enters the region and the time when the first output token leaves the region. The results are shown in Table III.

TABLE III
EXECUTION TIME AND LATENCY COMPARISON AMONG THE 4 SCHEDULES.

Schedule	Execution Time (s)	Speedup	Latency (s)
1	0.0243	1	0.0243
2	0.00878	2.77	0.0122
3	0.00517	4.7	0.0089
4	0.0041	5.9	0.0092

From the results, we see that exploiting graph level parallelism by itself reduces the execution time by scheduling sequential actors on multiple PDSP cores, and combining graph level parallelism and actor level parallelism further reduces the execution time. For different design constraints, different combinations can be employed. For example, schedule 3 has higher execution time but lower latency compared to schedule 4. If the system has a tight latency constraint, then schedule 3 is preferable. Our approach provides the designer with more optimization opportunities from both the actor level and graph level to help satisfy the given system design requirements.

VI. CONCLUSIONS

In this paper, we have introduced a new dataflow based design flow, called TDIF-PPG, for integrating graph level parallelism and actor level parallelism in MPSoC software optimization for DSP applications. Our approach is based on a new model, called the parallel processing group (PPG), for actor design, and an associated new plug-in to the targeted dataflow interchange format (TDIF) environment. This plug-in allows designers to express parallelism within actor designs, and integrate such intra-actor parallelism with the graph level parallelism that is already exposed in TDIF.

TDIF-PPG provides useful new features in the TDIF environment, including retargetable parallel actor design, parallel actor scheduling, and early performance evaluation. To demonstrate the utility of the PPG model and the TDIF-PPG design flow, we have presented case studies involving FIR filter and FFT actor design and mp-sched application implementation on a practical multicore programmable digital signal processor platform. We have also examined useful trade-offs in the integration of graph level parallelism and actor level parallelism.

Additionally, we have motivated several directions for future work to help strengthen the utility of PPG-based actor design and integration. These include exploration of algorithms for automated scheduling of dataflow graphs that employ PPG-based parallel actor implementations; accurate and efficient functional simulation of PPG-based designs for early-stage DSP system validation; and experimentation on other kinds of state-of-the-art digital signal processing platforms.

VII. ACKNOWLEDGMENTS

This research was sponsored in part by the Laboratory for Telecommunication Sciences, and Texas Instruments.

REFERENCES

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

[2] J. Ceng et al. MAPS: An integrated framework for MPSoC application parallelization. In *Proceedings of the Design Automation Conference*, pages 754–759, 2008.

[3] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1998.

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

[5] A. A. Jerraya, A. Bouchhima, and F. Petrot. Programming models and HW-SW interfaces abstraction for multi-processor SoC. In *Proceedings of the Design Automation Conference*, pages 280–285, 2006.

[6] S. Kwon, Y. Kim, W. Jeun, S. Ha, and Y. Paek. A retargetable parallel-programming framework for MPSoC. *ACM Transactions on Design Automation of Electronic Systems*, 13(3), July 2008.

[7] J. Mignolet and R. Wuyts. Embedded multiprocessor systems-on-chip programming. *IEEE Software*, 26(3):34–41, 2009.

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

[9] W. Plishker, G. Zaki, S. S. Bhattacharyya, C. Clancy, and J. Kuykendall. Applying graphics processor acceleration in a software defined radio prototyping environment. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 67–73, Karlsruhe, Germany, May 2011.

[10] C. Shen, S. Wu, N. Sane, H. Wu, W. Plishker, and S. S. Bhattacharyya. Design and synthesis for multimedia systems using the targeted dataflow interchange format. *IEEE Transactions on Multimedia*, 14(3):630–640, June 2012.

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

[12] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*, 2007.