In Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, Austin, Texas, February 2009.

System-level Clustering and Timing Analysis for GALS-based Dataflow Architectures

Chung-Ching Shen and Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering,
and Institute for Advanced Computer Studies,
University of Maryland, College Park, 20742, USA.
{ccshen, ssb}@umd.edu

Abstract — **In this paper, we propose an approach based on dataflow techniques for modeling application-specific, globally asynchronous, locally synchronous (GALS) architectures for digital signal processing (DSP) applications, and analyzing the performance of such architectures. Dataflow-based techniques are attractive for DSP applications because they allow application behavior to be represented formally, analyzed at a high level of abstraction, and synthesized to software/hardware implementations through an optimized, automated process. In our proposed methodology, we employ dataflow-based computational models to expose relevant structure in the targeted applications, and facilitate the manual or automatic derivation of efficient implementations. We demonstrate the utility of our modeling and analysis techniques by applying them as core parts of a novel clustering algorithm that is geared towards optimizing the throughput of GALS-based DSP architectures.**

I. Introduction and Related Work

Globally asynchronous, locally synchronous (GALS) [3] design is a general, hybrid approach to digital system design that combines the benefits of both the synchronous and asynchronous digital design methodologies. Key features of a GALS system are the absence of a global timing reference, and the use of distinct local clocks that are in general independent (e.g., the clocks can have different frequencies and relative phases). In a typical configuration of a GALS system, a GALS module (or so-called *synchronous island*) consists of a synchronous module, an asynchronous wrapper, and a local clock generator, where the synchronous module is encapsulated by the asynchronous wrapper. GALS modules communicate with each other via pre-defined asynchronous interfaces. On-chip ring oscillators are commonly used to implement the local clock generators that are used in synchronous islands (e.g., see [13]).

Dataflow is a well known computational model and is widely used for expressing the functionality of digital signal processing (DSP) applications, such as audio, digital communications, and video data stream processing (e.g., see [1], [12]). These applications usually require real-time processing capabilities, and therefore, system performance becomes critical. Dataflow provides a formal mechanism for describing specifications of DSP applications, imposes minimal data-dependency constraints in specifications, and is effective in exposing and exploiting concurrency. In dataflow specifications, computational processes communicate via first-in-first-out (FIFO) channels. This semantic behavior is similar to the specification of a GALS system, where the processes can be seen as synchronous components, and asynchronous FIFOs can be used as asynchronous interfaces.

For example, the authors in [18] introduce the idea of using the dataflow model of computation for designing GALS systems and discuss advantages and disadvantages associated with different GALS-based dataflow architectures. We refer to the dataflow architectures discussed in [18] as *isomorphic* GALS configurations for a given application. More specifically, by an isomorphic GALS configuration for a dataflow graph, we mean a configuration in which there is a one-to-one correspondence between hardware resources in the architecture and functional modules (dataflow graph vertices or "actors") in the dataflow graph.

In this paper, we propose a design and evaluation framework for modeling application-specific GALS-based dataflow architectures for DSP applications, and analyzing the associated system performance (or throughput). Dataflow modeling and analysis for DSP applications is attractive because the application behavior can be modeled formally, analyzed at high level, and synthesized to software/hardware implementations through an optimized, automated process. Our proposed framework incorporates dataflow-based modeling, design analysis, and explores optimization opportunities for such an automatic, DSP-oriented synthesis process.

Most previous research on GALS-based design is focused on circuit-level design of system components, such as designs for asynchronous interfaces (e.g., see [4], [19]), and analysis of different architectural design styles (e.g., see [18]).

A smaller proportion of the previous research has explored system-level optimization trade-offs — e.g., among power, area, and performance — and development of design automation support. For example, Hemani et al. proposed a GALS design method for partitioning a system into an optimal configuration of synchronous blocks, and explored relationships between power consumption and the number of synchronous blocks (i.e., the granularity of the derived configurations) [6]. Hemani's method is useful for large, high performance VLSI systems, where clock trees are a major source of power consumption. Two limitations of Hemani's approach are that the fixed sizes for synchronous blocks limit flexibility for any subsequent optimization steps, and the approach does not take system performance into account in the optimization process.

