INVITED PAPER

The Signal Passing Interface and Its Application to Embedded Implementation of Smart Camera Applications

A flexible, optimized communications interface suitable for smart camera systems has been demonstrated in systems for matching facial images and for efficient compression of speech.

By Sankalita Saha, Sebastian Puthenpurayil, Jason Schlessman, Shuvra S. Bhattacharyya, and Wayne Wolf, *Fellow IEEE*

ABSTRACT | Embedded smart camera systems comprise computation- and resource-hungry applications implemented on small, complex but resource-hardy platforms. Efficient implementation of such applications can benefit significantly from parallelization. However, communication between different processing units is a nontrivial task. In addition, new and emerging distributed smart cameras require efficient methods of communication for optimized distributed implementations. In this paper, a novel communication interface, called the signal passing interface (SPI), is presented that attempts to overcome this challenge by integrating relevant properties of two different, yet important, paradigms in this context—dataflow and message passing interface (MPI). Dataflow is a widely used modeling paradigm for signal processing applications, while MPI is an established communication interface in the general-purpose processor community. SPI is targeted toward computation-intensive signal processing applications, and due to its careful specialization, more performance-efficient for embedded implementation in this domain. SPI is also much easier and more intuitive to use. In this paper, successful application of this communication interface to two smart camera applications has been presented in detail to validate a new methodology for efficient distributed implementation for this domain.

KEYWORDS | Dataflow; face detection; multiprocessor communication; smart camera; speech compression

I. INTRODUCTION

The complexity of embedded smart camera systems is increasing rapidly, intensifying the need for multiprocessor implementation to meet execution time constraints. A class of important system-on-chip (SoC) architectures is emerging that provides advanced support for such embedded applications. These architectures are composed of heterogeneous subsystems, including CPUs, embedded memory, memory interfaces, and specialized I/O interfaces, along with application-specific intellectual property (IP) cores. However, the growing complexity of both the applications and the SoC platforms has made the job of design and implementation of such systems extremely difficult. One of the key challenges in this regard is the issue of communication between the different heterogeneous processing units.

In the domain of general-purpose processing, the most widely known endeavor in this regard is the message

Manuscript received December 1, 2007; revised April 1, 2008. First published October 17, 2008; current version published October 31, 2008. This work was support in part by the National Science Foundation under Grant 0325119.

S. Saha is with the Research Institute for Advanced Computer Studies, NASA Ames Research Center, Moffet Field, Mountain View, CA 94040 USA (e-mail: ssaha@riacs.edu).

S. Puthenpurayil and **S. S. Bhattacharyya** are with the Department of Electrical and Computer Engineering and the Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 USA (e-mail: purayil@umd.edu; ssb@umd.edu).

J. Schlessman is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: jschless@princeton.edu).

W. Wolf is with the Electrical and Computer Engineering Department, Georgia Institute of Technology, Atlanta GA 30332 USA (e-mail: wolf@ece.gatech.edu).

Digital Object Identifier: 10.1109/JPROC.2008.928744

passing interface (MPI) protocol [1]. The main advantage of MPI is that it is portable—MPI has been implemented for a wide range of architectures ([2], [5], [12]), and each implementation is typically optimized for the hardware on which it runs. Although MPI provides various features and various forms of flexibility, it has significant drawbacks especially in the context of embedded processing systems. Most importantly, since it is designed for general-purpose multiprocessor applications, MPI cannot leverage optimizations obtained by exploiting characteristics specific to the embedded application domain. Also, MPI is mainly intended for computer clusters, and does not scale well for SoC platforms.

The research work presented in this paper is focused at multiple levels, including the development of a suitable protocol and communication interface, and the derivation of an efficient implementation of this interface for generic signal processing applications. As mentioned earlier, smart camera applications—an important class of applications in the domain of signal processing—are increasingly being characterized by their computational intensity and distributed signal processing properties, and their real-time requirements have started creating a strong demand for sophisticated multiprocessor implementation. The new communication interface presented in this paper, called the *signal passing interface* (SPI) attempts to address this demand by integrating relevant properties of two different yet important paradigms in this context—dataflow and MPI. SPI is targeted towards signal processing applications, and through careful specialization, it is streamlined for embedded implementation in this domain. It attempts to overcome the overheads incurred by the use of MPI for signal processing applications by carefully integrating concepts of MPI with coarse-grain dataflow modeling and useful properties of interprocessor communication (IPC) that arise in this dataflow context. The resulting integration provides a new, more intuitive and easy-to-use paradigm for multiprocessor implementation of signal processing applications. A preliminary version of SPI was introduced earlier in [26].

Of the various signal-processing-oriented dataflow models of computation, *synchronous dataflow* (SDF) is a particularly popular one because of its static nature, which leads to compile-time predictability and potential for extensive analysis and optimization. Hence, the underlying dataflow model in SPI is SDF. However, SDF does not allow dynamic rates of data transaction between different subsystems in a dataflow graph, thereby limiting its applicability. A significant number of smart camera applications do not conform to the rigid dataflow structure imposed by SDF semantics. For example, consider a multiple-camera object tracking system. In such a setup, each camera typically communicates with other cameras at runtime to exchange and update data. The values of this data and, more importantly, the overall volume of the data cannot be determined statically since they depend on the current video input. Thus, to implement an effective communication interface for the smart camera domain, it is imperative to allow some amount of dynamic behavior and hence to look beyond conventional SDF for our proposed interface.

In this paper, we present the concept of *variable token size* (VTS)—earlier supported implicitly by various forms of dynamic dataflow—as an explicit modeling tool to apply efficient and intuitive SDF techniques to certain kinds of dynamic dataflow behavior. This concept is integrated into SPI to enable efficient handling of dynamic behavior in subsystems that need such flexibility, while static-dataflow-oriented subsystems are handled using conventional, fixed-token-size SDF semantics.

A significant motivation for using SDF in SPI is to exploit the powerful optimization techniques that are enabled by design and synthesis using SDF. One such technique is *resynchronization*. Resynchronization is used in the context of multiprocessor implementation of SDF graphs for reducing synchronization overhead between processors. Until now, this technique has been mainly used for shared-memory systems. Through the SPI framework, the concept of resynchronization is extended to distributed memory systems, and corresponding optimizations are incorporated into SPI.

An interface definition is incomplete without an optimized implementation. Hence, we have developed two different SPI library implementations. An SPI library consists of special communication-related, functional modules (communication actors) that take care of message passing and implementing the various synchronization and buffer management protocols for correct functioning of an SPI-based system. These special modules ensure that the communication part of a system is completely isolated from the computation part. As a first demonstration of SPI, we developed a software-based SPI library using MPI as the communication layer [26]. Subsequently, we developed a hardware description language (HDL) realization of the SPI library that supports VTS, and realized the functionality of this library on the Xilinx Virtex-4 field-programmable gate array (FPGA) platform. A preliminary version of this library was introduced in [27]. In this paper, a more comprehensive development is provided, along with new communication buffer optimization strategies for more efficient library implementation.

