In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, Munich, Germany, March 2008. To appear.

An Optimized Message Passing Framework for Parallel Implementation of Signal Processing Applications

[1]Sankalita Saha, [2]Jason Schlessman, [1]Sebastian Puthenpurayil, [1]Shuvra S. Bhattacharyya and [3]Wayne Wolf
[1]Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, USA, 20742.
[2]Department of Electrical Engineering, Princeton University, Princeton, USA, 08554.
[3]Electrical and Computer Engineering Department, Georgia Institute of Technology, Atlanta, USA, 30332.
[1]{ssaha, purayil and ssb}@umd.edu, [2]jschless@princeton.edu, [3]wayne.wolf@ece.gatech.edu

Abstract

Novel reconfigurable computing platforms enable efficient realizations of complex signal processing applications by allowing exploitation of parallelization resulting in high throughput in a cost-efficient way. However, the design of such systems poses various challenges due to the complexities posed by the applications themselves as well as the heterogeneous nature of the targeted platforms. One of the most significant challenges is communication between the various computing elements for parallel implementation. In this paper, we present a communication interface, called the signal passing interface (SPI), that attempts to overcome this challenge by integrating relevant properties of two different yet important paradigms in this context — dataflow and the message passing interface (MPI). SPI is targeted towards signal processing applications and, due to its careful specialization, more performance-efficient for their embedded implementation. It is also more easier and intuitive to use. Earlier, a preliminary version of SPI was presented [12] which was restricted to static dataflow behavior. Here, we present a more complete version of SPI with new features to address both static and dynamic dataflow behavior, and to provide new optimization techniques. We develop a hardware description language (HDL) realization of the SPI library, and demonstrate its functionality on the Xilinx Virtex-4 FPGA. Details of the HDL-based SPI library along with experiments with two signal processing applications on the FPGA are also presented.

1. Introduction

The complexity of signal processing applications is increasing rapidly, intensifying the need for multiprocessor architectures to meet execution time constraints. Recent times have witnessed the growing popularity of *platform FPGAs*, which include CPUs, embedded memory, memory interfaces, and specialized I/O interfaces along with the FPGA fabric. Such integrated platforms provide advanced support for today's embedded applications. However, the growing complexity of both the applications and the platforms makes the job of design and implementation more difficult. One of the most important issues faced in this regards is communication between the different heterogeneous processing units.

In general-purpose processing, the most widely-known endeavor in this regard is the message passing interface (MPI) protocol [3]. The main advantages of MPI are its portability — MPI has been implemented for a wide-range of architectures — and optimizations for each implementation for the hardware on which it runs. Due to various drawbacks in MPI, however, alternative protocols have been proposed, such as OpenMP [2]. MPI has been adapted for FPGA based multiprocessor systems as well [13]. However, all of these are software techniques that target general-purpose applications, and are not tuned towards the signal processing domain and hence cannot leverage optimizations obtained by exploiting characteristics specific to this application domain. In recent times, specialized interfaces and middleware for signal processing applications have come into focus [6]. However, none of these address the need for a standardized interface portable over different platforms, while retaining application-domain-specific optimizations.

In a previous effort [12], a preliminary version of a new multiprocessor communication protocol, called the signal passing interface (SPI) was proposed which is specialized for signal processing — an increasingly important application domain for embedded multiprocessor software. SPI 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 created a new paradigm for multiprocessor implementation of signal processing applications. However, the previous version of SPI had restrictions imposed by the

static nature of the underlying model of synchronous dataflow (SDF) [8]. SDF does not allow dynamic rates of data transaction between various subsystems modeled by the graph, thereby limiting the applicability of the purely-SDF-based previous version of SPI. Also, only a software implementation was developed. However, for the new emerging platforms, a corresponding hardware-oriented library implementation is necessary as well.