As another example, Wuu [21] presents a method for automatic synthesis of asynchronous digital systems from high-level dataflow specifications. The authors present an extended dataflow model that accurately reflects the behavior of the employed asynchronous components so that the dataflow specification can be directly mapped into a hardware realization. Data transfers between pairs of communicating blocks rely on a handshaking protocol, and the asynchronous signals are bundled with the data signals to ensure proper operation. Wuu's approach is developed for *fine-grained* dataflow graphs — that is, graphs in which the dataflow graph vertices (*actors*) correspond to primitive, combinational functions, such as addition and multiplication.

In contrast to Wuu's approach, the system-level optimization methodology that we present in this paper applies to the more general class of *mixed-grain* dataflow graphs, where actors can have arbitrary complexity — in our context of DSP system design, the actors typically range in complexity from primitive operations to complete signal processing modules, such as digital filters and fast Fourier transform units. Furthermore, our methods consider both combinational and sequential actors, and opportunities to pipeline combinational actors to increase their levels of throughput.

In the proposed GALS-based dataflow framework, we introduce a systematic technique aimed at analyzing the performance of an application-specific GALS-based system at a high level of abstraction. We employ dataflow-based computational models to expose relevant structure in our targeted applications, and facilitate the manual or automatic derivation of efficient implementations.

The rest of paper is organized as follows. Section II provides relevant background on dataflow modeling. Section III introduces a novel clustering methodology for optimized implementation of GALS-based dataflow architectures. Section IV develops methods for modeling the timing of GALS--based dataflow actors, and corresponding methods for analyzing the performance of systems that are composed of such actors. DSP application examples and experimental results are presented in Section V. Finally, Section VI concludes the paper and discusses useful directions for future work.

II. Background

A. Dataflow Modeling

Dataflow models of computation are widely used for modeling DSP applications. Dataflow graphs are often used to represent computational behavior for such systems due to the intuitive match with signal processing block diagrams or signal flow graphs. At the same time, dataflow models of computation provide formal semantics that are valuable for effective system design and analysis (e.g., see [1]). There is wide variety of development environments that utilize dataflow models to aid in the design and implementation of DSP applications (e.g., see [5], [8], [10], [15]). In these tools, an application designer is able to develop and describe the functionality of an actor, and capabilities are provided for automated system simulation or synthesis.

Synchronous dataflow (SDF) [12] is a restricted form of dataflow that is useful for an important class of DSP applications, and is used in a variety of commercial design tools. SDF in its pure form can only represent applications in which actors produce and consume constant amounts of data with respect to their input and output ports — that is, data-dependent or otherwise dynamic data rates are not supported in SDF. By using SDF, we can efficiently check conditions such as whether a given SDF graph deadlocks, and whether it can be implemented using a finite amount of memory (e.g., see [17]).

An SDF graph, $G = (V, E)$, is a directed graph in which each vertex (actor) $v \in V$ represents a computational module for executing a given task, and each directed edge, $e \in E$, represents a first-in-first-out (FIFO) buffer for holding data values (tokens) and imposing data dependencies. SDF graphs represent computations that are executed iteratively on data streams that typically contain large, often unbounded numbers of data samples. Each SDF actor therefore executes through a sequence of executions (*invocations*) as the enclosing SDF graph operates.

An actor is enabled to execute its computational task (i.e., its next invocation) only if sufficient numbers of tokens are available on its input edges. An actor produces certain numbers of tokens on its output edges as it executes. For proper operation, the output edges onto which an actor invocation produces data must have sufficient numbers of "empty spaces" (vacant positions in the corresponding FIFO buffers).

A non-negative-integer *delay* is associated with each dataflow edge. Each unit of delay corresponds to an initial token that is stored in the associated buffer. An SDF graph in which every actor consumes and produces only one token from each of its inputs and outputs is called a *homogeneous SDF* graph (*HSDF* graph).

In this paper, we assume that the given application representation is in the form of an HSDF graph. A general SDF graph that is not an HSDF graph can always be converted into an equivalent HSDF graph (e.g., see [12], [17]).