Smart camera applications not only include image processing systems but often involve voice and speech applications as well. A typical example is a conference room that employs smart cameras to capture streaming video of the speaker in the room as well as microphones to record and transmit the voice. The applications—face detection and speech compression—that we have used in our experiments for this paper reflect this multimodal characteristic. The results of these experiments show the efficiency of the SPI interface and its implementation, as well as its robustness in handling dynamic behavior.

II. RELATED WORK

With an increasing shift toward heterogeneous hardware/software platforms, the need for improved parallel software has been correspondingly increasing. To this end, the most popular programming paradigm supported by most multiprocessor machines has been MPI [13]. Due to various drawbacks in MPI, however, alternative protocols have been proposed, such as message passing using the parallel vector machine (PVM) [31], extensions to the C language in unified parallel C (UPC) [8], and in recent years, OpenMP [10] for shared memory architectures. However, all of these are software techniques target general-purpose applications, and are not tuned towards the signal processing domain.

In terms of hardware, a wide variety of architectures for optimized implementation of communication networks have been explored as well. An efficient communication mechanism for Multiple Instruction Multiple Data (MIMD) processors was proposed in [14]. An architecture that integrates communication and computation in hardware is shown in [28]. A system-oriented approach as shown in [32], where communication for the Eindhoven multiprocessor system (EMPS) is presented that utilizes a truly distributed interrupt mechanism available to operating system primitives for message passing and remote procedure calls. A review of various practical hardware implementations of communication mechanisms is provided by Henry in [18]. Henry also proposes a novel mechanism that directly couples hardware with applications by bypassing intermediate operating system handling [18]. In recent years, specialized interfaces and middleware for signal processing applications have come into focus as well (e.g., see [9], [17]).

However, the above body of work does not address the need for a standardized interface that is portable over different platforms, while retaining application-domain-specific optimizations, particularly in the field of signal processing. The broad range of signal processing applications includes many computation- intensive applications, such as those for processing image and video signals. Due to their real-time processing requirements, smart camera systems are among the most important candidates for multiprocessor implementation. With the advent of distributed smart camera systems ([1], [6]), the need for effective communication has increased further. In this work, this gap is addressed systematically through a novel communication interface that integrates application-specific streamlining with a system-level approach to yield an intuitive and efficient IPC system for computation-intensive smart camera applications.

III. BACKGROUND

A. Dataflow-Based Modeling of Signal Processing Systems

The format of coarse-grain dataflow graphs is one of the most natural and intuitive modeling paradigms for signal processing systems (e.g., see [21], [29]). In dataflow, a program is represented as a directed graph in which the nodes (called *actors*) correspond to computational modules, and data (encapsulated as *tokens*) is passed between actors through first-in first-out (FIFO) queues that correspond to the edges of the graph. Actors can *fire* (perform their respective computations) when sufficient data is available on their input edges. Delays on edges represent initial tokens and specify dependencies between iterations of the actors in iterative execution. When dataflow is applied to represent signal processing applications, actors can have arbitrary complexity. Typical signal processing actors are finite and infinite impulse response filters, fast Fourier transform computations, and decimators.

When dataflow is used to model signal processing applications for parallel computation, four important classes of multiprocessor implementation can be realized: fully static, self-timed, static assignment, and fully dynamic methods [22]. Among these classes, the self-timed method is often the most attractive option for embedded multiprocessors due to its ability to exploit the relatively high degree of compile-time predictability in signal processing applications, while also being robust with regards to actor execution times that are not exactly known or that may exhibit occasional variations [29]. For this reason, we target our SPI methodology to the self-timed scheduling model, although adaptations of the methodology to other scheduling models is feasible and is an interesting topic for further investigation.

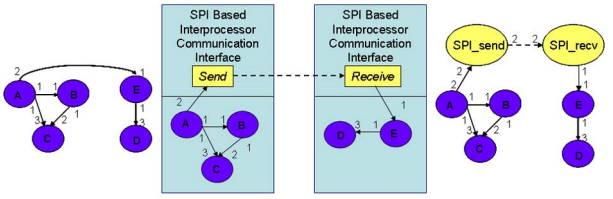


Fig. 1. ***Model for implementing the SDF graph on the left on a two-processor system and the new SDF graph.***

B. The Signal Passing Interface

The FIFO buffers associated with edges in a dataflow graph queue tokens that are passed from the output of one actor to the input of another. When an actor is executed, it consumes a certain number of tokens from each of its inputs, and produces a certain number of tokens on its outputs. The SDF model is a restricted form of dataflow in which the number of tokens consumed/produced by an actor on each input/output port is constant. For a given dataflow graph, SPI inserts a pair of special actors (called SPI actors) for sending and receiving associated IPC data whenever an edge exists between actors that are assigned to two different processors [26], as shown in Fig. 1.

The static properties of SDF together with self-timed scheduling provide important opportunities for streamlining of interprocessor communication. This leads to the following differences between SPI and MPI implementations.

- In MPI, a *send* message contains the identifier (ID) of the destination actor. In SPI, instead, the message contains the ID of the corresponding edge of the graph.
- In the static version of SPI, unlike MPI, there is no need to specify the size of the buffer associated with a message. Once a valid schedule is generated for the graph, all the edge buffer sizes are known, and hence, the edge ID can be used to determine the buffer size. Although buffer overflow conditions may occur if one of the processors produces data at a faster rate than the corresponding consumer processor consumes it, these conditions can be systematically eliminated in SPI through appropriate buffer synchronization protocols [26].
- SPI allows only nonblocking calls as opposed to provisions in MPI. Any buffer overwriting or race conditions are avoided by the self-timed execution model, which enforces that 1) any actor can begin execution only after all its input edges have sufficient data, and 2) each actor is implemented on one processor only, and not distributed across multiple processors.

IV. VIRTUAL TOKEN SIZE (VTS) MODEL

As mentioned in Section III, a significant limitation of SDF is that it does not allow dynamic variation in token size. In this section, we propose the concept of VTS to deal with dynamically varying data rates encountered in various signal processing applications. In *dynamic* dataflow, the production/consumption rates of an actor may change at run time depending on current or previous values of its input data. Dynamic dataflow originates at dynamic ports. An actor is called dynamic if at least one of its ports is dynamic. General dynamic dataflow, with no *a priori* information about the dynamic behavior, has high overhead and low predictability for synthesis of efficient real-time implementations, since, for example, this requires fully dynamic memory management.

Our approach to accommodating a significant degree of dynamic dataflow is through the use of variable token sizes, while maintaining static production/consumption rates of actors in terms of the numbers of tokens that are produced or consumed. Furthermore, to enable static memory allocation, we require that an upper bound on the token size is specified for each dynamic port. VTS provides a mechanism to "repack" tokens in such a way that the new ("packed") tokens flow at static rates, in situations where the underlying raw ("unpacked") tokens were flowing at dynamic rates.