In this paper we introduce a number of new features in SPI to address the restrictions in the preliminary version:

1. We apply the concept of variable token size (VTS) to enable dynamic-rate data transactions (with a few restrictions) between the different subsystems modeled by a dataflow graph while maintaining many of the useful properties of SDF. This concept is then carefully integrated with SPI for efficient implementation. While previous dataflow systems for signal processing have implicitly supported VTS, we introduce the use of VTS as a more explicit modeling tool — in particular, as a means for applying more efficient and intuitive SDF techniques to certain kinds of dynamic dataflow behaviors.

2. Resynchronization is a technique used in the context of multiprocessor implementation of SDF for reducing synchronization overheads between processors. Until now, this technique has been mainly used for shared-memory systems. We extend the concept for distributed memory systems and incorporate corresponding optimizations into SPI.

3. We develop a hardware description language (HDL) realization of the SPI library, and demonstrate its functionality on the Xilinx Virtex-4 FPGA. 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 separated from the computation part.

2. Signal Passing Interface (SPI)

The format of coarse-grain dataflow graphs is one of the most natural and intuitive modeling paradigms for signal processing systems ([8, 14]). Dataflow based modeling of signal processing applications for parallel computation results in four important classes of multiprocessor implementation: fully static, self-timed, static assignment and fully dynamic methods [9]. Among these, 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 being robust with regards to actor execution times that are not exactly known or that may exhibit occasional variations [14]. Hence, the SPI methodology uses the self-timed scheduling model, although adaptations of the methodology to other scheduling models is feasible, and is an interesting topic for further investigation. The SDF model is a restricted form of dataflow which allows only static exchange of data (tokens) between actors. 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.

3. Variable Token Size (VTS) model

As mentioned in section 2, a significant limitation of SDF is that it does not allow dynamic variation in token size. In *dynamic* dataflow (which originates at dynamic ports), production/consumption rates of an actor may change at run time depending on current or previous values of its input data. General dynamic dataflow, with no apriori 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 providing for a significant degree of dynamic dataflow is 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 be 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 figure 1. The production rate of edge (A, B) varies dynamically with an upper bound of 10 and the consumption rate varies with an upper limit of 8. These varying rates can be captured using variable token sizes. Thus, A has a production rate of 1 with a token size of x and B has a consumption rate of 1 with token size of y, where x has an upper bound of 10 and y has an upper bound of 8. 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.

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

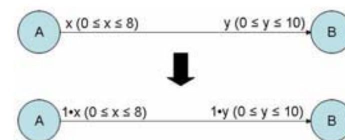

Figure 1. SDF graph with dynamic data rates and corresponding VTS conversion.

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 [14]. $c_{sdf}(e)$ is computed on the graph after VTS conversion, so it is computed on a pure SDF graph. The total size of packed tokens $c(e)$ is then given as

$$c(e) = c_{sdf}(e) \times b_{max}(e), \quad (1)$$

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 figure 1) times the maximum number of bytes in a single raw (unpacked) token for the port. The IPC buffer bounds, as computed in [14], now gets modified to

$$B(e) = (\rho_G(src(e), snk(e)) + delay(e)) \times c(e), \quad (2)$$

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 in equation 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.

3.1 Prior work on dataflow modeling and its relation to 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 of SDF. Well-behaved stream flow graphs (WBSFGs) [4], allow two specific non-SDF dynamic actors (*switch* and *select*) to model conditionals and data-dependent iteration in a restricted fashion. In Boolean dataflow (BDF) [1], the number of tokens produced or consumed by an actor is either fixed, or is a two-valued function of a control token present on a control terminal of the same actor. Reactive process networks integrates reactive behavior with dataflow to capture interaction between reactive and streaming components in multimedia applications [5]. The VTS approach developed in this paper differs from the above by providing an explicit way to allow dynamic token transfer rates within the SDF framework.

Bounded dynamic dataflow (BDDF) [11], allows arbitrary data rates if an upper bound is specified for the data rate of each dynamic port. Our approach is similar to BDDF since it requires bounds on dynamic behavior but differs by concentrating on token sizes instead of data rates. Thus, in contrast with BDDF, we can apply SDF-based analysis techniques to our VTS-based system representations. Kahn process networks (KPNs) [7] provide another modeling paradigm that is popular for signal processing. It 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 which are not present in the general KPN model, it 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 as demonstrated by tools such as Compaan [15] — is a promising direction for future work.

