Multithreaded Simulation for Synchronous Dataflow Graphs

CHIA-JUI HSU and JOSÉ LUIS PINO, Agilent Technologies, Inc.
SHUVRA S. BHATTACHARYYA, University of Maryland, College Park

For system simulation, Synchronous DataFlow (SDF) has been widely used as a core model of computation in design tools for digital communication and signal processing systems. The traditional approach for simulating SDF graphs is to compute and execute static schedules in single-processor desktop environments. Nowadays, however, multicore processors are increasingly popular desktop platforms for their potential performance improvements through thread-level parallelism. Without novel scheduling and simulation techniques that explicitly explore thread-level parallelism for executing SDF graphs, current design tools gain only minimal performance improvements on multicore platforms. In this article, we present a new multithreaded simulation scheduler, called MSS, to provide simulation runtime speedup for executing SDF graphs on multicore processors. MSS strategically integrates graph clustering, intracluster scheduling, actor vectorization, and intercluster buffering techniques to construct InterThread Communication (ITC) graphs at compile-time. MSS then applies efficient synchronization and dynamic scheduling techniques at runtime for executing ITC graphs in multithreaded environments. We have implemented MSS in the Advanced Design System (ADS) from Agilent Technologies. On an Intel dual-core, hyper-threading (4 processing units) processor, our results from this implementation demonstrate up to 3.5 times speedup in simulating modern wireless communication systems (e.g., WCDMA3G, CDMA 2000, WiMax, EDGE, and Digital TV).

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Algorithms, Design

Additional Key Words and Phrases: Synchronous dataflow, multithreaded simulation, scheduling

ACM Reference Format:
Hsu, C.-J., Pino, J. L., and Bhattacharyya, S. S. 2011. Multithreaded simulation for synchronous dataflow graphs. ACM Trans. Des. Autom. Electron. Syst. 16, 3, Article 25 (June 2011), 23 pages.
DOI = 10.1145/1970353.1970358 http://doi.acm.org/10.1145/1970353.1970358

1. INTRODUCTION

System-level modeling, simulation, and synthesis using Electronic Design Automation (EDA) tools are key steps in the design process for communication and signal processing systems. The Synchronous DataFlow (SDF) [Lee and Messerschmitt 1987] model of computation and its closely related models are widely used in EDA tools for these purposes. A variety of commercial and research-oriented tools incorporate SDF semantics, including ADS [Pino and Kalbasi 1998] and SystemVue [Hsu et al. 2010] from Agilent Technologies, CoCentric System Studio from Synopsis [Buck and Vaidyanathan 2000], Compaan from Leiden University [Stefanov et al. 2004],

Authors' addresses: C.-J. Hsu (corresponding author) and J. L. Pino, Agilent Technologies, Inc.; email: jerry_hsu@agilent.com; S. S. Bhattacharyya, Department of Electrical and Computer Engineering, University of Maryland, College Park, MD.

DOI 10.1145/1970353.1970358 http://doi.acm.org/10.1145/1970353.1970358

LabVIEW from National Instruments, PeaCE from Seoul National University [Sung et al. 1997], Ptolemy II from U. C. Berkeley [Eker et al. 2003], and DIF from the University of Maryland [Hsu et al. 2005].

Nowadays, multicore processors are increasingly popular for their potential performance improvements through on-chip, thread-level parallelism. However, without novel scheduling and simulation techniques that explicitly explore thread-level parallelism for executing SDF graphs, current EDA tools gain only minimal performance improvements from these new sets of processors. This is largely due to the sequential (single-thread) SDF execution semantics that underlies these tools.

In this article, we focus on multithreaded simulation of SDF graphs. Our objective is to *speed up* simulation runtime for modern wireless communication and signal processing systems. According to Hsu et al. [2006], SDF representations of such systems typically involve large complexity and heavily multirate behavior.

The key problem behind multithreaded SDF simulation is scheduling SDF graphs for thread-level parallel computation. Scheduling in our context consists of the following related tasks.

(1) *Clustering*. Partitioning and clustering actors in the SDF graph into multiple clusters such that actors in the same cluster are executed sequentially by a single software thread.
(2) *Ordering*. Ordering actor firings inside each cluster, while maintaining SDF consistency.
(3) *Buffering*. Computing buffer sizes for edges inside and across clusters. In dataflow semantics, edges generally represent infinite FIFO buffers, but for practical implementation, it is useful to impose bounds on buffer sizes.
(4) *Assignment*. Creating certain numbers of threads, and assigning clusters to threads for concurrent execution, under the constraint that each cluster can only be executed by one software thread at any given time.
(5) *Synchronization*. Determining when a cluster is executed by a software thread, and synchronizing between multiple concurrent threads such that all data precedence and buffer bound constraints are satisfied.

In this article, we develop the *Multithreaded Simulation Scheduler* (MSS) to systematically exploit multithreading capabilities when simulating SDF-based designs. The compile-time scheduling framework in MSS strategically integrates graph clustering, actor vectorization, intracluster scheduling, and intercluster buffering techniques to jointly perform static clustering, static ordering, and static buffering for trading off among throughput, synchronization overhead, and buffer requirements. From this compile-time framework, *InterThread Communication* (ITC) SDF graphs are constructed for multithreaded execution. The runtime scheduling in MSS then applies a self-scheduled multithreaded execution model to schedule and synchronize multiple software threads for executing ITC graphs at runtime.

The organization of this article is as follows: We review dataflow background in Section 2 and related work in Section 3. In Section 4, we present Ω-scheduling, the theoretical foundation of MSS. We then introduce our compile-time scheduling framework in Section 5, and our runtime scheduling approach in Section 6. In Section 7, we demonstrate simulation results, and we conclude in the final section.

2. BACKGROUND

In the dataflow modeling paradigm, the computational behavior of a system is represented as a directed graph $G = (V, E)$. A vertex (*actor*) $v \in V$ represents a computational module or a hierarchically nested subgraph. A directed edge $e \in E$ represents a

FIFO buffer from its source actor $src(e)$ to its sink actor $snk(e)$, and imposes precedence constraints for proper scheduling of the dataflow graph. A dataflow edge e can have a nonnegative integer *delay* $del(e)$ associated with it. This delay value specifies the number of initial data values (*tokens*) that are buffered on the edge before the graph starts execution.

Dataflow graphs operate based on *data-driven* execution: an actor v can execute (*fire*) only when it has sufficient numbers of data values (tokens) on all of its input edges $in(v)$. When firing, v consumes certain numbers of tokens from its input edges, executes its computation, and produces certain numbers of tokens on its output edges $out(v)$. In SDF, the number of tokens produced onto (consumed from) an edge e by a firing of $src(e)$ ($snk(e)$) is restricted to be a constant positive integer that must be known at compile time; this integer is referred to as the *production rate* (*consumption rate*) of e and is denoted as $prd(e)$ ($cns(e)$). We say that an edge e is a *single-rate* edge if $prd(e) = cns(e)$, and a *multirate* edge if $prd(e) \neq cns(e)$.

Before execution, a *schedule* of a dataflow graph is determined. Here, by a schedule, we mean any static or dynamic mechanism for executing actors. An SDF graph $G = (V, E)$ has a valid schedule (is *consistent*) if it is *free from deadlock* and is sample rate consistent, that is, it has a *periodic schedule* that fires each actor at least once and produces no net change in the number of tokens on each edge [Lee and Messerschmitt 1987]. In more precise terms, G is *sample rate consistent* if there is a positive integer solution to the *balance equations*.

$$\forall e \in E,\ prd(e) \times \boldsymbol{x}[src(e)] = cns(e) \times \boldsymbol{x}[snk(e)] \tag{1}$$

When it exists, the minimum positive integer solution for the vector $\boldsymbol{x}$ is called the *repetitions vector* of G, and is denoted by $\boldsymbol{q}_G$. For each actor v, $\boldsymbol{q}_G[v]$ is referred to as the *repetition count* of v. A *valid minimal periodic schedule* (which is abbreviated as *schedule* hereafter in this article) is then a sequence of actor firings in which each actor v is fired $\boldsymbol{q}_G[v]$ times, and the firing sequence obeys the data-driven properties imposed by the SDF graph.