Consider actors A and B in Fig. 2. The production rate of edge (A, B) varies dynamically but has an upper bound of 8.

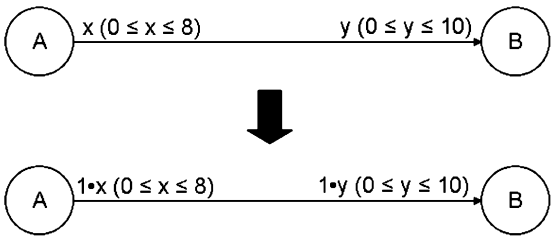


Fig. 2. *SDF graph with dynamic data rates and corresponding VTS conversion.*

Similarly, the consumption rate varies with an upper limit of 10. These varying rates can be captured using variable token sizes. Thus, A has a production rate of x with a token size of 1 and B has a consumption rate of y with token size of 1, where x has an upper bound of 8 and y has an upper bound of 10. If by application of the above principle to all possible edges, a consistent graph is obtained, then bounded memory for all the edge buffers can be guaranteed. We call such a conversion *VTS conversion* of the original dataflow graph. The edge buffer memory for this simple edge is 10; with this consideration, all SDF-based optimizations can be applied on the new converted graph. A more detailed method for computing the bound on edge buffer memory is derived later in this section.

However, it may be noted, that a VTS conversion may not be always possible, like in the case shown in Fig. 3. In this graph, consider the situation when actor A produces nonzero number of tokens on edge e_{AB} but no token (zero tokens) on edge e_{AC}; actor B will be able to fire but actor C will not be able to fire thereby preventing actor from D firing. This would cause unbounded accumulation of tokens on edge e_{BD} and hence unbounded memory requirements.

An upper bound on the total size of the *packed* tokens on an edge e is required to ensure bounded buffer memory using VTS. This may be computed as follows. Let $c_{sdf(e)}$ be an SDF buffer bound of e, i.e., an upper bound on the

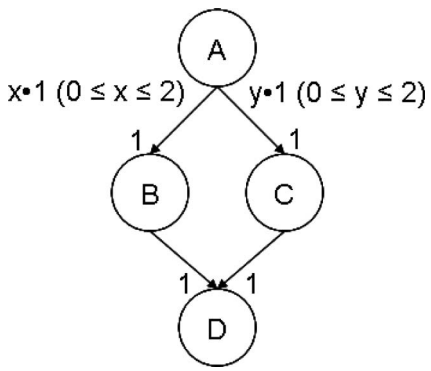


Fig. 3. *Example showing how VTS may cause unbounded buffer memory requirements.*

buffer size of e in terms of the maximum number tokens that coexist on e at any given time. $c_{sdf(e)}$ can be computed using any of the existing techniques for computing SDF buffer bounds (e.g., see [29], [30]) and is computed on the graph after VTS conversion, so it is computed on a pure SDF graph. The total size of *packed* tokens is then given as

$$c(e) = c_{sdf}(e) \cdot b_{\max}(e) \tag{1}$$

where $b_{\max}(e)$ is the maximum number of bytes in a *packed* token associated with e in the VTS conversion. In our application of VTS, we require that the bound $b_{\max}(e)$ exists and is known in advance. When the bound exists, it can be determined from any available bound on the maximum variable data rate for a port (e.g., the bound on x in Fig. 2) times the maximum number of bytes in a single raw (unpacked) token for the port.

The IPC buffer bounds, as computed in [29], now gets modified to

$$B(e) = (\rho(src(e), snk(e)) + delay(e)) \times c(e) \tag{2}$$

where $B(e)$ is the upper bound on the IPC buffer size, ρ_G is the total delay on a minimum-delay path directed from $src(e)$ to $snk(e)$, $delay(e)$ is the initial delay on edge e, and $c(e)$ is as defined (1).

In terms of implementation of VTS in the context of SPI, there have to be provisions for notifying the receiving actor of the token size of the current tokens being sent. This can be done either by transmitting a token size header or by using a special delimiter that is then used by the receiver to determine the length of the message. The most efficient method to use is dependent on the implementation platform. For example, if the final target is an FPGA (as in our case), using a delimiter can be expensive as it would then involve extra operations on the receiver side to determine the length of the message. Thus, sending the size using a field in the header of the message is much more efficient.

Note that VTS conversion of a graph requires that actors operate in terms of packed tokens, and this mode of operation can in general result in actors that have unbounded storage requirements for their internal state. Thus, to ensure overall bounded memory for an application, it must be ensured that the actors themselves operate within bounded memory (e.g., by disallowing dynamic memory allocation within actor implementations). Our preliminary experiments, however, indicate that in practice, VTS conversion can be performed for useful applications in conjunction with bounded memory actor implementation, and therefore the technique can be an important technique to consider when encountering dynamic dataflow behavior.

A. Related Work on VTS

Many extensions to the SDF model have been proposed to broaden the range of applications that can be represented while maintaining the compile-time predictability properties of SDF as much as possible. Well-behaved stream flowgraphs (WBSFGs), proposed by Gao *et al.* [15], allow the use of non-SDF dynamic actors for modeling conditionals and data-dependent iteration, but in a restricted fashion such that the model retains the key predictability properties of SDF. In Buck's Boolean dataflow (BDF) [7], the number of tokens produced or consumed by an actor on an edge is either fixed, or is a two-valued function of a control token present on a control terminal of the same actor. Integration of reactive behavior with dataflow (reactive process networks) to capture the interaction between reactive and streaming components in multimedia applications has been introduced by Geilen and Basten [16].

The VTS approach developed in this paper differs from the above models by providing an explicit way to allow dynamic token transfer rates within the SDF framework. Bounded dynamic dataflow (BDDF), introduced by Pankert *et al.* [24], allows arbitrary data rates as long as an upper bound is specified for the data rate of each dynamic port. Our application of VTS is similar to BDDF in the sense that it requires bounds on dynamic behavior. However, in our approach, dynamic behavior is captured by varying token sizes instead of varying data rates—i.e., repacking of tokens at dynamic-rate ports to enforce SDF behavior. Thus, in contrast with BDDF, we can apply SDF-based analysis techniques to our VTS-based system representations.

Kahn process networks (KPNs) [19] provide another modeling paradigm that is popular for signal processing. The KPN model of computation is closely related to dataflow, and in the context of signal processing, the terms are sometimes used interchangeably. Since the current version of SPI exploits static predictability properties of SDF and associated inter-processor communication strategies, and these properties are not present in the general KPN model, SPI cannot be used in conjunction with arbitrary KPN representations. However, integration of SPI with KPN—especially, restricted versions of KPN that are more amenable to formal analysis (e.g., see [11])—is a promising direction for future work.