In addition to using HSDF for modeling application behavior, we also use HSDF to represent application-specific, GALS-based dataflow architectures. In this architectural representation, each actor v corresponds to a GALS synchronous island that implements the functionality corresponding to v. Similarly, each edge e corresponds to an asynchronous FIFO that implements a communication channel between a pair of synchronous islands.

In a practical sense, there is some loss of generality in restricting ourselves to HSDF representations as we do in this paper. This is because the conversion from an SDF graph to its equivalent HSDF graph can require running time that scales exponentially with the number of actors in the input SDF graph. Therefore, the conversion may become infeasibly time-consuming for some applications (e.g., for heavily multirate applications, such as those explored in [7]). GALS-based techniques that operate directly on general SDF graphs, and avoid conversion to HSDF representations are a useful direction for further study.

B. Dataflow Interchange Format

The Dataflow Interchange Format (DIF), and the associated design language called The DIF Language (TDL) [8] provide a standard language for specifying mixed-grain dataflow models of DSP systems. TDL provides a unified textual language for expressing different kinds of dataflow semantics. The associated DIF package is a software package that provides representations and utilities for analyzing dataflow graphs that are captured using TDL. By using the TDL, a DSP application can be described and modeled as a dataflow graph at a coarse-grained level. With the support of module libraries for the dataflow actors, an efficient software implementation for such an application can be automatically synthesized through a software synthesis process that has been demonstrated in [9]. This synthesis process is capable of exploring a wide range of useful implementation trade-offs that are exposed effectively through TDL-based, mixed grain dataflow representations [9].

In this paper, we employ TDL as a design language for specifying and analyzing DSP applications for GALS-based

implementation. In TDL specifications, each dataflow actor is allowed to have a set of attributes, which can involve any combination of built-in (supported through TDL keywords) or user-defined attributes. This feature provides a convenient way to annotate a dataflow design with platform-specific properties (e.g., power consumption, timing information, etc.) of actors when a specific hardware platform is targeted for implementation.

III. Topological Sorting Clustering for GALS

Graph *clustering* refers to the grouping together of vertices in a graph so that they can be treated together as a single unit in subsequent stages of a given design flow. Thus, clustering provides a conceptual framework for incrementally constraining the solution space for a graph. Clustering techniques have been discussed for many application domains (e.g., see [11]). In order to take into account relevant topological properties associated with GALS-based dataflow architectures, we propose a novel clustering methodology called *topological sorting clustering* (or *TS clustering*). TS clustering is geared towards transforming a given application dataflow graph into a clustered dataflow graph that represents the GALS structure of a dataflow architecture. In TS clustering, each cluster (grouped set of actors) corresponds to a synchronous island in the targeted implementation.

TS clustering is based on guiding the clustering process with a specific topological sort. Given an acyclic directed graph, a topological sort is an ordering of the graph vertices, such that for every edge (x, y) in the graph, x occurs earlier in the ordering than y. A topological sort can be computed efficiently, e.g. using depth-first-search (DFS) techniques. Different topological sorts can be mapped into different candidate clusterings using our concept of TS clustering; this provides a flexible means for design space exploration since a variety of strategies can be selected and configured to generate topological sorts (e.g, randomized DFS or evolutionary algorithms) based on the time budget allowed for exploration. Moreover, the "best" solution chosen from these clustering candidates depends entirely on the given evaluation metrics. Arbitrary collections of relevant metrics can be integrated with our clustering methodology to provide for single- or multi-objective optimization processes. For example, in the prototyping stage, a simple deterministic DFS might be used to generate a functionally correct solution, while late stages of the design process might encode topological sortings into an evolutionary algorithm to derive highly optimized clustering results.

Fig. 1 outlines the TS clustering approach. Here, $TS_1, \ldots, TS_N$ represents a topological sort for the given dataflow graph $G = (V, E)$. The core of the clustering process is a loop that instantiates a new cluster only when it finds an adjacent pair of vertices (actors in G) in the topological sort that are not connected by an edge in G. Each group of actors that lies between successive boundaries of "non-connected adjacent pairs" is consolidated as a distinct cluster.