SDF clustering is an important operation in scheduling SDF graphs [Bhattacharyya et al. 1996]. Given a connected, consistent SDF graph $G = (V, E)$, clustering a connected subset $Z \subseteq V$ into a *supernode* α means: (1) extracting a subgraph $G_\alpha = (Z, \{e \mid src(e) \in Z \text{ and } snk(e) \in Z\})$; and (2) transforming G into a reduced form $G' = (V', E')$, where $E' = E - \{e \mid src(e) \in Z \text{ or } snk(e) \in Z\} + E^*$ and $V' = V - Z + \{\alpha\}$. Here, E^* is a set of "modified" edges in G that originally connect actors in Z to actors outside of Z. More specifically, for every edge $e \in E$ that satisfies ($src(e) \in Z$ and $snk(e) \notin Z$), there is a modified version $e^* \in E^*$ such that $src(e^*) = \alpha$ and $prd(e^*) = prd(e) \times \boldsymbol{q}_{G_\alpha}(src(e))$, and similarly, for every $e \in E$ that satisfies ($src(e) \notin Z$ and $snk(e) \in Z$), there is a modified version $e^* \in E^*$ such that $snk(e^*) = \alpha$ and $cns(e^*) = cns(e) \times \boldsymbol{q}_{G_\alpha}(snk(e))$. In the transformed graph G', execution of α corresponds to executing one iteration of a minimal periodic schedule for G_α. SDF clustering guides the scheduling process by transforming G into a reduced form G' and isolating a subgraph G_α of G such that G' and G_α can be treated separately, for example, by using different optimization techniques.

Homogeneous Synchronous DataFlow (HSDF) is a restricted form of SDF in which every actor produces and consumes only one token from each of its input and output edges in a firing. HSDF is widely used in throughput analysis and multiprocessor scheduling. Any consistent SDF graph can be converted to an equivalent HSDF graph based on the *SDF-to-HSDF transformation* [Lee and Messerschmitt 1987; Sriram and Bhattacharyya 2009] such that samples produced and consumed by every invocation of each actor in the HSDF graph remain identical to those in the original SDF graph.

Let Z^+ denote the set of positive integers. Given an HSDF graph $G = (V, E)$, we denote the execution time of an actor v by $t(v)$, and denote $f_T : V \rightarrow Z^+$ as an actor execution time function that assigns $t(v)$ to a finite positive integer for every $v \in V$ (the actual execution time $t(v)$ can be interpreted as cycles of a base clock).

The *Cycle Mean* (CM) of a cycle c in an HSDF graph is defined as

$$CM(c) = \frac{\sum_{v \text{ in } c} t(v)}{\sum_{e \text{ in } c} del(e)}. \tag{2}$$

The *Maximum Cycle Mean* (MCM) of an HSDF graph G is defined as

$$MCM(G) = \max_{\text{cycle } c \text{ in } G} CM(c). \tag{3}$$

According to Reiter [1968], given a strongly connected HSDF graph G, when actors execute as soon as data is available at all inputs, the iteration period is $MCM(G)$, and the maximum achievable throughput is $1/MCM(G)$.

3. RELATED WORK

A preliminary version of this work was published in Hsu et al. [2008]. This preliminary version focused on covering the major ideas, intuitive descriptions, and results of the MSS. This extended new version goes beyond the preliminary version by including extensive theoretical derivations and precise algorithm formulations to provide a rigorous foundation for the MSS.

Various scheduling algorithms and techniques have been developed for different applications of SDF graphs. For software synthesis onto embedded single-processor implementations, Bhattacharyya et al. [1996] and Ko et al. [2004] have developed algorithms for joint code and memory minimization for certain types of SDF graphs; and Murthy and Bhattacharyya [2006] have developed various methods to share memory spaces across multiple buffers. Some of these algorithms and methods are implemented in Hsu et al. [2005] for synthesis of embedded DSP software.

For simulation of SDF graphs in single-processor environments, the cluster-loop scheduler has been developed based on Buck [1993] in the Ptolemy environment [Buck et al. 1994] as a fast heuristic. This approach recursively encapsulates adjacent groups of actors into loops to enable possible data rate matches and then clusters the adjacent groups. However, this approach suffers large runtimes and buffer requirements in heavily multirate systems [Hsu et al. 2006]. We have developed the *Simulation-Oriented Scheduler* (SOS) in Hsu et al. [2006] that integrates several techniques for graph decomposition and SDF scheduling to provide effective, joint minimization of time and memory requirements for simulating highly multirate SDF graphs.

Heuristics for minimum buffer scheduling have been developed in, for instance, Bhattacharyya et al. [1996] and Cubric and Panangaden [1993]. Various methods have been developed to compute minimum buffer requirements for SDF graphs, for example, Ade et al. [1997] and Geilen et al. [2005], to analyze throughput in SDF graphs, for example, Sriram and Bhattacharyya [2009] and Ghamarian et al. [2006], and to explore trade-offs between buffer sizes and throughput, for example, Govindarajan et al. [2002] and Stuijk et al. [2006]. However, the complexities of these approaches are not polynomially bounded in the graph size.

Multiprocessor scheduling for HSDF and related models has been extensively studied in the literature, for instance, see Kim and Browne [1988], Sarkar [1989], Sriram and Bhattacharyya [2009], and Kianzad and Bhattacharyya [2006]. Sarkar [1989] presented partitioning and scheduling heuristics that apply bottom-up clustering of tasks to trade off communication overhead and parallelism. Sriram and

Bhattacharyya [2009] reviewed an abundant set of scheduling and synchronization techniques for embedded multiprocessors, including various techniques for interprocessor communication conscious scheduling, the ordered transactions strategy, and synchronization optimization in self-timed systems.

Task-level vectorization, or block processing, is a useful dataflow graph transformation that can significantly improve execution performance by allowing subsequences of data items to be processed through individual task invocations. Block processing has been studied in single-processor software synthesis in various previous efforts, for example, Ritz et al. [1993], Lalgudi et al. [2000], and Ko et al. [2006]. In contrast to these efforts, we focus here on actor vectorization techniques that are suited to multithreaded implementation contexts.

For SDF scheduling that is specific to multithreaded simulation, Kin and Pino [2003] developed a scheduler, called the *thread cluster scheduler*, in Agilent ADS. This approach applies recursive two-way partitioning on single-processor schedules that are derived from the cluster loop scheduler and then executes the nested two-way clusters with multiple threads in a pipelined fashion. Experimental results in Kin and Pino [2003] show an average of 2 times speedup on a four-processor machine. However, according to our recent experiments, in which we used the same scheduler to simulate several wireless designs, this approach does not scale well to simulating highly multirate SDF graphs. More details are discussed in Section 7.

4. Ω-SCHEDULING

As described in Section 1, the problem of scheduling SDF graphs for multithreaded execution is highly complex. Our first step is to assume unbounded processing resources. In this case, the scheduling tasks of clustering, ordering, and assignment become trivial because the best strategy is to assign each actor exclusively to a processor. Then the problem can be simplified as follows: given unbounded processing resources, how do we schedule (including buffering and synchronization) SDF graphs to achieve maximal throughput? In this section, we develop a set of theorems and algorithms to solve this problem.

4.1 Definitions and Methods for Throughput Analysis

For throughput analysis, we denote the number of tokens (the *state*) on an edge e at time t by $tok(e, t)$. We assume actor firing is an *atomic* operation, and define that the *state transition*, that is, the change in the number of tokens, on an edge e happens immediately when either $src(e)$ or $snk(e)$ finishes its firing (execution).