V. SYNCHRONIZATION GRAPH MODELING

The SPI methodology uses the synchronization graph model, which is a graph-theoretic model for analyzing the performance and synchronization structure of a self-timed multiprocessor implementation [3]. In this section, we develop a precise definition of the synchronization graph for SPI in the context of distributed memory systems, and the associated resynchronization technique to reduce synchronization overhead.

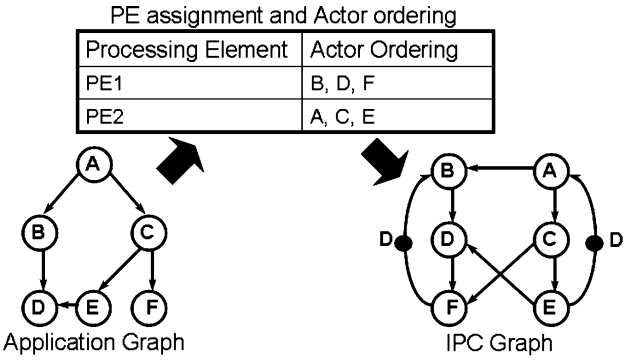


Fig. 4. ***Example of derivation of an IPC graph from an application graph.***

A. Synchronization and SPI

The *SPI_BBS* and *SPI_UBS* protocols derived earlier [26] in the context of shared memory systems provide buffer synchronizations between the sender and receiver side. Here BBS and UBS stand for bounded buffer and unbounded buffer synchronization, respectively. These protocols were modified slightly for better optimization in the context of SPI for distributed memory systems to lead to *SPI_BBS* and *SPI_UBS* protocols [26]. If it can be guaranteed that a buffer will not exceed a predetermined size, then the *SPI_BBS* protocol is used. The *SPI_UBS* protocol is used when it cannot be guaranteed statically that an IPC buffer will not overflow through any admissible sequence of send/receive operations on the buffer. Note that the protocols remain invariant in case any of the interprocessor edges uses VTS, since that is already encapsulated in the header of the message.

B. Synchronization Graph

Given the dataflow graph for an application G and a multiprocessor schedule for G, we derive a data structure called the IPC graph G_{ipc} by instantiating a vertex for each task, connecting an edge from each task to the task that succeeds it on the same processor, and adding an edge that has unit delay from the last task on each processor to the first task on the same processor. Also, for each edge (x, y) in G that connects tasks that execute on different processors, an IPC edge is instantiated in G_{ipc} from x to y. Fig. 4 shows an application graph and how the corresponding IPC graph is derived using processor assignment/actor ordering. Each edge (v_j, v_i) in G_{ipc} represents the constraint

$$start(v_j, v_k) \geq end\big(v_j, k - delay(v_j, v_i)\big) \qquad (3)$$

where $start(v_j, v_k)$ and $end(v_j, k)$ represent the time at which invocation k of actor v_j begins and completes execution, respectively.

An IPC edge in G_{ipc} represents both data communication and synchronization functions. The synchronization graph G_S, derived from G_{ipc}, shows synchronization constraints only. Initially G_S is identical to G_{ipc}. However, resynchronization modifies the synchronization graph by adding and deleting synchronization edges. The synchronization graph construction depends on the underlying target platform (e.g., shared memory [3] or domain specific [20]). Thus, the construction of such a graph is a general part of SPI methodology.

C. Resynchronization

The process of adding one or more new synchronization edges and removing any redundant synchronization edges that result is called *resynchronization*. Resynchronization exploits the well-known observation that in a given multiprocessor implementation, certain synchronization operations may be redundant in the sense that their associated sequencing requirements are ensured by other synchronizations in the system. The goal of resynchronization is to introduce new synchronizations in such a way that the number of additional synchronizations that become redundant exceeds the number of new synchronizations that are added, and thus the net synchronization cost is reduced.

Fig. 5 (adapted from [29]) illustrates how this concept can be used to reduce the total number of synchronizations in a multiprocessor implementation. Here, the dashed edges represent synchronization edges. Observe that if we insert the new synchronization edge $d_0(C, H)$, then two of the original synchronization edges—(B, G) and (E, J)—become redundant and can be removed.

SPI-based implementation of a system that uses the *SPI_UBS* protocol can result in multiple redundant acknowledgments which increases the synchronization overhead. These overheads can be removed by careful and systematic application of resynchronization to the complete system. Thus, resynchronization for SPI-based implementation involves removal of redundant acknowledgment edges for SPI actors. In our HDL-based SPI library implementation, the corresponding actors do not implement synchronization acknowledgments, thereby implementing the *SPI_BBS* protocol. This technique is further illustrated using a practical application in Section VI-C.

VI. EXPERIMENTS

Experiments were carried out with two applications: a face detection application and a speech compression using *linear predictive coding* (LPC). In this section, we present details on our SPI library implementation and the targeted applications, as well as on the system design and implementation of the applications using SPI.

A. SPI Library Implementation

To accommodate static as well as dynamic behavior, we propose a two-phase SPI interface consisting of *SPI_static*

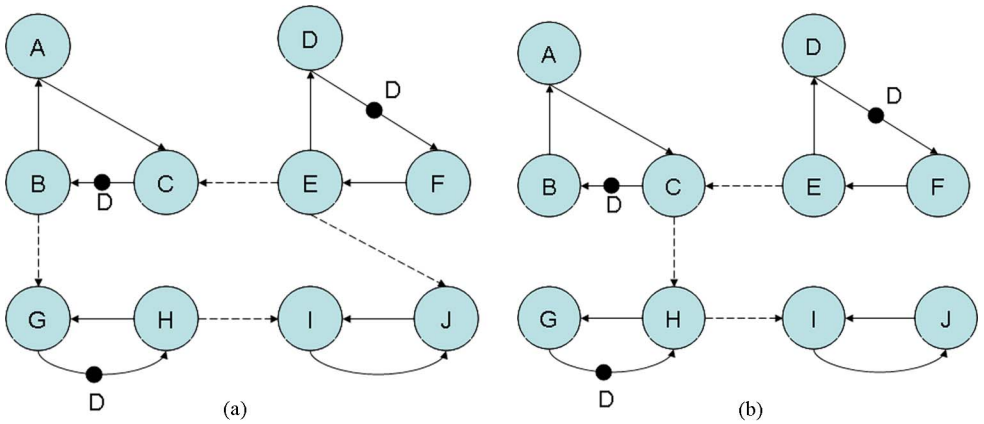


Fig. 5. ***An example of resynchronization.***

and *SPI_dynamic* components. *SPI_static* handles the static communication part, i.e., communication between subsystems whose behavior is determined before run-time and follows the format introduced in the preliminary version of SPI [26]. For edges that exhibit dynamic communication behavior, *SPI_dynamic* is used. However, there are limitations to the range of dynamic behavior that can be encompassed. The most important restriction is that the communication edges should be known before run-time and they cannot vary at run-time. Once this is fixed, varying data rates can be supported by using VTS, given an upper bound on the variation, as described in Section IV.