The result of this clustering process is a *clustered graph* $G_c = (V_c, E_c)$, where each vertex $u \in V_c$ represents a cluster, and each edge $(x, y) \in E$ corresponds to an edge in G that connects a vertex in cluster x to a vertex in cluster y. Multiple edges between the same pair of clustered subsets in G result in multiple edges between the corresponding pair of vertices in G_c, and thus, technically, G_c can be viewed as a directed multigraph. As implied earlier, each vertex $u \in V_c$ corresponds to a synchronous island (asynchronous FIFO) in the candidate GALS implementation being evaluated.

Since a clustered graph is also a dataflow graph (representing "inter-cluster" or "inter-island" dataflow), we can extend certain well-developed forms of dataflow graph analysis to analyze clustered graphs, and the GALS implementations that they represent. For example, performance analysis for self-timed multiprocessor implementation (e.g., see [2, 17]) is applied in this paper to evaluate the system performance associated with a clustering solution. This form of performance analysis is discussed further in Section IV.

Based on our introduced clustered graph representation, we can identify two special cases of clustering results that correspond, respectively, to fully synchronous implementation and "uniformly-primitive" (single-actor-islands) GALS implementation. More specifically, a clustered graph that consists of only one vertex (i.e., all actors in the application graph are clustered into a single cluster) represents a fully synchronous configuration. In such a case, all actor implementations share the same clock source, and no asynchronous FIFOs are used in the final implementation. Conversely, a clustered graph in which each application graph actor is mapped to its own separate cluster corresponds to a primitive GALS configuration, where asynchronous interfacing is associated with every edge in the application graph.

Our proposed clustering methodology, when integrated with the performance analysis methods described in the following section, provides an efficient approach for exploring GALS implementation trade-offs between these conventional special cases or "extremes" of fully-synchronous and uniformly-primitive configurations. Such exploration is useful for performance-critical applications because in general both of these extremes may be significantly outperformed by solutions that exhibit intermediate levels of granularity in the constructed synchronous islands.

IV. System-level Timing and Performance Analysis

A. Timing Analysis

We develop a system-level timing analysis model based on

```
TopClustering(G, TS_1, ..., TS_N)
  for i = 1 to N
    create an initial cluster C_ik
    p = C_ik, k = 1
    for j = 1 to |TS_j|
      add v_j to p
      if no edge exists between v_j and v_{j+1}
        k = k + 1
        create a new cluster C_ik
        p = C_ik
      end
    end
    construct a graph G_{c,i} based on all cluster results
    (i.e. C_ik) and G
  end
  return G_{c,1}, G_{c,2}, ..., G_{c,N}
```

Figure 1. Algorithm for constructing clustered graphs.

the following timing attributes that are defined for an implementation of a given actor v. Here, we consider the computation time (i.e., the time to complete the required processing of data) and assume that setup time and hold time of registers are negligible compared to the computation time of each actor v. The timing attributes for an implementation of an actor v are defined as follows. Here, path lengths are interpreted in terms of the cumulative computational delays. In our experiments, we have calculated the computational delay of a path as the cumulative gate delay, but these models and our analysis can easily be extended to use path lengths that are in terms of cumulative gate and interconnect delays.

- *Maximum input combinational latency* ($l_{\alpha, v}$): the computational latency of a longest combinational path between an input and a register in v. If there is no combinational path between an input and a register, then $l_{\alpha, v} = 0$. Note when such a combinational path exists, there is not necessarily a *unique* longest combinational path.
- *Maximum inter-register latency* ($l_{\beta, v}$): the computational latency of a longest combinational path between any two registers in v. If there is no pair of registers that is connected by a combinational path, then $l_{\beta, v} = 0$.
- *Maximum output combinational latency* ($l_{\gamma, v}$): the computational latency of a longest combinational path between a register and an output in v. If no such combinational path exists, then $l_{\gamma, v} = 0$.
- *Maximum input/output combinational latency* ($l_{\theta, v}$): the computational latency of a longest combinational path between an input and an output in v. If there is no such combinational path, then $l_{\theta, v} = 0$.

Fig. 2 shows an example for defining these timing attributes for an implementation of a 4-tap finite impulse response (FIR) filter actor. In this example, there are three registers (each register is labeled with an "R" in the figure).