4. Synchronization and SPI

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 [14]. In this section, we provide details of synchronization graph and associated resynchronization technique for SPI.

The *SPI_BBS* and *SPI_UBS* protocols derived earlier in the context of shared memory systems provide buffer synchronizations between the sender and receiver [12]. Here, *BBS* and *UBS* stand for bounded buffer and unbounded buffer synchronization, respectively. These protocols were modified slightly for adaptation to distributed memory systems. They are implemented using special pointers shared between the dataflow actors and SPI actors. If it can be guaranteed that a buffer will not exceed a predetermined size, then *SPI_BBS* protocol is used. *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. Both protocols use acknowledgments to ensure consistency of data. Note that the protocols remain invariant if any of the interprocessor edges uses VTS, since that is already encapsulated in the header of the message [12].

4.1 Resynchronization

Given the dataflow graph for an application G and its multiprocessor schedule, the IPC graph G_{ipc} is derived 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 to x to y [14]. Each edge (v_j, v_i) in G_{ipc} represents the synchronization constraint

$$start(v_i, k) \geq end(v_j, k - delay(v_j, v_i)), \quad (3)$$

where $start(v, k)$ and $end(v, k)$ represent the time at which invocation k of actor v 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}, resynchronization modifies the synchronization graph. The construction of this graph depends on the underlying target platform and is a general part of SPI methodology.

The process of adding one or more new synchronization edges and removing any redundant synchronization edges that result is called *resynchronization* [14]. 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.

SPI-based implementation of a system that uses the *SPI_UBS* protocol can result in multiple redundant acknowledgements 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 acknowledgement edges for SPI actors. In our HDL-based SPI library implementation, the corresponding actors do not implement synchronization acknowledgments, they are implemented as separate messages. This technique is further illustrated using practical applications in sections 5.2 and 5.3.

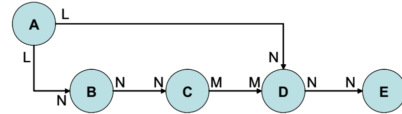

Figure 2. Dataflow graph for application 1.

5. Experiments and results

5.1 SPI library implementation

To accommodate static as well as dynamic behavior, we propose a two-phase SPI interface consisting of *SPI_static* 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 [12]. For edges that exhibit dynamic communication behavior, *SPI_dynamic* is used with limitations discussed in section 3. We developed an FPGA library for SPI 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. 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.

5.2 Speech compression (Application 1)

The first application is a LPC (linear predictive coding) based acoustic data compression (ADC). The input signal contains L samples, and these samples are divided into frames each of size N. For each frame, predictor coefficients are generated which are used to determine the predicted value of the input signal. The prediction error and its coefficients are quantized which is the compressed data. In figure 2, A reads a segment of input data, B implements Fast Fourier transform (FFT) operation on the input samples, C performs LU decomposition to find predictor coefficients, D generates the error on samples and E implements Huffman coding on the error samples.

Most of the actors have high computational intensity and the FPGA resources were not enough to fit a multiprocessor version of the whole system. Thus, we explored the parallelization of only the error generation actor (D) in hardware; thus this experiment of SPI is in the context of an overall hardware/software co-design solution. Actor D requires the input samples and the predictor coefficients. The number of coefficients (that depend on the model order M) and the size of the input frame N are not known before run-time. This leads to dynamic data transactions and use of *SPI_dynamic*. For n processing elements (PEs) computing

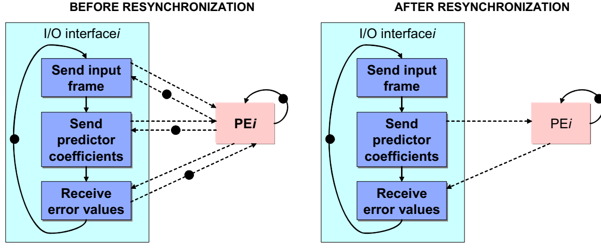

Figure 3. Resynchronization for 3-PE implementation of actor *D* in application 1.

the errors in parallel, each PE computes $\lceil N/n \rceil$ error values. Each such PE is a customized hardware unit.