The first library was implemented in software and used an underlying MPI layer for communication [26]. A new FPGA library for SPI was developed later using the Xilinx System Generator. *SPI_init*, *SPI_send* and *SPI_receive* actors for both *SPI_static* and *SPI_dynamic* were implemented. The message header for *SPI_static* consists of the ID of the interprocessor edge only while that of *SPI_dynamic* also contains the message size. Note that for *SPI_dynamic*, the need for a message header can be eliminated by using an appropriate delimiter. However, for FPGA implementation, use of such header space is relatively inexpensive and more efficient. Also, in our targeted implementations, the message datatype for all communication edges is known at compile time, and hence need not be included in the message header.

1) Communication Buffer Usage Optimization: An important consideration in the library implementation is how the communication buffering is implemented. Two possible buffering strategies are shown in Figs. 6 and 7. The first method (*buffer strategy 1*) clearly provides memory savings since only one buffer is used. However, this can lead to blocking on the part of the sending actor *A* as it needs to receive an acknowledgment to release the buffer. In the second method (*buffer strategy 2*), this problem is eliminated as long as it can be ensured that the communication buffer does not overflow—a condition ensured by the protocols employed by the SPI methodology. However, this comes at the cost of increased memory as well as extra read and write operations. In the FPGA-based SPI library that has been implemented in this work, experiments with the *buffer strategy 1* was performed, since on our experimental platform the extra cost of reading and writing was expensive enough to overshadow the gains

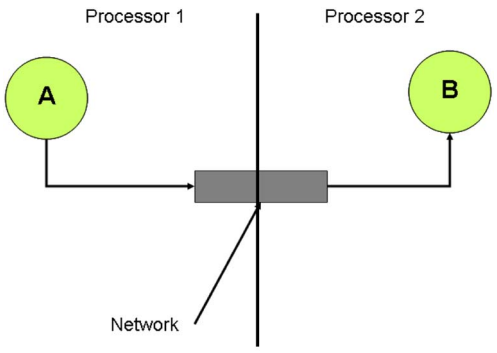


Fig. 6. ***Communication buffer usage: buffer strategy 1.***

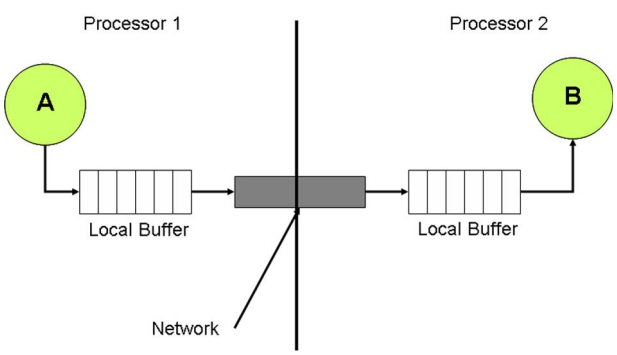


Fig. 7. ***Communication buffer usage: buffer strategy 2.***

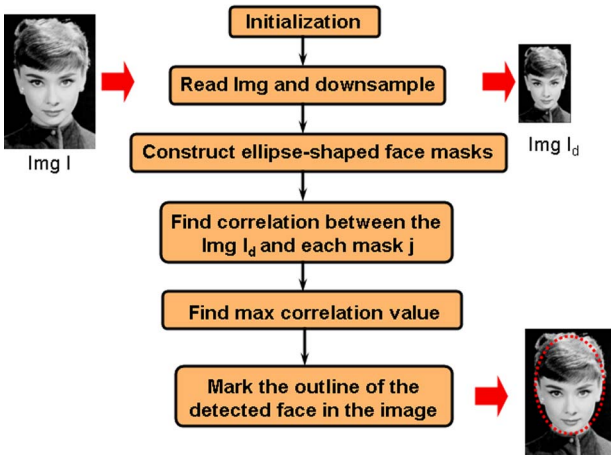


Fig. 8. *Algorithmic flow for face detection system.*

obtained from more flexible parallelization execution (i.e., do the provision for nonblocking sending operations). However, to analyze the differences in implementations, comparison with the implementation using *buffer strategy 2* was also performed. In general, both methods should be considered for best performance, since the underlying tradeoff is heavily platform-dependent.

B. Face Detection

Smart camera applications such as face detection are good candidates for parallelization as they are computation intensive, and at the same time inherently parallelizable. In this work, a shape-based approach proposed by Moon *et al.* [23] is used. Fig. 8 shows the complete flow of the employed face detection algorithm. At the implementation level, this system involves finding out correlations between a set of ellipse-shaped masks with the image in which a face is to be detected. Profiling results show that this operation (mask correlation) is computationally the most expensive operation. However, this operation is parallelizable. The mask set can be divided into subsets and each subset can be handled by a single processor independent from the rest.

A coarse-grain dataflow model of the resulting face detection system is shown in Fig. 9 for the case of three processors for mask correlation, with the corresponding actor-to-processor assignment shown in Fig. 10. Besides using multiple processors for the correlation operation, we use a separate processor to handle the required I/O operations. The actors in the coarse-grain dataflow model are explained as follows.

- Actor *RI*: Reads Image *I* and downsamples it.
- Actor BM_i: Creates the mask set for PE_i.
- Actor PE_i: Computes correlation for mask set BM_i and image *I* and finds the local best match.
- Actor *FR*: Finalizes results by finding the best match among all the local matches and marking the outline.

This system consists of static operations only and hence does not require VTS or *SPI_dynamic*. It was implemented using our software SPI library. Our focus in these experiments is on SPI only. However, to study how SPI compares with MPI, a study was carried out earlier [26] which showed that performance-wise SPI does as well or better than MPI for the face detection system. The other advantage provided by SPI is a streamlined library that makes programming much simpler and easier compared to MPI. Also, because of this DSP-based streamlining, it is expected that a completely dedicated software implementation of SPI would result in a small communication library which is extremely beneficial for embedded systems having resource scarcity, while providing a standard interface for communication with different kinds of target platforms such as FPGAs. FPGAs are targets in the experiment that we describe in the next section.