In dataflow-based design tools, actors may have internal state that prevents executing multiple invocations of the actors in parallel. A delay line implemented within an FIR filter is an example of such internal state. Furthermore, whether or not an actor has internal state may be a lower-level detail in the actor's implementation that is not visible to the tool (e.g., to algorithms that operate on the dataflow graph). This is, for example, the case in Agilent ADS, the specific design tool that provides the context for our study and the platform for our experiments. Thus, exploring data-level parallelism by duplicating actors onto multiprocessors, for example, as explored in Saha et al. [2006], is out of the scope of this article.

In pure dataflow semantics, data-driven execution simply assumes infinite edge buffers. For practical implementations, it is necessary to impose bounds on buffer sizes. Given an SDF graph $G = (V, E)$, we denote the buffer size of an edge $e \in E$ (i.e., the bound on the size of a FIFO buffer or the size of a circular buffer [Bhattacharyya and Lee 1994] or other types of buffer implementations) by $buf(e)$, and denote $f_B : E \rightarrow Z^+$ as a buffer size function that assigns $buf(e)$ to a finite positive integer

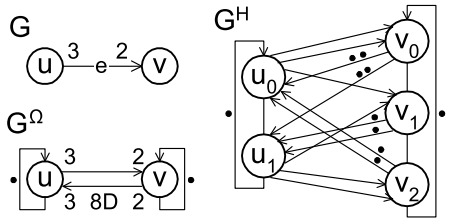

Fig. 1. An SDF graph G, the Ω-SDF graph G^{Ω} given $buf(e) = 8$, and the transformed Ω-HSDF graph G^{H}.

for every $e \in E$. In the following definition, we refine the notion of *fireability* for a dataflow actor to take bounded buffers into account.

Definition 4.1. Given an SDF graph $G = (V, E)$ and a buffer size function f_B, an actor $v \in V$ is (*data-driven*) *bounded-buffer fireable* at time t: if (1) v is fireable, that is, v has sufficient numbers of tokens on all of its input edges (data-driven property): $\forall e \in in(v)$, $tok(e, t) \geq cns(e)$, and (2) v has sufficient numbers of "empty spaces" on all of its output edges (bounded-buffer property): $\forall e \in out(v)$, $buf(e) - tok(e, t) \geq prd(e)$.

Recall that the task of synchronization is to maintain data precedence and bounded-buffer constraints. As a result, an intuitive scheduling strategy for maximal throughput is to fire an actor as soon as it is bounded-buffer fireable. We define such a scheduling strategy as follows, where actors are synchronized by bounded-buffer fireability.

Definition 4.2. Given a consistent SDF graph $G = (V, E)$, Ω-*scheduling* is defined as the SDF scheduling strategy that: (1) statically assigns each actor $v \in V$ to a separate processing unit, (2) statically determines a buffer bound $buf(e)$ for each edge $e \in E$, and (3) fires an actor as soon as it is bounded-buffer fireable.

In the following definitions, we define the concepts of the Ω-SDF graph and the Ω-HSDF graph, which are important to throughput analysis for Ω-scheduling.

Definition 4.3. Given an SDF graph $G = (V, E)$ and a buffer size function f_B, the Ω-*SDF graph* of G is defined as $G^{\Omega} = (V, \{E + E_b^{\Omega} + E_s^{\Omega})\}$. Here, E_s^{Ω} is the set of self-loops that models the sequential execution constraint for multiple firings of the same actor (on a single processing unit), that is, for each actor $v \in V$, there is an edge $e_s \in E_s^{\Omega}$ such that $src(e_s) = v$, $snk(e_s) = v$, $prd(e_s) = 1$, $cns(e_s) = 1$, $del(e_s) = 1$. E_b^{Ω} is the set of edges that models the bounded-buffer constraint in Ω-scheduling, that is, for each edge $e \in E$, there is a corresponding edge $e_b \in E_b^{\Omega}$ such that $src(e_b) = snk(e)$, $snk(e_b) = src(e)$, $prd(e_b) = cns(e)$, $cns(e_b) = prd(e)$, $del(e_b) = buf(e) - del(e)$.

Definition 4.4. Given an SDF graph $G = (V, E)$ and a buffer size function f_B, the Ω-*HSDF graph* G^H of G is defined as the HSDF graph that is transformed from the Ω-SDF graph G^{Ω} based on the SDF-to-HSDF transformation (as described in Section 2).

Figure 1 presents an SDF graph G, the Ω-SDF graph G^{Ω} given $buf(e) = 8$, and the corresponding Ω-HSDF graph G^H. Next, we analyze the throughput upper bound for Ω-scheduling in the following theorems.

THEOREM 4.5 [HSU 2007]. *Suppose that we are given a consistent SDF graph G, a buffer size function f_B, and an actor execution time function f_T. Then the maximum achievable throughput in Ω-scheduling is the inverse of the maximum cycle mean of the corresponding Ω-HSDF graph G^H —*

$$\frac{1}{MCM(G^H)}. \tag{4}$$

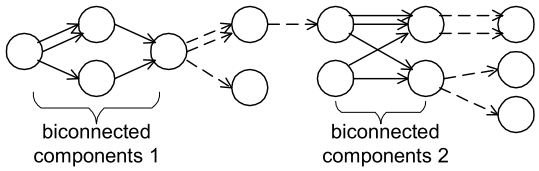

Fig. 2. Biconnected components and bridges.

THEOREM 4.6 [HSU 2007]. *Suppose that we are given a consistent, acyclic SDF graph* $G = (V, E)$ *and an actor execution time function* f_T. *Then the maximum achievable throughput in* Ω*-scheduling is*

$$\frac{1}{\max_{v \in V}(\boldsymbol{q}_G[v] \times t(v))}. \tag{5}$$

THEOREM 4.7 [HSU 2007]. *Given a consistent SDF graph* $G = (V, E)$ *and an actor execution time function* f_T, *Eq.* (5) *is a throughput upper bound in* Ω*-scheduling, that is, the maximum achievable throughput is less than or equal to that given by Eq.* (5).

In summary, Eq. (5) is an upper bound on throughput for a consistent SDF graph in Ω-scheduling, and is the maximum achievable throughput for a consistent, acyclic SDF graph in Ω-scheduling.

4.2 Buffering

Many existing techniques for joint buffer and throughput analyses rely on prior knowledge of actor execution times. However, exact actor execution time information may be unavailable in practical situations. In this article, we focus on minimizing buffer requirements under the maximum achievable throughput in Ω-scheduling without prior knowledge of actor execution times.

First of all, it is useful to employ the notion of parallel edge sets and biconnected components.

Definition 4.8. Given a graph $G = (V, E)$, a *parallel edge set* $[u, v]$ is defined as a set of edges $\{e \in E | src(e) = u \text{ and } snk(e) = v\}$ that connect from the same source vertex $u \in V$ to the same sink vertex $v \in V$.

Definition 4.9. Given a connected graph $G = (V, E)$, a *biconnected component* is a maximal set of parallel edge sets $A \subseteq E$ such that any pair of parallel edge sets in A lies in a simple undirected cycle. A *bridge* is then a parallel edge set that does not belong to any biconnected component, or equivalently, a parallel edge set whose removal disconnects G.

Figure 2 shows biconnected components and bridges of an example graph, where bridges are marked with dashed lines. In the following theorem, we first provide minimum buffer analysis for two-actor SDF graphs.