Figure 3 shows the synchronization graph before and after resynchronization for a 3 PE implementation. Here, the dashed edges represent synchronization edges. Each PEi performs error calculation for the part of the frame that is sent to it which is split into overlapping sections. 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. The synchronization graph is constructed by considering appropriate SPI actors jointly with their corresponding dataflow actors. We do not show the SPI actors in the illustrated synchronization graphs for purposes of clarity and since the main aim of the graph is to illustrate the synchronization operations.

5.3 Particle filter (Application 2)

The second application used is a particle filter based system for tracking crack failure length in the blades of a turbine engine [10]. Particle filters recursively estimate the unknown state of a system from a collection of noisy observations. In figure 4, E estimates the current state using state equations, U updates the state using external observation and the observation model and S selects particles for the next iteration using resampling. In our implementation, particles are equally distributed among PEs after which all the steps execute in parallel and communicate only during resampling. Thus for N particles (N is typically large and for this system varies from 50 to 300) and n PEs, each PE handles $\lceil N/n \rceil$ particles in each iteration.

All the steps in a particle filter can be completely parallelized, except resampling. In our resampling scheme, the new samples selected are exact replicas of some of the old samples, but occurring with multiplicities proportional to

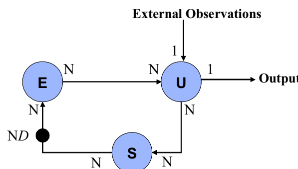

Figure 4. Dataflow graph for application 2.

their previous weights. For distributed implementation, first multiplicity factors for the particles of a given PE are calculated locally (*local resampling*). Then excess new particle values are communicated to the other PEs to ensure that all PEs have the same number of particles for the following iteration (*intra-resampling*). Thus, in this case all communication between PEs is not-deterministic.

The synchronization graphs before and after resynchronization for a 2 PE implementation is shown in figure 5. The resampling step is split into three steps — (1) calculating a partial sum and communicating it to other PEs; (2) local resampling; and (3) intra-resampling. There are two messages passed between the PEs: the first one is to exchange local sums (known length and hence *SPI_static* is used), and the second one is to exchange particles for which *SPI_dynamic* is used since it varies at run-time.

5.4 Results

The implementations were designed using Xilinx System Generator 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. Figures 6 and 7 show the performance results obtained for actor D of application 1 and application 2. For both cases, n represents the number of PEs used. Table 1 shows the FPGA area requirements for a 4 PE implementation of actor D of application 1 along with the SPI library resource requirements relative to the full system, while table 2 shows the same for a 2 PE implementation of application 2. The computational requirement for the application 2 was relatively high and hence only 2PEs could be accommodated. The SPI library area sizes are small in both cases.

6. Conclusions

In this paper, we have presented a new and significantly more flexible and optimized version of SPI. 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

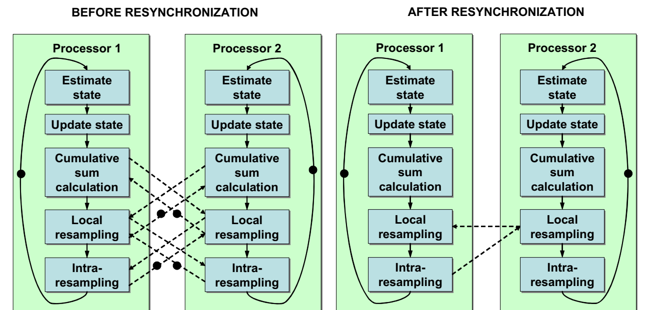

Figure 5. Resynchronization for 2-PE implementation of application 2.

upper bound on the associated data rate variation, our 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 existing framework. Resynchronization in the context of SPI has also been explored, and related synchronization optimizations have been added to the interface. A new, HDL-based library implementation for SPI has been created and demonstrated on two useful signal processing applications. These developments provide the first integration of SPI in FPGA based design. The FPGA resources used by SPI have been shown to be small compared to the overall system resources, further demonstrating the utility of SPI. Our work further 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 between communication and computation for easier development of embedded multiprocessor systems.