The computational latency of an actor implementation is limited by $l_{\alpha, v}$, $l_{\beta, v}$, $l_{\gamma, v}$ or $l_{\theta, v}$, as well as the number of pipeline stages in v, which we denote as ps_v. That is,

$$t(v) = max\{l_{\alpha, v}, l_{\beta, v}, l_{\gamma, v}, l_{\theta, v}\} \times (ps_v + 1), \quad (1)$$

where, the term $max\{l_{\alpha, v}, l_{\beta, v}, l_{\gamma, v}, l_{\theta, v}\}$ can be viewed as the length of a critical path in the given implementation of actor v.

Note that for an actor v that is implemented by combinational circuits and without pipelining (i.e., $ps_v = 0$), we have that $l_{\alpha, v} = l_{\beta, v} = l_{\gamma, v} = 0$. Thus,

$$t(v) = l_{\theta, v}. \quad (2)$$

On the other hand, for an actor v that is implemented by pipelined combinational circuitry (i.e., $ps_v \neq 0$) and $l_{\theta, v} = 0$), we have that

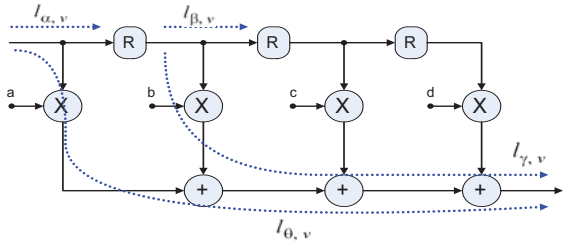

Figure 2. Timing attributes for a 4-tap FIR filter actor.

$$t(v) = max\{l_{\alpha, v}, l_{\beta, v}, l_{\gamma, v}\} \times (ps_v + 1). \quad (3)$$

In this case, $l_{\beta, v} = 0$ if $ps_v = 1$ (a single pipeline stage); otherwise, $l_{\beta, v} \neq 0$.

For an actor v that is implemented by sequential circuitry or pipelining, the maximum frequency allowed for clocking the registers in v is restricted to the critical path, and therefore is given by

$$f_{max, v} = \frac{1}{max\{l_{\alpha, v}, l_{\beta, v}, l_{\gamma, v}, l_{\theta, v}\}}. \quad (4)$$

We integrate these timing models into our proposed TS clustering framework by first assigning timing attributes to all actors at design time. As discussed in Section III, a TS clustering result obtained from an application graph $G = (V, E)$ forms a clustered graph $G_c = (V_c, E_c)$, where $u \in V_c$ represents a synchronous island that is based on a subset of actors in G. When mapping G_c into a GALS implementation, pipelining registers can in general be instantiated within synchronous islands of G_c in order to improve performance after clustering. In other words, it may be possible to improve performance by inserting pipelining registers along selected *IAG* (*intra-cluster application graph*) edges (i.e., edges of the form $(x, y) \in G$, where x and y belong to the same cluster u in G_c). Fig. 3 provides an algorithm for determining, for each IAG, whether or not such a pipelining register improves the overall system performance.

The algorithm shown in Fig. 3 ensures that the critical path for any $u \in V_c$ depends on an actor clustered within u that has maximal critical path among all actors clustered within u. Therefore, the maximum allowed clock frequency for an instantiated synchronous island $u \in V_c$ is given by

$$f_{max, u} = \frac{1}{max\{\Phi_1, \ldots, \Phi_{|u|}\}}, \quad (5)$$

where $\Phi_i = max\{l_{\alpha, v_i}, l_{\beta, v_i}, l_{\gamma, v_i}, l_{\theta, v_i}\}$ for $i = 1, 2, \ldots, |u|$. Furthermore, the computational latency for $u \in V_c$ is

$$t(u) = \frac{(\mathrm{PS}(u) + 1)}{f_{max, u}}. \quad (6)$$

Here, $\mathrm{PS}(u)$ denotes the total number of stages in a synchronous island, which can be computed by summing the pipeline stage counts of the actors in the island plus the number of registers that are inserted on IAG edges during the cluster timing analysis process.