THEOREM 4.10 [HSU 2007]. *Given a consistent, acyclic, two-actor SDF graph* $G = (\{u, v\}, [u, v])$, *the minimum buffer size to sustain the maximum achievable throughput in* Ω*-scheduling over any actor execution time function is given by Eq.* (6).

$$\forall e_i \in [u, v], buf(e_i) = \begin{cases} (p_i + c_i - g_i) \times 2 + d_i - d^* \times g_i \,, \\ \quad \text{if } 0 \le d^* \le (p^* + c^* - 1) \times 2 \\ d_i, \text{ otherwise} \end{cases} \tag{6}$$

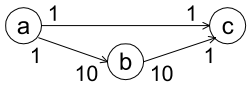

Fig. 3. Buffering deadlock example.

```
Ω-ACYCLIC-BUFFERING(G)
input: a consistent acyclic SDF graph G = (V, E)
 1  E_B = BRIDGES(G)
 2  for each [u, v] ∈ E_B  compute buffer sizes by Equation (6)  end
 3  {E_1, E_2, ..., E_N} = BICONNECTED-COMPONENTS(G)
 4  for each biconnected subgraph G_i = (V_i, E_i) from i = 1 to N
 5     {V_i^1, V_i^2, ..., V_i^M} = BICONNECTED-FREE-PARTITION(G_i)
 6     G'_i = (V'_i, E'_i) = CLUSTER(G_i, {V_i^1, V_i^2, ..., V_i^M})
 7     compute buffer sizes for E'_i by Equation (7) on G'_i
 8     for each partitioned subgraph G_i^j = (V_i^j, E_i^j) from j = 1 to M
 9        Ω-ACYCLIC-BUFFERING(G_i^j)
10     end
11  end
```

Fig. 4. Ω-acyclic-buffering algorithm.

Here, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, $c^* = c_i/g_i$, *and* $d^* = \min_{e_i \in [u,v]} \lfloor d_i/g_i \rfloor$.

In the following theorem, we generalize Theorem 4.10 to acyclic SDF graphs that do not contain biconnected components.

THEOREM 4.11 [HSU 2007]. *Suppose that we are given a consistent SDF graph* $G = (V, E)$*, and suppose that* G *does not contain any biconnected components. Then the minimum buffer sizes to sustain the maximum achievable throughput in* Ω*-scheduling over any actor execution time function are given by Eq.* (7)*.*

$$\forall [u,v] \in E, \forall e_i \in [u,v], buf(e_i) = \begin{cases} (p_i + c_i - g_i) \times 2 + d_i - d^* \times g_i \,, \\ \quad \text{if } 0 \le d^* \le (p^* + c^* - 1) \times 2 \\ d_i, \text{ otherwise} \end{cases} \tag{7}$$

Here, for each parallel edge set $[u, v] \in E$*, and for each edge* $e_i \in [u, v]$, $p_i = prd(e_i)$, $c_i = cns(e_i)$, $d_i = del(e_i)$, $g_i = \gcd(p_i, c_i)$, $p^* = p_i/g_i$, $c^* = c_i/g_i$, *and* $d^* = \min_{e_i \in [u,v]} \lfloor d_i/g_i \rfloor$.

Applying Eq. (7) to general acyclic SDF graphs may cause deadlock in Ω-scheduling. Figure 3 presents such an example: if the buffer size for edge (a, c) is set to 2, then the graph is deadlocked because neither b nor c can fire due to insufficient buffer size on (a, c). In order to allocate buffers for general acyclic SDF graphs in Ω-scheduling, we have developed the Ω-*acyclic-buffering* algorithm as shown in Figure 4, and we prove the validity of the algorithm in Theorem 4.12.

In Figure 4, we compute bridges E_B of G and set buffer sizes of each parallel edge set in E_B by Eq. (6) in lines 1–2. Next, we compute the biconnected components $E_1, E_2, \ldots, E_N$ of G in line 3. For each biconnected subgraph $G_i = (V_i, E_i)$ induced from the biconnected component E_i (where V_i is the set of source and sink actors of edges in E_i) for $i \in \{1, 2, \ldots, N\}$, we first compute a *biconnected-free partition* $V_i^1, V_i^2, \ldots, V_i^M$ of V_i in line 5 such that clustering $V_i^1, V_i^2, \ldots, V_i^M$ in G_i in line 6 does not introduce any biconnected component in the clustered version $G'_i = (V'_i, E'_i)$ of G_i. After that, we apply Eq. (7) on G'_i to compute buffer sizes for E'_i. Then in line 8, we apply the

Ω-acyclic-buffering algorithm *recursively* to each acyclic subgraph $G_i^j = (V_i^j, E_i^j)$ that is induced from the partition V_i^j for $j \in \{1, 2, \ldots, M\}$.

Note that a biconnected-free partition of an acyclic biconnected subgraph always exists for any *2-way* partition based on a topological sort. For efficiency, our approach simply computes a topological sort for each biconnected subgraph and chooses the best 2-way cut that results in least buffer requirements for cross edges (E_i'). With efficient data structures, the operations in Ω-acyclic-buffering can be implemented in linear time (i.e., in time that is linear in the number of nodes and edges in the graph) [Hsu 2007].

THEOREM 4.12 [HSU 2007]. *Given a consistent, acyclic SDF graph $G = (V, E)$, the Ω-acyclic-buffering algorithm gives buffer sizes that sustain the maximum achievable throughput in Ω-scheduling over any actor execution time function.*

5. COMPILE-TIME SCHEDULING FRAMEWORK

In this section, we develop compile-time scheduling techniques (including techniques for clustering, ordering, and buffering) based on the Ω-scheduling concept to construct InterThread Communication (ITC) SDF graphs for multithreaded execution.

5.1 Clustering and Actor Vectorization

The simplest way to imitate Ω-scheduling in multithreaded environments is to execute each actor by a separate thread and block actor execution until it is bounded-buffer fireable. However, threads share processing resources, and the available resources on a multicore processor is limited. As a result, threads are competing for processing resources for both execution and synchronization, that is, checking bounded-buffer fireability. Since the ideal situation is to spend all processing time in actor execution, minimizing synchronization overhead becomes a key factor. In Ω-scheduling, synchronization overhead increases with the repetitions vector of the SDF graph because bounded-buffer fireability must be maintained for every actor firing. Here, we use $Q_G = \sum_{v \in V} \boldsymbol{q}_G[v]$ to represent the synchronization overhead associated with a consistent SDF graph $G = (V, E)$ in Ω-scheduling.

Clustering combined with static intracluster scheduling is one of our strategies to reduce synchronization overhead. After clustering partitions of nodes, each cluster is subject to single-thread execution, and the schedule of each cluster (subgraph) is statically computed. Indeed, SDF clustering [Bhattacharyya et al. 1996], as introduced in Section 2, guarantees that firing a clustered node is equivalent to executing one iteration of a minimal periodic schedule for the subgraph. We formalize this scheduling strategy in Definition 5.1 and Definition 5.2. The strategy is defined in a general way such that each cluster is assigned to a processing unit (instead of to a thread specifically) and is applicable to scheduling SDF graphs for resource-constrained multiprocessors by controlling the number of partitions.

Definition 5.1. Given a consistent SDF graph $G = (V, E)$, a *consistent partition* P of G is a partition $Z_1, Z_2, \ldots, Z_{|P|}$ of V such that the SDF graph G_P resulting from clustering $Z_1, Z_2, \ldots, Z_{|P|}$ in G is consistent.

Definition 5.2. Given a consistent SDF graph $G = (V, E)$, Π-*scheduling* is defined as the SDF scheduling strategy that: (1) transforms G into a smaller consistent SDF graph $G_P = (V_P = \{v_1, v_2, \ldots, v_{|P|}\}, E_P)$ by clustering a consistent partition $P = Z_1, Z_2, \ldots, Z_{|P|}$ of G; (2) statically computes a *minimal periodic schedule* S_i for each subgraph $G_i = (Z_i, E_i = \{e \in E | src(e) \in Z_i \text{ and } snk(e) \in Z_i\})$ such that execution

of supernode $v_i \in V_P$ corresponds to executing one iteration of S_i; and (3) applies Ω-scheduling on G_P.

Note that actors in a subset Z_i are not necessarily connected. For this reason, we extend the definition of SDF clustering [Bhattacharyya et al. 1996] to allow clustering a disconnected subset $Z_i \subset V$ by adding the following provision: if $G_i = (Z_i, E_i = \{e \in E | src(e) \in Z_i \text{ and } snk(e) \in Z_i\})$ is disconnected, $\boldsymbol{q}_{G_i}[v]$ is defined as $\boldsymbol{q}_G[v]/\text{gcd}_{z \in Z_i} \boldsymbol{q}_G[z]$ for each actor $v \in Z_i$.