7. Acknowledgements

This research was supported in part by grant number 0325119 from the U.S. National Science Foundation.

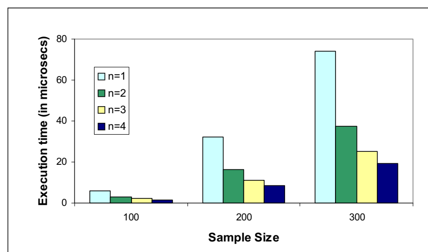

Figure 6. Performance results for actor D of application 1.

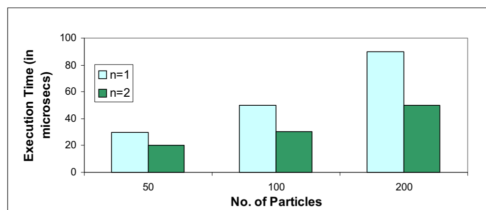

Figure 7. Performance results for application 2.

Table 1. FPGA resource requirements for 4 PE implementation of actor *D* of application 1.

	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%

8. References

[1] J.T. Buck. Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model., Ph.D. thesis, University of California at Berkeley, Sept.,1993.

[2] L. Dagnum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, pages 46–55, 1998.

[3] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, pages 84–90, 1996.

[4] G.R. Gao, R. Govindarajan, and P. Panangaden. Well-Behaved Programs for DSP Computation. In *Proc. of the Intl. Conf. on Acoustics, Speech, and Signal Processing*, March, 1992.

[5] M. Geilen and T. Basten. Reactive Process Networks. In *Proc. of Intl. Wkshp. on Embedded Software,* Sept. 2004.

[6] T. Henriksson and P. van der Wolf. TTL Hardware Interface: A High-Level Interface for Streaming Multiprocessor Architectures. In *Proc. of the IEEE Wkshp. on Embedded Systems for Real-Time Multimedia*, pp. 127-132, Korea, Oct. 2006.

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

[8] E.A. Lee and D.G. Messerschmitt. Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing. *IEEE Trans. on Computers*, Vol. C-36, No. 2, Feb., 1987.

[9] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *Proc. of IEEE Global Telecommunications Conf. and Exhibition*, pages 1279–1283, Nov 1989.

[10] M.Orchard, B. Wu and G. Vachtsevanos. A Particle Filter Framework for Failure Prognosis. In *Proc. of World Tribology Congress III*, USA, Sept. 2005.

[11] M. Pankert, O. Mauss, S. Ritz, H. Meyr. Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis. In *Proc. of the IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Vol. 2, pp 449-452, Australia, Apr. 1994.

[12] S. Saha, S. S. Bhattacharyya, and W. Wolf. A communication interface for multiprocessor signal processing systems. In *Proc. of the IEEE Wkshp. on Embedded Systems for Real-Time Multimedia*, pp. 127-132, Korea, Oct. 2006.

[13] M. Saldaña and P. Chow: TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. I*n Proc. of International Conference on Field Programmable Logic and Applications*, pp. 1-6, Aug. 2006.

[14] S. Sriram and S. S. Bhattacharyya. Embedded Multiprocessors: Scheduling and Synchronization. Marcel Dekker, Inc., 2000.

[15] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using Kahn process networks: The Compaan/Laura approach. In *Proc. Design Automation Test in Europe (DATE)*, 2004, vol.1, pp. 340–345.

Table 2. FPGA resource requirements for 2 PE implementation of application 2.

	Slices	Slice FFs	4 input LUTs	Block RAMs	DSP 48s
Full system	90.74 %	47.52%	65.48%	18.23%	56.25%
SPI library (relative to full system)	0.2%	0.08%	0.27%	11.43%	0%