B. System Performance Analysis

We evaluate overall system performance by analyzing the

ClusterTimingAnalysis($u \in V_c$)
for each IAG (v_i, v_{i+1}) associated with u
  $\Phi = max\{l_{\alpha, v_i}, l_{\beta, v_i}, l_{\gamma, v_i}, l_{\theta, v_i}\}$
  $\Omega = max\{l_{\alpha, v_{i+1}}, l_{\beta, v_{i+1}}, l_{\gamma, v_{i+1}}, l_{\theta, v_{i+1}}\}$
  if $max\{l_{\gamma, v_i}, l_{\theta, v_i}\} + max\{l_{\alpha, v_{i+1}}, l_{\theta, v_{i+1}}\} \geq max\{\Phi, \Omega\}$
    instantiate a register between (v_i, v_{i+1})
  end
end

Figure 3. Algorithm for IAG register insertion.

clustered graph after TS clustering and register insertion (cluster timing analysis). Based on our method for cluster (synchronous island) timing analysis, we first optimizing the timing for each synchronous island (i.e., each $u \in V_c$). Then we estimate the overall system throughput (i.e., the performance across all clusters) using maximum cycle mean analysis for self-timed parallel computations (e.g., see [16], [17]).

As we have discussed earlier, each cluster in a clustered graph represents a separate synchronous island for GALS implementation. *Bounded buffers* and *unbounded buffers* are two general types of storage structures that can be synthesized for implementing the asynchronous FIFOs (self-timed communication) that connect communicating synchronous islands. Bounded buffers are buffers for which guaranteed upper bounds on storage requirements can be derived statically based on properties of the GALS architecture and its underlying HSDF representation. In contrast, no such graph-based guarantees are available for unbounded buffers, and therefore, circuitry must be provided to prevent an actor from writing into a buffer that is already full. The overhead associated with such overflow checking can be avoided entirely for bounded buffers. A detailed treatment of bounded buffer optimizations for self-timed dataflow graph implementation is developed in [2, 17].

We use the *Convert-to-SC-graph* and *DetermineDelays* algorithms from [2] to bound and estimate the sizes of all asynchronous FIFOs prior to synthesizing a GALS implementation from a clustered graph. The goal of applying *Convert-to-SC-graph* in our context is to transform a clustered graph that is not strongly connected into a strongly connected graph. A strongly connected directed graph or multigraph is one in which there is a directed path from x to y for any ordered pair (x, y) of distinct vertices. If the HSDF representation of a GALS configuration is strongly connected, then every edge e can be synthesized as a bounded buffer, and furthermore, the buffer depth for each edge e can be determined as

$$min(Delay(C) | C \text{ is a cycle that contains } e), \quad (7)$$

where $Delay(C)$ represents the sum of all edge delays (numbers of initial tokens) over the edges in the cycle C.

The *DetermineDelays* algorithm is applied here to prevent any unexpected deadlock and lower estimated throughput (higher maximum cycle mean) from using the *Convert-to-SC-graph* algorithm for graph conversion. Due to page limitations, we refer the reader to [17] for further details about the *Convert-to-SC-graph* and *DetermineDelays* algorithms.

After applying *Convert-to-SC-graph* and *DetermineDelays* on a clustered graph G_c, the throughput of a GALS-based implementation of G_c can be evaluated as

$$T(G_c) = \frac{1}{MCM(G_c)}, \quad (8)$$

where the denominator represents the maximum cycle mean of G_c [16]:

$$MCM(G_c) = \max_{\text{cycle } C \text{ in } G_c} \left\{ \frac{\sum_{u \text{ is on } C} t(u)}{Delay(C)} \right\}. \quad (9)$$

Here, $t(u)$ represents the computational latency of the synchronous island u in G_c. The value of $t(u)$ for each u can be derived efficiently based on the timing model presented in Section IV.A, and our developed methods for cluster timing analysis. Moreover, $Delay(C)$ here is estimated by considering the number of registers used in each IAG in the cycle C in G_c.

Our overall evaluation framework for GALS-based DSP architectures is outlined in Fig. 4. Here *AllTopologicalSorts* represents the set of all topological sorts to be considered in the given design evaluation phase. This framework allows us to optimize across a broad space of candidate GALS architectures, where each candidate corresponds to a set of clusters, and each such cluster forms a synchronous island that communicates with other synchronous islands through asynchronous FIFOs.