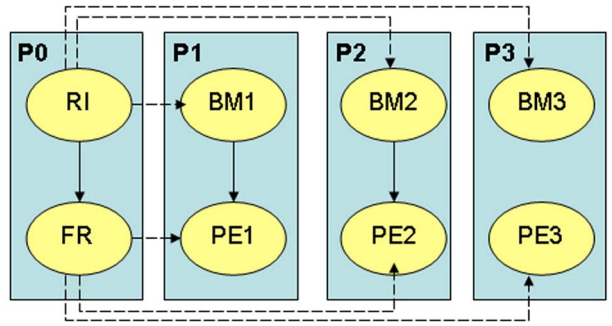


Fig. 10. *Actor-to-processor assignment for face detection system.*

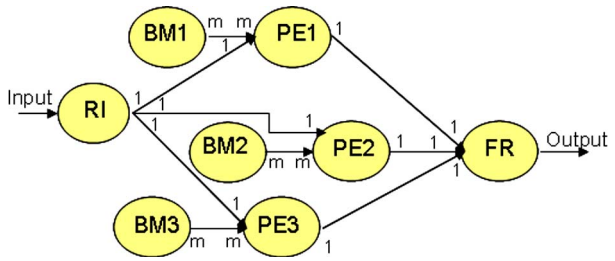


Fig. 9. *Coarse-grain dataflow model for face detection system.*

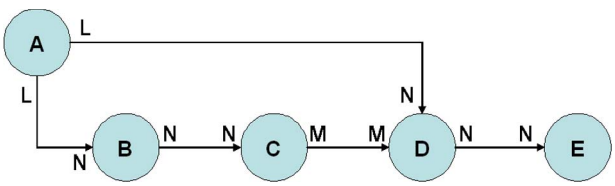


Fig. 11. *Dataflow graph for speech compression system.*

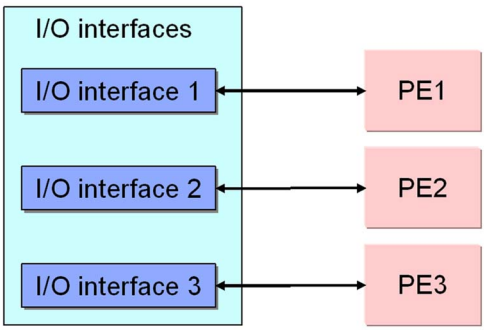


Fig. 12. *3-PE architecture for error generation actor of speech compression application.*

C. Speech Compression

Linear predictive coding (LPC) is one of the most powerful speech analysis techniques and a popular method for encoding good-quality speech at a low bit rate; which motivated us to choose it as the second application for our experiments. The basic components of LPC consist of framing, determination of predictor coefficients, and coding. For the process of framing, the input signal for a particular duration of time containing L samples is divided into frames each of size N. Each frame is subjected to the linear predictor, which generates predictor coefficients that depends on the LPC model order M. These coefficients are then used to determine the predicted value of the input signal. The prediction error and its coefficients are quantized, and this quantized data is denoted as the compressed data. To retrieve the original data in a lossless manner, a corresponding decompression algorithm can be applied in which the predictor coefficients and error data are processed. For further details, the reader is referred to [25].

The data flowgraph for this application is shown in Fig. 11. The computational blocks are represented by the actors A to E.

- Actor A: Reads a segment of input data and stores it in a buffer.
- Actor B: Implements a fast Fourier transform (FFT) operation on the input samples.
- Actor C: Performs LU decomposition for determining the predictor coefficients.
- Actor D: Generates the error on samples using autoregressive (AR) model.
- Actor E: Implements Huffman coding on the error samples.

Most of the actors have high computational intensity and the FPGA hardware resources were not enough to fit a multiprocessor version of the whole system. Thus, for this application, we explore the parallelization of only the error generation actor (D) in hardware, while the rest of the actors were implemented in software Thus, this experiment of SPI is in the context of an overall hardware/software codesign solution.

Actor E requires the input samples as well as the predictor coefficients for error computation. Since the number of coefficients that depend on the model order M and the size of the input frame are not known before runtime, this leads to dynamic data transactions and use of *SPI_dynamic*. For processing elements (PEs) computing the errors in parallel, each PE computes error $\lceil (N/n) \rceil$ values.

The system architecture for a 3-PE implementation of the system is shown in Fig. 12. Each PE_i performs error calculation for the part of the frame that is sent to it. Note that for the parallel version of error calculation, the input frame is split into overlapping sections and each PE finds out the error values corresponding to $\lceil (N/n) \rceil$ such sections. The I/O interface for a given PE sends the predictor coefficients and the input frame subsections to the PE and receives the computed error values. Fig. 13 shows the

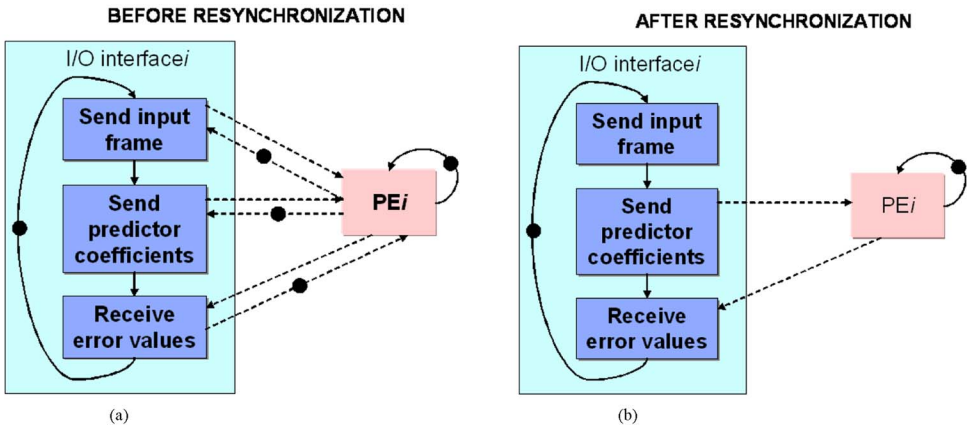


Fig. 13. *Resynchronization for 3-PE implementation of error generation actor of speech compression application.*

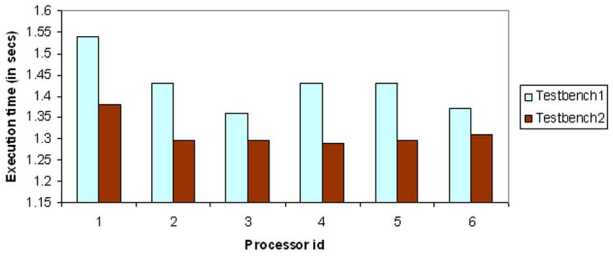


Fig. 14. *Execution time results for face detection system.*

Fig. 16. *Comparison of execution times for different buffer implementations for speech compression system. Here input sample size $L = 300$.*

synchronization graph before and after resynchronization is performed. Here, the dashed edges represent synchronization edges. Note that the synchronization graph is constructed by considering appropriate *SPI_send* and *SPI_receive* actors jointly with their corresponding dataflow actors. We do not show these SPI actors in the illustrated synchronization graphs for purposes of clarity and also since the main aim of the graph is to illustrate the synchronization operations only.