After clustering a subset Z_i into a supernode v_i, the repetition count of v_i in G_P becomes $\boldsymbol{q}_{G_P}[v_i] = \text{gcd}_{v \in Z_i} \boldsymbol{q}_G[v]$. With a well-designed clustering algorithm, clustering can significantly reduce synchronization overhead from the range of $Q_G = \sum_{v \in V} \boldsymbol{q}_G[v]$ down to the range of $Q_{G_P} = \sum_{v \in V_P} \boldsymbol{q}_{G_P}[v]$, but at the expense of buffer requirements and throughput. Clustering may increase buffer requirements because the interface production and consumption rates of the resulting supernodes are multiplied in order to preserve multirate consistency [Bhattacharyya et al. 1996]. Clustering also decreases throughput due to less parallelism. In the following theorem, we analyze the effect of clustering on the throughput of Ω-scheduling, assuming negligible runtime overhead in executing static schedules and determining bounded-buffer fireability.

THEOREM 5.3 [HSU 2007]. *Suppose that we are given a consistent SDF graph $G = (V, E)$, a buffer size function f_B, and an actor execution time function f_T. Suppose also that $P = Z_1, Z_2, \ldots, Z_{|P|}$ is a consistent partition of G and $G_P = (V_P = \{v_1, v_2, \ldots, v_{|P|}\}, E_P)$ is the SDF graph resulting from clustering P. Then a throughput upper bound for G in Π-scheduling, or equivalently, a throughput upper bound for G_P in Ω-scheduling is*

$$\frac{1}{\max_{Z_i \in P}(\sum_{v \in Z_i}(\boldsymbol{q}_G[v] \times t(v)))}. \tag{8}$$

Moreover, if G_P is acyclic, Eq. (8) *gives the maximum achievable throughput.*

Given a consistent SDF graph $G = (V, E)$, Theorem 5.3 tells us that for clustering a set of actors $Z_i \subseteq V$ into a supernode v_i, a metric that significantly affects the overall throughput is the sum of the repetition count (in terms of G) – execution time products among all actors $v \in Z_i$. For convenience, we denote this value by *SRTP* and define $SRTP(v_i) = SRTP(Z_i) = \sum_{v \in Z_i}(\boldsymbol{q}_G[v] \times t(v))$. Based on Theorem 5.3, the cluster with the largest SRTP value dominates the overall throughput.

In single-processor environments (single-processing unit), the ideal iteration period for executing a consistent SDF graph $G = (V, E)$ is $SRTP(G) = \sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$, that is, the time to execute one iteration of a minimal periodic schedule of G. Now considering an N-core processor (N-processing units), the ideal speedup over a single-processor is N. In other words, the ideal iteration period on an N-core processor is $\sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))/N$, and equivalently, the ideal throughput is $N/\sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$. In the clustering process, by imposing Eq. (9) as the constraint for each cluster (partition) Z_i, the ideal N-fold speedup can be achieved theoretically when the SRTP threshold parameter M in Eq. (9) is greater than or equal to N.

$$\sum_{v \in Z_i}(\boldsymbol{q}_G[v] \times t(v)) \leq \sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))/M \tag{9}$$

In practice, exact actor execution time is in general unavailable, and execution time estimates may cause large differences between compile-time and runtime SRTP

values. As a result, the SRTP threshold parameter M is usually set larger than N in order to tolerate unbalanced runtime SRTP values, that is, by having more small (in terms of compile-time SRTP value) clusters and using multiple threads to share processing units. Based on our experiments, when $N = 4$, the best M is usually between 16 and 32 depending on the graph size and other factors.

Actor vectorization (*actor looping*) is our second strategy to reduce synchronization overhead. Previous work related to actor vectorization in other contexts is discussed in Section 3. The main idea in our approach to actor vectorization is to vectorize (loop together) actor executions by a factor of the repetition count of the associated actor. We define actor vectorization as follows.

Definition 5.4. Given a consistent SDF graph $G = (V, E)$, *vectorizing* (*looping*) an actor $v \in V$ by a factor k of $\boldsymbol{q}_G[v]$ means: (1) replacing v by a vectorized actor v^k such that a firing of v^k corresponds to executing v consecutively k times; and (2) replacing each edge $e \in in(v)$ by an edge $e' \in in(v^k)$ such that $cns(e') = cns(e) \times k$, and replacing each edge $e \in out(v)$ by an edge $e' \in out(v^k)$ such that $prd(e') = prd(e) \times k$. For consistency, the *vectorization factor* must be a factor of the repetition count of v. After vectorization, $t(v^k) = t(v) \times k$ and $\boldsymbol{q}_G[v^k] = \boldsymbol{q}_G[v]/k$.

In practical highly multirate SDF graphs, repetitions vectors usually consist of large, nonprime numbers [Hsu et al. 2006]. As a result, actor vectorization is suitable for synchronization reduction in this context, but at the possible expense of larger latency (due to delaying the availability of output tokens) and larger buffer requirements (due to the multiplication of production and consumption rates).

5.2 Overview of Compile-Time Scheduling Framework

In our *Multithreaded Simulation Scheduler* (MSS), we have developed a compile-time scheduling framework that jointly performs the clustering, ordering, and buffering tasks as described in Section 1 at compile time. In this framework, we strategically integrate several graph clustering and actor vectorization algorithms in a bottom-up fashion such that each subsequent algorithm works on the clustered/vectorized version of the graph from the preceding algorithm. This architecture is presented in Figure 5. We also incorporate into this framework intracluster scheduling techniques (which include ordering and buffering) as we developed in the Simulation-Oriented Scheduler (SOS) [Hsu et al. 2007] such that static intracluster schedules (as well as buffer sizes) can be computed along the way in the bottom-up clustering process. Finally, we apply the intercluster buffering techniques as presented in Section 4.2 to compute buffer sizes for the top-level graph to achieve the maximum achievable throughput theoretically in Ω-scheduling.

Given a consistent SDF graph $v \in V$ as input to the compile-time scheduling framework, the resulting graph $G_{itc} = (V_{itc}, E_{itc})$ is called an *InterThread Communication* (ITC) SDF graph (or simply *ITC graph*) because each node (cluster) in G_{itc} is executed by a thread. The ITC graph is then passed to the runtime scheduling part of MSS for multithreaded execution (see Section 6). In MSS, ITC graphs are carefully constructed from input SDF graphs for proper trade-offs among the following three related metrics: (1) synchronization overhead, (2) throughput, and (3) buffer requirements.

Simulation tools usually run on workstations and high-end PCs where memory resources are abundant. However, without careful design, clustering and actor vectorization may still run out of memory due to large multirate complexity [Hsu et al. 2006]. In our approach, total buffer requirements are managed within the given upper bound. A proper *buffer memory upper bound* can be derived from the available memory resources in the simulation environment and other relevant considerations.

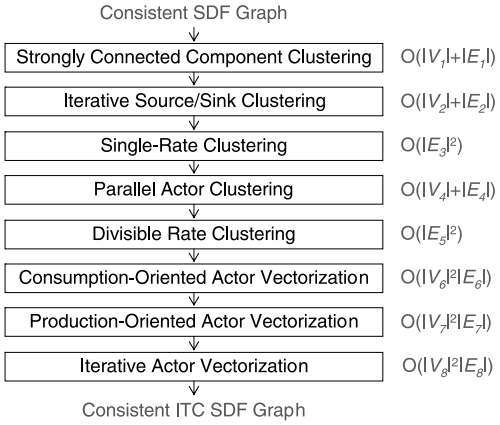

Fig. 5. Compile-time scheduling framework in MSS.

In this framework, all of the integrated algorithms emphasize low complexity for minimizing the time spent in compile-time scheduling. In addition, clustering algorithms are carefully designed to prevent introduction of cycles in the clustered version of the graph. This is because cycles may cause deadlock due to cyclic data dependence. Furthermore, even without deadlock, cycles may cause limitations in the maximum achievable throughput. The following theorem provides a precise condition for the introduction of a cycle by a clustering operation.