V. Experiments

We use a depth-1, uniform-tree filterbank in our experiments. This kind of DSP application it is commonly used in audio coding [20].

Fig. 5(a) and (b) show, respectively, the SDF graph and the corresponding HSDF graph for the targeted filterbank application. Fig. 6 shows specifications for all of the actors used in the application. Here, all FIR actors have 18 taps, but have varying pipelining configurations (i.e., either with 16 or 17 pipelining stages). The multipliers and adders used in the FIR filters are assumed to have 4 time units and 2 time units, respectively, as their computational latencies. These latency values are taken from representative design examples in [14].

Two possible clustering results from applying TS clustering are illustrated in 5(c). The dashed edges represent edges that are added by the post-processing step of Convert-to-SC-graph. By applying our methods for cluster timing analysis, we find that a pipeline register is instantiated on all IAG edges.

Fig. 7 shows experimental results for evaluating system performance using (8) and (9) on the uniformly-primitive

SystemPerformanceEvaluation ($G = (V, E)$)
  $\{TS_1, \ldots, TS_N\}$ = *AllTopologicalSorts*(G);
  $\{G_{c,1}, \ldots, G_{c,N}\}$ = *TopClustering*(G, $TS_1, \ldots, TS_N$)
  for $i = 1$ **to** N
    G_c = *Convert-to-SC-graph*($G_{c,i}$);
    G_c = *DetermineDelays*(G_c);
    $T_i(G_c)$ = *EstimatePerformance*(G_c); (based on (8) and (9))
  end
  return system performance evaluation for all clustering results, i.e. $T_1, T_2, \ldots, T_N$

Figure 4. Performance evaluation algorithm for GALS-based dataflow architectures.

Actors	Timing Attributes (units)	Pipeline Stages
READ	(0 ,0 , 0, 3)	0
FORK	(0 ,0 , 0, 3)	0
FIR1	(4, 6, 6, 0)	17
FIR2	(6 , 6, 6, 0)	16
FIR3	(4 , 6, 6, 0)	17
FIR4	(6 , 6, 6, 0)	16
ADD	(0 ,0 , 0, 2)	0
WRITE	(0 ,0 , 0, 3)	0

Figure 6. Details of actor implementations in the filterbank application.