The main aim of this particular experiment was to demonstrate the flexibility of SPI with regards to target platforms, i.e., SPI can be readily used in embedded software as well as dedicated hardware with resource constraints. Although new FPGAs are quite powerful with special on-chip memory and support for DSP acceleration, they are not always adequate for the kinds of computation- and memory-intensive systems that are often encountered in smart camera systems, thereby leading to critical resource constraints for such applications [20]. For such platforms, SPI—besides being simple, intuitive, and flexible—can provide a low-overhead communication interface as shown by the results for the library size (Figs. 17 and 18).

VII. RESULTS

The software implementation for the face detection system was targeted towards multiprocessor platform consisting of a combination of IBM Netfinity and Dell Poweredge hardware nodes: each node was a dual-processor PIII-550 (2xPIII-550 MHz) with 1-GB memory and 2×18 GB of disk capacity. Each node had a 1-Gb link to the other nodes. Two test benches involving 126 and 114 masks, each of size 121×191 and 127×183 pixels, and image of size 128×192 pixels were used. The execution time results for the two test benches are shown in Fig. 14.

The hardware implementations experimented with were designed using Xilinx System Generator version 9.1 and synthesized using Xilinx ISE 9.2. The target device family was Virtex 4 with a speed grade −10. Although the FPGA board could support a clock frequency of 500 MHz, this frequency could not be attained in most cases. Fig. 15 shows the performance results that were obtained for the error calculation module of the speech compression system. Here, n represents the number of PEs used; $n = 1$ implies the serial implementation, and L is the input frame size. In Fig. 16, comparison between implementations using different buffer strategies is shown for input frame size of 300. Figs. 17 and 18 show the FPGA area requirements for a 4-PE implementation of the speech compression system using the different buffer strategies. Clearly, for this implementation platform, buffer strategy 1 is a better choice both in terms of performance and resource requirements.

VIII. CONCLUSION

In this paper, a new, flexible and optimized communication interface for signal processing applications has been

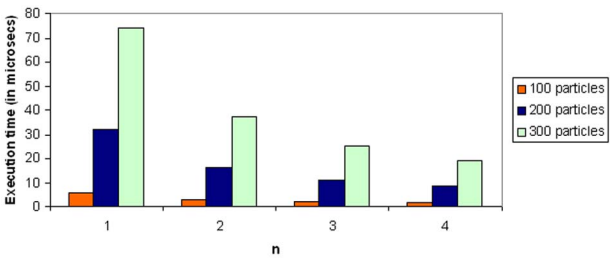


Fig. 15. *Performance results for error generation module for speech compression system using buffer strategy 2. L is the input frame size.*

	Slices	Slice FFs	4 input LUTs	Block RAMs
Full system	2.63%	1.88%	2.15%	8.33%
SPI library (relative to full system)	11.88%	12.5%	13.94%	50%

Fig. 17. *FPGA resource requirements for a 4-PE implementation of error generation module for speech compression system using buffer strategy 1.*

	Slices	Slice FFs	4 input LUTs	Block RAMs
Full System	3.36%	4.14%	5.55%	12.5%
SPI library (relative to full system)	31.01%	20.75%	33.33%	66.67%

Fig. 18. ***FPGA resource requirements for a 4-PE implementation of error generation module for speech compression system using buffer strategy 2.***

presented. SPI achieves significant streamlining and associated advantages by integrating with the MPI framework the properties of dataflow graph modeling, synchronous dataflow analysis, and self-timed scheduling.

The novel concept of applying variable token sizes for dynamic data-rate transactions between different actors in a data flow graph has been presented, analyzed, and demonstrated. Given an upper bound on the associated data rate variation, the scheme can handle dynamic data-rate behavior through an enclosing framework of efficient, SDF-based analysis. Capability to support dynamic data rates between different subsystems for SPI has been added by thorough integration of this concept into the SPI framework. Resynchronization for distributed embedded systems in the context of SPI has also been explored, and related synchronization optimizations have been added to the interface. Analysis of different buffering strategies in the SPI library implementation has also been carried out.

Two communication libraries for SPI have been created and demonstrated on two useful smart camera applications. This work demonstrates the capability of SPI to provide a standard, low-cost, and modular message-passing interface with careful streamlining for signal processing applications, and with efficient separation of communication and computation for easier development of embedded multiprocessor systems. ■

REFERENCES

[1] I. F. Akyildiz, T. Melodia, and K. R. Chowdhury, "Wireless multimedia sensor networks: Applications and testbeds," in *Proc. IEEE (Special Issue on Distributed Smart Cameras)*, vol. 96, no. 10, pp. 1588–1605, Oct. 2008.

[2] M. Banikazemi, R. K. Govindaraju, R. Blackmore, and D. K. Panda, "MPI-LAPI: An efficient implementation of MPI for IBM RS/6000 SPsystems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1081–1093, Oct. 2001.

[3] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Optimizing synchronization in multiprocessor DSP systems," *IEEE Trans. Signal Process.*, vol. 45, no. 6, pp. 1605–1618, Jun. 1997.

[4] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Resynchronization for Multiprocessor DSP Implementation Part 1: Maximum-Throughput Resynchronization," Digital Signal Processing Lab., Univ. Maryland, College Park, Jul. 1998, Tech. Rep.

[5] G. D. Burns, R. B. Daoud, and J. R. Vaigl, "Lam: An open cluster environment for MPI," in *Proc. Supercomputing Symp. '94*, Toronto, ON, Canada, 1994.

[6] A. C. Sankaranarayanan, A. Veeraraghavan, and R. Chellappa, "Object detection, tracking and recognition for multiple smart cameras," in *Proc. IEEE (Special Issue on Distributed Smart Cameras)*, vol. 96, no. 10, pp. 1606–1624, Oct. 2008.

[7] J. T. Buck, "Scheduling Dynamic Dataflow Graphs With Bounded Memory Using the Token Flow Model," Univ. California at Berkeley, Tech. Rep.UCB/ERL 93/69, Sep. 1993, Ph.D. thesis.

[8] W. W. Carlson, J. M. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," IDA/CCS, Tech. Rep. CCS-TR-99-157.

[9] A. Chirila-Rus, K. Denolf, B. Vanhoof, P. R. Schumacher, and K. A. Vissers, "Communication primitives driven hardware design and test methodology applied on complex video applications," in *Proc. IEEE Intl. Workshop Rapid System Prototyping*, 2005, pp. 246–249.

[10] L. Dagnum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, p. 4655, 1998.

[11] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, and M. Sen, "Affine nested loop programs and their binary cyclo-static dataflow counterparts," in *Proc. Int. Conf. Application Specific Systems, Architectures, and Processors*, Steamboat Springs, CO, Sep. 2006, pp. 186–190.

[12] K. Dincer, "A ubiquitous message passing interface implementation in Java:jmpi," in *Proc. 13th Int. and 10th Symp. Parallel and Distributed Processing*, Apr. 12–16, 1999, pp. 203–207.

[13] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "A message passing standard for MPP and workstations," *Commun. ACM*, p. 8490, 1996.