THEOREM 5.5 [HSU ET AL. 2007]. *Given a connected, acyclic SDF graph $G = (V, E)$, clustering a subset $R \subseteq V$ introduces a cycle in the clustered version of G if and only if there is a path $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ ($n \geq 3$) in G such that $v_1 \in R$, $v_n \in R$, and $v_2, \ldots, v_{n-1} \in \{V - R\}$. Clustering R is cycle free if and only if no such a path exists.*

In the following subsections, we introduce our algorithms for clustering and actor vectorization. The complexity of an algorithm is represented in terms of the number of vertices $|V|$ and edges $|E|$ in the input graph $G = (V, E)$ for each individual algorithm (*not* the overall graph to the scheduling framework). Based on this, $|V|$ and $|E|$ (the input sizes for the various algorithms) get progressively smaller through the bottom-up clustering process. For complexity analysis, we make the assumption that every actor has a constant (limited) number of input and output edges, that is, $|V|$ and $|E|$ are within a similar range. This is a reasonable assumption because actors in simulation tools are usually predefined, and practical SDF graphs in communications and signal processing domains are usually sparse in their topology [Bhattacharyya et al. 1996].

5.3 Strongly Connected Component Clustering

According to Theorem 4.7, the existence of cycles in an ITC SDF graph may decrease the maximum achievable throughput depending on the locations and magnitudes of edge delays in those cycles. Moreover, the presence of cycles restricts application of many useful scheduling techniques in our framework. Clustering Strongly Connected Components[1] (SCCs) is a well-known technique to generate acyclic graphs [Cormen et al. 2001] in linear time.

[1]A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $Z \subseteq V$ such that for every pair of vertices u and v in Z, there is a path from u to v and a path from v to u.

Based on our analysis, if the SRTP value of each SCC satisfies Eq. (9), clustering SCCs does not cause limitations in the achievable throughput. In MSS, the resulting acyclic SDF graph is further processed by the subsequent algorithms, and each SCC subgraph is scheduled efficiently by the simulation-oriented scheduler [Hsu et al. 2007].

5.4 Iterative Source/Sink Clustering

In practical communication and signal processing systems, subsystems having the form of chain or tree structures arise frequently. Based on Theorem 5.5, clustering such subsystems at the source-end or sink-end does not introduce cycles because there is only one connection between the subsystems and the rest of the graph. In addition, if the SDF production and consumption rate (data rate) behavior involved in such a subsystem is successively divisible in certain ways, then clustering such a subsystem does not increase the production or consumption rates of the resulting supernode. The idea of *Iterative Source/Sink Clustering* (ISSC) is to jointly explore the chain or tree structures and the successively divisible rate behavior in a low-complexity manner such that clustering is always cycle free and does not increase the buffer requirements.

Here, we first define some notation that is useful to our development. Given a directed graph $G = (V, E)$, we say that a vertex $v \in V$ is a *source* if v does not have any input edges ($in(v) = \varnothing$), and is a *sink* if v does not have any output edges ($out(v) = \varnothing$). For an edge $e \in E$, we say that $src(e)$ is a *predecessor* of $snk(e)$, and $snk(e)$ is a *successor* of $src(e)$. For a vertex v, we denote all of v's predecessors by $pre(v)$, denote all of v's successors by $suc(v)$, and denote all of v's adjacent actors by $adj(v) = \{pre(v) + suc(v)\}$. We then define the ISSC technique as follows.

Definition 5.6. Given a consistent, acyclic SDF graph $G = (V, E)$, the *Iterative Source/Sink Clustering* (ISSC) technique *iteratively* clusters:

(1) a source actor u with its successor if (1-a) u has one and only one successor v, (1-b) $\boldsymbol{q}_G[u]$ is divisible by $\boldsymbol{q}_G[v]$, and (1-c) $SRTP(u) + SRTP(v)$ is less than or equal to the SRTP threshold; or
(2) a sink actor v with its predecessor if (2-a) v has one and only one predecessor u, (2-b) $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[u]$, and (2-c) $SRTP(u) + SRTP(v)$ is less than or equal to the SRTP threshold.

Clustering based on these conditions continues until no further clustering can be performed. After each iteration, G represents the clustered version of the graph that is subject to the next iteration.

Because of divisible data rates, each two-actor cluster constructed in ISSC iterations can be scheduled efficiently by *flat scheduling* [Bhattacharyya et al. 1996]. ISSC can be implemented in linear-time complexity [Hsu 2007].

5.5 Single-Rate Clustering

Single-rate subsystems (subsystem in which all actors execute at the same average rate) arise commonly in practical communication and signal processing designs, even within designs that are heavily multirate at a global level. Since clustering single-rate subsystems does not increase production and consumption rates at the interface of the resulting supernodes, we integrate the *Single-Rate Clustering* (SRC) technique [Hsu et al. 2007] in our framework.

Definition 5.7. Given a connected, consistent, acyclic SDF graph $G = (V, E)$, the *Single-Rate Clustering* (SRC) technique clusters disjoint subsets $R_1, R_2, \ldots, R_N \subseteq V$ such that: (1) in the subgraph $G_i = (R_i, E_i)$, we have $prd(e_i) = cns(e_i)$ for every edge

$e_i \in E_i = \{e \in E \mid src(e) \in R_i$ and $snk(e) \in R_i\}$; (2) the clustering of R_i does not introduce any cycles into the clustered version of G; and (3) the SRTP value of R_i is less than or equal to the SRTP threshold.

According to Hsu et al. [2007], SRC has $O(|E|^2)$ complexity. Due to their simple structure, single-rate subgraphs (clusters) can be statically scheduled and optimized effectively by flat scheduling [Bhattacharyya et al. 1996] in linear time.

5.6 Parallel Actor Clustering

Subsets of *parallel actors* often exist in practical communication and signal processing systems. Here, we say actors u and v are *parallel* if there is no path from u to v nor from v to u. According to Theorem 5.5, clustering parallel actors is cycle free. In addition, based on our extended definition of SDF clustering, clustering parallel actors with the same repetition count does not increase the production and consumption rates of the resulting supernode. The idea of *parallel actor clustering* is to jointly explore both properties in a careful manner; this is because arbitrarily clustering parallel actors often introduces or expands biconnected components, and which may increase the overall buffer requirements in Ω-acyclic-buffering.

Here, we first present a topological ranking technique that is important to our development for exploring parallel structures with low complexity.

Definition 5.8. Given a directed acyclic graph $G = (V, E)$, *topological ranking* means to assign an integer value (*rank*) $r(v)$ to each vertex $v \in V$ such that: (1) for each edge $e \in E$, we have $r(snk(e)) > r(src(e))$, and (2) the difference between the largest and the smallest rank is minimal.

Property 5.9. Given a topological rank of a directed acyclic graph, vertices with the same rank are parallel.

Note that parallel actors may not have the same rank. The topological ranking technique is primarily developed as a linear-time approach to explore certain parallel structures in directed acyclic graphs. With the preceding technique, we present the parallel actor clustering technique as follows.

Definition 5.10. Given a consistent, acyclic SDF graph $G = (V, E)$ and a topological rank of G, the *Parallel Actor Clustering* (PAC) technique *iteratively* clusters a set of actors R ($|R| > 1$) that satisfy the following conditions until no further clustering can be made:

(1) all actors in R have the same rank;
(2) all actors in R have the same repetition count;
(3) [(all actors in R have the same predecessor v) and (the edges from v to R, $E_{v,R}$, belong to the same biconnected component, or $E_{v,R}$ are bridges)] or [(all actors in R have the same successor u) and (the edges from R to u, $E_{R,u}$, belong to the same biconnected component, or $E_{R,u}$ are bridges)]; and
(4) $SRTP(R)$ is less than or equal to the SRTP threshold.

Such an R is defined as a parallel actor subset. After each iteration, the resulting supernode inherits the same rank, and G represents the clustered version of the graph that is subject to the next iteration.