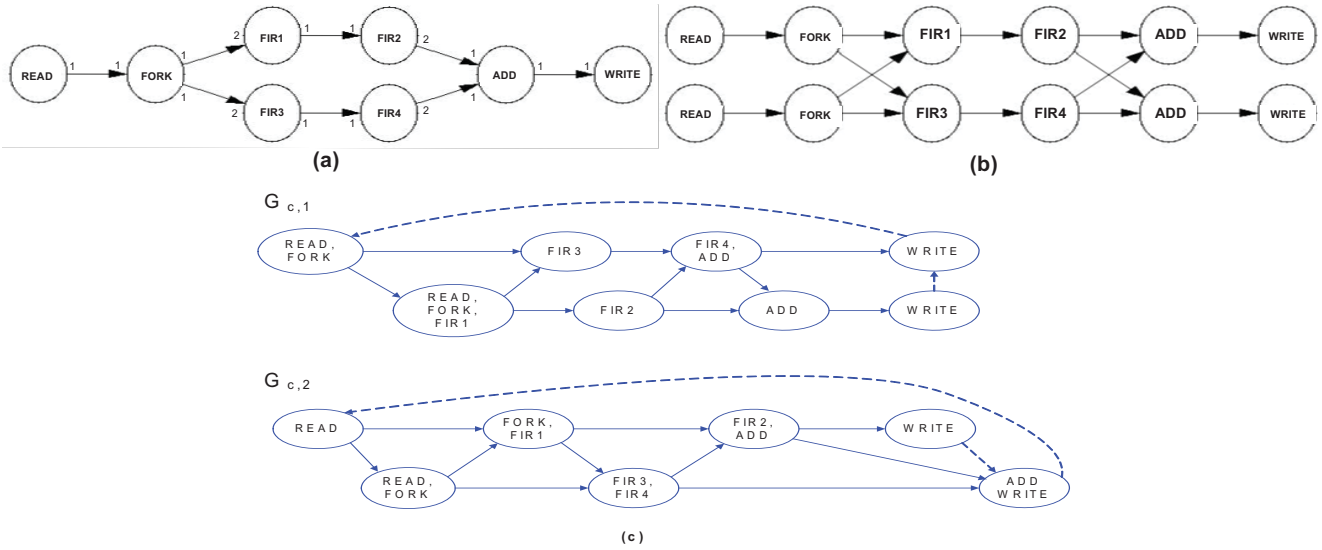

Figure 5. (a) SDF graph; (b) HSDF graph; and (c) two alternative clustering results for the filterbank application.

GALS configuration as well as on GALS configurations that are derived from the selected clustering results. From the results shown in Fig. 7, we observe that through our clustering approach, system performance can be improved by approximately 8% compared to the primitive GALS configuration.

VI. Conclusion and Future Work

We have developed a framework for modeling the performance of dataflow actors, and evaluating and optimizing the overall system performance of GALS-based architectures for DSP applications. We have also introduced a clustering technique called TS clustering. TS clustering is geared towards exploring alternative GALS architectures for a given dataflow-based application representation. Using our clustering technique associated with our performance analysis methods, we have demonstrated efficient, high-level exploration of alternative GALS architectures, and potential for significant performance improvement compared to conventional GALS-based implementation methods.

Useful directions for future work including incorporating automated hardware synthesis capabilities as a back-end to the proposed methods. We are also expanding the proposed evaluation framework by considering other important implementation metrics, such as communication latency, power consumption, and area for a more comprehensive, multi-objective optimization.

References

[1] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 21(2):151-166, June 1999.

[2] S. S. Bhattacharyya, S. Sriram, and E. A. Lee. Optimizing synchronization in multiprocessor DSP systems. *IEEE Transactions on Signal Processing*, 45(6):1605-1618, June 1997.

[3] D. M. Chapiro, *Globally-asynchronous Locally-synchronous Systems*, Ph.D thesis, Stanford University, October 1984.

[4] T. Chelceq and S. M. Nowick. Low-latency asynchronous FIFO's using token rings. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuirs and Systems*, pp. 210-220, April 2000.

[5] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. In *Proceedings of the IEEE*, pp. 127-144, January 2003.

[6] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, and D.Lundqvist. Lowering power consumption in clock by using globally asynchronous locally synchronous design style. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pp. 873-878, 1999.

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

[8] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya. DIF: An interchange format for dataflow-based design tools. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, July 2004.

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

[10] G. W. Johnson. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*. McGraw-Hill, 1997.

[11] V. Kianzad and S. S. Bhattacharyya. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):667-680, July 2006.

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

[13] K. Niyogi and D. Marculescu. System level power and performance modeling of GALS point-to-point communication interfaces. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, pp. 381-386, August 2005.

[14] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, Inc., 1999.

[15] J. L. Pino and K. Kalbasi. Cosimulating synchronous DSP applications with analog RF circuits. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1998.

[16] R. Reiter. Scheduling parallel computations. *Journal of the Association for Computing Machinery*, October 1968.

[17] S. Sriram and S. S. Bhattacharyya. Embedded *Multiprocessors: scheduling and synchronization*. Marcel Dekker, Inc., 2000.

[18] S. Suhaib, D. Mathaikutty, and S. Shukla. Dataflow architectures for GALS. *Electronic Notes in Theoretical Computer Science*, 200(1): 33-50, February 2008.

[19] P. Teehan, M. Greenstreet, and G. Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design and Test of Computers*, 24(5): 418-428, September 2007.

[20] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*, Prentice Hall, 1993.

[21] Tzyh-Yung Wuu and Sarma B. K. Vrudhula. Synthesis of asynchronous systems from data flow specifications. *T*echnical Report ISI/RR-93-366, Information Science Institute, University of Southern California, December 1993.

Configurations for GALS	System performance (Throughput)
primitive	0.146
clustering 1	0.154
clustering 2	0.158

Figure 7. Evaluation results.