[14] M. Dubois and F. A. Briggs, "Efficient interprocessor communication for MIMD multiprocessor systems," in *Proc. 8th Annu. Symp. Computer Architecture*, 1981, pp. 187–196.

[15] G. R. Gao, R. Govindarajan, and P. Panangaden, "Well-behaved programs for DSP computation," in *Proc. Int. Conf. Acoustics, Speech, and Signal Processing*, San Francisco, CA, Mar. 1992.

[16] M. Geilen and T. Basten, "Reactive process networks," in *Proc. Int. Workshop on Embedded Software*, Sep. 2004, pp. 137–146.

[17] T. Henriksson and P. van der Wolf, "TTL Hardware interface: A high-level interface for streaming multiprocessor architectures," in *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia*, Seoul, Korea, Oct. 2006, pp. 127–132.

[18] D. S. Henry, "Hardware Mechanisms for Efficient Interprocessor Communication," Ph.D. dissertation, MIT, Cambridge, MA, 1996.

[19] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congr.*, 1974.

[20] V. Kianzad, S. Saha, J. Schlessman, G. Aggarwal, S. S. Bhattacharyya, W. Wolf, and R. Chellappa, "An architectural level design methodology for embedded face detection," in *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, Jersey City, NJ, Sep. 2005, pp. 136–141.

[21] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 2, Feb. 1987.

[22] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time DSP.," in *Proc. IEEE Global Telecommunications Conf. and Exhibition*, Nov. 1989, p. 12791283.

[23] H. Moon, R. Chellappa, and A. Rosenfeld, "Optimal edge-based shape detection," *IEEE Trans. Image Process.*, vol. 11, no. 11, pp. 1209–1227, Nov. 2002.

[24] M. Pankert, O. Mauss, S. Ritz, and H. Meyr, "Dynamic data flow and control flow in high level DSP code synthesis," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, Adelaide, Australia, Apr. 1994, vol. 2, pp. 449–452.

[25] S. Puthenpurayil, R. Gu, and S. S. Bhattacharyya, "Energy-aware data compression for wireless sensor networks," in *Proc. Int. Conf. Acoustics, Speech, and Signal Processing*, Honolulu, HI, Apr 2007, vol. 2, pp. 45–48.

[26] S. Saha, S. S. Bhattacharyya, and W. Wolf, "A communication interface for multiprocessor signal processing systems," in *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia*, Seoul, Korea, Oct. 2006, pp. 127–132.

[27] S. Saha, J. Schlessman, S. Puthenpurayil, S. S. Bhattacharyya, and W. Wolf, "An optimized message passing framework for signal processing applications," in *Proc. Design Automation and Test in Europe*, 2008.

[28] S. Sakai, H. Matsuoka, Y. Kodama, M. Sato, A. Shaw, H. Hirono, K. Okamoto, and

T. Yokota, "RICA: Reduced interprocessor-communication architecture-concept and mechanisms," in *Proc. 5th IEEE Symp. Parallel and Distributed Processing*, Dec. 1–4, 1993, pp. 122–125.

[29] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. New York: Marcel Dekker, 2000.

[30] S. Stuijk, M. Geilen, and T. Basten, "Exploring tradeoffs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Proc. Design Automation Conf.*, Jul. 2006.

[31] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek, "The PVM concurrent computing system: Evolution, experiences, and trends," *Parallel Comput.*, pp. 531–545, 1994.

[32] A. J. van der Wal and G. J. W. Dijk, "Efficient interprocessor communication in a tightly-coupled homogeneous multiprocessor system," in *Proc. 2nd IEEE Workshop on Future Trends of Distributed Computing Systems*, 1990, pp. 362–368.

ABOUT THE AUTHORS

Sankalita Saha received the B.Tech. (Bachelor of Technology) degree in electronics and electrical communication engineering from Indian Institute of Technology, Kharagpur, India, in 2002 and the Ph.D. degree in electrical and computer engineering from the University of Maryland, College Park, in 2007.

She is currently a Postdoctoral Scientist working at RIACS/NASA Ames Research Center, Moffett Field, CA. Her research interests are in embedded systems design with special focus on system synthesis for multiprocessor signal processing systems.

Sebastian Puthenpurayil received the B.S. degree in electronics and communications engineering from College of Engineering Trivandrum, University of Kerala, India, and the M.S. degree in electrical engineering from the University of Texas Pan American, Edinburg, TX, in 2002.

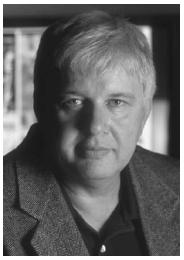
He is currently working towards the Ph.D. degree in computer engineering in the Department of Electrical and Computer Engineering at the University of Maryland, College Park. His research interests include hardware/software codesign particularly in parameterized modeling and design of energy-aware algorithms for embedded systems.

Jason Schlessman received the M.S. and B.S. degrees in computer engineering at Lehigh University, Bethlehem, PA, and is finalizing his Ph.D. dissertation in the Department of Electrical and Computer Engineering, Princeton University, Princeton, NJ. His thesis advisor is Professor W. Wolf of the Georgia Institute of Technology.

His research interests focus on embedded systems for media-centric applications. In particular, the development of FPGA- and DSP-based architectures for computer vision and biometrics. More recently, his focus has been on medical applications of embedded computer vision such as remote patient monitoring. He is the author of 28 papers ranging from his current research interests to high-performance implementations of crystallographic structure algorithms. He has been awarded several research grants, in particular through the National Science Foundation and the Yokogawa Electric Company, and is named on four patents arising from his research.

Shuvra S. Bhattacharyya received the B.S. degree from the University of Wisconsin at Madison, and the Ph.D. degree from the University of California at Berkeley.

He is a Professor in the Department of Electrical and Computer Engineering, University of Maryland at College Park. He holds a joint appointment in the University of Maryland Institute for Advanced Computer Studies (UMIACS), and an affiliate appointment in the Department of Computer Science. He is coauthor or coeditor of four books and the author or coauthor of more than 100 refereed technical articles. His research interests include VLSI signal processing; biomedical circuits and systems; embedded software; and hardware/software codesign. He has held industrial positions as a Researcher at the Hitachi America Semiconductor Research Laboratory (San Jose, CA), and Compiler Developer at Kuck & Associates (Champaign, IL).



Wayne Wolf (Fellow, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1980, 1981, and 1984, respectively.

He is Farmer Distinguished Chair and Georgia Research Alliance Eminient Scholar at the Georgia Institute of Technology, Atlanta. He was with AT&T Bell Laboratories from 1984 to 1989 and was on the faculty at Princeton University, Princeton, NJ, from 1989 to 2007. His research interests included embedded computing, embedded video and computer vision, and VLSI systems.

Prof. Wolf received the ASEE Terman Award and the IEEE Circuits and Systems Society Education Award. He is a Fellow of ACM.