In Definition 5.10, condition 3 prevents introducing biconnected components. For a parallel actor subset, a static schedule can be constructed by firing each actor once in any order. The PAC technique can be implemented in linear time [Hsu 2007].

5.7 Divisible-Rate Clustering

In practical communication and signal processing systems, data rate behavior associated with an actor and its surrounding actors possesses certain valuable properties that can be explored in our clustering framework. In this section, we present the *divisible-rate clustering* technique to explore both single-rate and multirate behavior of an actor in relation to its adjacent actors such that buffer requirements can be maintained after clustering.

Definition 5.11. Given a consistent, acyclic SDF graph $G = (V, E)$, the *Divisible-Rate Clustering* (DRC) technique *iteratively* clusters an actor v with one of its candidate adjacent actors $u \in adj(v)$ such that:

(1) either (1-a) $\boldsymbol{q}_G[v] = \boldsymbol{q}_G[u]$ or (1-b) for every $x \in adj(v)$, $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[x]$, and $\boldsymbol{q}_G[u]$ is divisible by $\boldsymbol{q}_G[x]$;
(2) $SRTP(v) + SRTP(u)$ is less than or equal to the SRTP threshold; and
(3) clustering $\{v, u\}$ is cycle free.

This clustering process continues until no further clustering can be performed. After each iteration, G represents the clustered version of the graph that is subject to the next iteration.

In the divisible-rate clustering process, a candidate pair of adjacent actors $\{v, u\}$ is carefully chosen in each iteration. Suppose $\{v, u\}$ satisfies condition (1-a) in Definition 5.11, then clustering $\{v, u\}$ does not increase the interface data rates of the resulting supernode. Suppose $\{v, u\}$ satisfies condition (1-b) and suppose α represents the resulting supernode, then based on SDF clustering [Bhattacharyya et al. 1996], for an edge $e \in in(\alpha)$, if $cns(e)$ has been increased due to clustering, then $prd(e)$ is divisible by $cns(e)$, and for an edge $e \in out(\alpha)$, if $prd(e)$ has been increased, then $cns(e)$ is divisible by $prd(e)$. In many intracluster scheduling and intercluster buffering scenarios, this divisible-rate property maintains the buffer requirements of input and output edges for the resulting supernodes. In addition, the two-actor cluster $\{v, u\}$ can be scheduled efficiently by flat scheduling [Bhattacharyya et al. 1996] because $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[u]$. DRC can be performed in $O(|E|^2)$ complexity [Hsu 2007].

5.8 Consumption-/Production-Oriented Actor Vectorization

The techniques introduced from Section 5.4 to Section 5.7 are effective in clustering SDF graphs (for synchronization reduction) while maintaining the buffer requirements. The resulting SDF graphs in general have significantly smaller sizes and contain mixes of single-rate and multirate edges.

In the remainder of this section, we focus on actor vectorization techniques. Given a consistent, acyclic SDF graph $G = (V, E)$, the ideal situation for actor vectorization is to vectorize each actor $v \in V$ by its repetition count $\boldsymbol{q}_G[v]$, and have the total buffer requirement of the vectorized version of G remain within the given upper bound. Again, in our context, a proper buffer memory upper bound can be derived from the available memory resources and other relevant considerations. In this case, the resulting ITC graph is just a single-rate SDF graph, and the synchronization overhead is reduced to the range of $|V|$. However, due to large-scale and heavily multirate behavior involved in modern communication and signal processing systems, the ideal situation may not happen in general.

For this purpose, we develop actor vectorization techniques to strategically trade off buffer cost for synchronization reductions. In the vectorization process, the total buffer requirement is carefully kept under control within the given upper bound to prevent out-of-memory problems. Given a buffer computation function f_B for G, we

use $f_I(G, f_B, v \rightarrow v^k)$ to denote the increase in buffer requirements when vectorizing an actor v by a factor k.

By Definition 5.4, a vectorization factor must be a factor of the actor's repetition count in order to maintain graph consistency. However, heavily multirate systems often result in extremely high repetition counts, for instance, even up to the range of millions, as we show in Hsu et al. [2006]. As a result, the complexity to determine optimal vectorization factors is in general unmanageable in highly multirate systems. In this subsection, we use divisible multirate properties associated with an actor v and its adjacent actors to determine possible vectorization factors, that is, for an adjacent actor u of v, if $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[u]$, then $\boldsymbol{q}_G[v]/\boldsymbol{q}_G[u]$ is considered as a vectorization factor for v.

The idea of *Consumption-oriented Actor Vectorization* (CAV) technique is to take advantage of consumption-rate-divisible edges (i.e., $cns(e)$ is divisible by $prd(e)$) for actor vectorization and to explore single-rate clustering opportunities exposed by such actor vectorizations. CAV favors latency because the source actor of a consumption-rate-divisible edge is vectorized to match the rate of the sink actor. The design of CAV also prevents propagation of indivisible rates; without careful design, such propagation may cause larger buffer requirements and reduce opportunities for proper clustering. Here, we present the CAV technique as follows.

Definition 5.12. Suppose that we are given a consistent, acyclic SDF graph $G = (V, E)$, a buffer computation function f_B, and a buffer memory upper bound U. The *Consumption-oriented Actor Vectorization* (CAV) technique *iteratively* selects an actor v for vectorization and clustering until no further vectorization can be performed or the total buffer requirement approaches the upper bound U. An actor $v \in V$ is considered as a *candidate* for vectorization if:

(1) for every $x \in suc(v)$, $\boldsymbol{q}_G[v]$ is divisible by $\boldsymbol{q}_G[x]$;
(2) there exists an actor $u \in suc(v)$ such that for every $x \in suc(v)$, $\boldsymbol{q}_G[u]$ is divisible by $\boldsymbol{q}_G[x]$; and
(3) the buffer cost increase $f_I(G, f_B, v \rightarrow v^k)$ resulting from vectorizing v by a factor $k = \boldsymbol{q}_G[v]/\boldsymbol{q}_G[u]$ does not overflow the upper bound U.

In each iteration, CAV selects a candidate actor v whose repetition count $\boldsymbol{q}_G[v]$ is maximal over all candidates and then vectorizes v by the factor k. After vectorization, CAV *iteratively* clusters v with its adjacent actor $u \in adj(v)$ if:

(a) $\boldsymbol{q}_G[v] = \boldsymbol{q}_G[u]$ (single-rate);
(b) $SRTP(v) + SRTP(u)$ is less than or equal to the SRTP threshold; and
(c) clustering $\{v, u\}$ is cycle free.

After each actor vectorization and clustering iteration, G represents the vectorized or clustered version of the graph that is subject to the next iteration.

In Definition 5.12, conditions 1 and 2 prevent propagation of indivisible rates, and condition 3 prevents buffer overflow. After each clustering operation, the two-actor cluster $\{v, u\}$ can be scheduled efficiently by flat scheduling [Bhattacharyya et al. 1996] because $\boldsymbol{q}_G[v] = \boldsymbol{q}_G[u]$. CAV can be performed in $O(|V|^2|E|)$ time [Hsu 2007].

We also develop a similar *Production-oriented Actor Vectorization* (PAV) technique to explore production-rate-divisible edges (i.e., $prd(e)$ is divisible by $cns(e)$). In MSS, CAV is applied before PAV because buffer cost can be traded off for both synchronization overhead and latency by CAV.

5.9 Iterative Actor Vectorization

In this subsection, we present a general actor vectorization approach, called *Iterative Actor Vectorization* (IAV). This approach trades off buffer cost for synchronization cost, and handles both divisible and indivisible multirate interconnections.

According to Section 5.1, after vectorizing an actor $v \in V$ by a factor k of $\boldsymbol{q}_G[v]$, the amount of synchronization reduction can be represented by $\boldsymbol{q}_G[v](1 - 1/k)$. Then, a general strategy is to vectorize a properly chosen actor by a well-determined factor such that the synchronization reduction can be maximized while the penalty in buffer cost is minimal. Based on this observation, we define the *synchronization reduction to buffer increase ratio* (or simply *S/B ratio*) —

$$R_{S/B} : \frac{\boldsymbol{q}_G[v](1 - 1/k)}{f_I(G, f_B, v \rightarrow v^k)} \tag{10}$$

as the cost function for actor vectorization.

In IAV, for an adjacent actor u of v, if $\boldsymbol{q}_G[v] > \boldsymbol{q}_G[u]$, then $\boldsymbol{q}_G[v]/\gcd(\boldsymbol{q}_G[v], \boldsymbol{q}_G[u])$ is considered as a vectorization factor for v. Based on the preceding derivations, we develop the iterative actor vectorization technique as follows. This technique can be implemented in $O(|V|^2|E|)$ complexity [Hsu 2007].

Definition 5.13. Suppose that we are given a consistent, acyclic SDF graph $G = (V, E)$, a buffer computation function f_B, and a buffer memory upper bound U. The *Iterative Actor Vectorization* (IAV) technique *iteratively* vectorizes an actor v by a factor k of $\boldsymbol{q}_G[v]$ until no further vectorization can be performed or the total buffer requirement approaches the upper bound U. An actor $v \in V$ is considered for vectorization if v is a *local maximum* — that is,

(1) for every adjacent actor $u \in adj(v)$, $\boldsymbol{q}_G[v] \geq \boldsymbol{q}_G[u]$, and
(2) there exists at least one adjacent actor $u \in adj(v)$ such that $\boldsymbol{q}_G[v] > \boldsymbol{q}_G[u]$.

For such a local maximum actor v, the vectorization factor k is determined from the factors

$$\left\{ \frac{\boldsymbol{q}_G[v]}{\gcd(\boldsymbol{q}_G[v], \boldsymbol{q}_G[u])}, \forall u \in \{u \in adj(v) | \boldsymbol{q}_G[u] < \boldsymbol{q}_G[v]\} \right\} \tag{11}$$

such that the S/B ratio is maximized — that is, we maximize

$$R_{S/B} : \frac{\boldsymbol{q}_G[v](1 - 1/k)}{f_I(G, f_B, v \rightarrow v^k)}$$

subject to the constraint that the buffer cost increase $f_I(G, f_B, v \rightarrow v^k)$ does not overflow the upper bound U. In each iteration, a local maximum actor v is chosen such that the S/B ratio is maximized for v over all local maximum actors. After vectorization, IAV *iteratively* clusters v with its adjacent actor $u \in adj(v)$ if:

(a) $\boldsymbol{q}_G[v] = \boldsymbol{q}_G[u]$ (single-rate);
(b) $SRTP(v) + SRTP(u)$ is less than or equal to the SRTP threshold; and
(c) clustering $\{v, u\}$ is cycle free.

After each iteration, G represents the vectorized version of the graph that is subject to the next iteration.

6. RUNTIME SCHEDULING

In this section, we develop runtime scheduling techniques for the assignment and synchronization tasks in scheduling ITC graphs for multithreaded execution.

The simplest way to schedule an ITC graph $G_{itc} = (V_{itc}, E_{itc})$ for multithreaded execution is to allocate a number of threads equal to the number of ITC nodes $|V_{itc}|$ and assign each ITC node $v \in V_{itc}$ to a separate thread. Each thread executes the associated ITC node v as soon as v is bounded-buffer fireable and blocks otherwise. We refer to this approach as *self-timed multithreaded execution model*.

In multithreaded environments, multithread APIs and operating systems schedule the activities of threads and the usage of processing units. In the self-timed multithreaded execution model, the number of threads to be scheduled is equal to the number of nodes in an ITC graph, even though the processing units are very limited, for example, 2, 4, or 8 processing units in multicore processors that are used currently in typical desktop simulation environments. When the number of fireable ITC nodes is larger than the number of processing units, multithreading APIs and operating systems take responsibility for scheduling. Motivated by this observation, we develop the *self-scheduled multithreaded execution model* to provide an alternative method for executing ITC graphs in multithreaded environments.

Definition 6.1. Given a consistent ITC graph $G_{itc} = (V_{itc}, E_{itc})$, the self-scheduled multithreaded execution model allocates a number of threads equal to the number of processing units. Each thread dynamically selects and executes an ITC node $v \in V_{itc}$ that is bounded-buffer fireable and free for execution (i.e., v is not executed by other thread), and blocks when none of the ITC nodes is bounded-buffer fireable and free for execution.

This execution model performs dynamic assignment between ITC nodes and threads and synchronizes threads based on bounded-buffer fireability.

Figure 6 presents the *SELF-SCHEDULED-EXECUTION* function that is executed by each thread in the self-scheduled execution model. For a calling thread, the input graph G in Figure 6 is the ITC graph G_{itc}. Initially before calling this function, for each ITC node $v \in V_{itc}$, if v is bounded-buffer fireable, we push v onto a fireable list L and set v's state to *fireable*, otherwise, we set v's state to *not-fireable*. The input list L then contains the ITC nodes that are initially bounded-buffer fireable. Here, the state of an ITC node v is used to verify whether v is in L or whether v is under execution in constant time such that other concurrent threads do not mistakenly reinsert v into L (line 22).

Once execution control enters the *while* loop in line 3, we check whether there are ITC nodes in L. If the result is true, we pop the first ITC node v from L, and execute v for a number of times n that is determined at runtime. If L is empty, that is, no ITC nodes are bounded-buffer fireable and free for execution, we force the thread to wait for a signal indicating changes in L (line 31). Returning back to line 8, after firing v, we update the number of tokens on input and output edges of v, and examine whether v and whether the adjacent nodes of v are bounded-buffer fireable; this is because state transitions in surrounding edges of v only affect bounded-buffer fireability of v and its adjacent nodes. If they become bounded-buffer fireable, we push them onto L. Finally, we signal the possible changes in L, and if there are threads waiting for fireable ITC nodes, this will wake them up. Again, for synchronization purposes and for correctness in multithreaded implementation, the lock mechanism is applied whenever there is a change of state related to ITC nodes, ITC edges, and the fireable list L.

7. SIMULATION RESULTS

In practical implementation of MSS, estimates of actor execution times are required in order to compute the SRTP value of each actor and the SRTP threshold for graph clustering at compile-time. In general, a single actor's functionality may range from

```
SELF-SCHEDULED-EXECUTION(G,L)
input: a consistent SDF graph G = (V, E), a fireable list L
 1  while simulation is not terminated
 2    lock L's lock
 3    if L is not empty
 4      pop the first actor v from L
 5      unlock L's lock
 6      n = min(min_{e∈in(v)} ⌊tok(e)/cns(e)⌋, min_{e∈out(v)} ⌊(buf(e) − tok(e))/prd(e)⌋)
 7      fire v for n times
 8      for each edge e ∈ in(v)
 9        lock e's lock, tok(e) = tok(e) − n × cns(e), unlock e's lock
10      end
11      for each edge e ∈ out(v)
12        lock e's lock, tok(e) = tok(e) + n × prd(e), unlock e's lock
13      end
14      lock v's lock
15      if v is bounded-buffer fireable
16        unlock v's lock, lock L's lock, push v in L, unlock L's lock
17      else
18        set v's state to not-fireable, unlock v's lock
19      end
20      for each node u ∈ adj(v)
21        lock u's lock
22        if u is bounded-buffer fireable and u's state is not-fireable
23          set u's state to fireable, unlock u's lock
24          lock L's lock, push u in L, unlock L's lock
25        else
26          unlock u's lock
27        end
28      end
29      lock L's lock, signal L's condition-variable, unlock L's lock
30    else
31      wait for L's condition-variable to be signaled
32      unlock L's lock
33    end
34  end
```

Fig. 6. Self-scheduled multithreaded execution function.

simple operations, such as addition, multiplication, etc., to complex operations such as FFT, FIR, etc. Due to this reason (and also based on our experiments), setting uniform actor execution times usually causes unacceptable results. Furthermore, using an actor's production and consumption rates as execution-time cost functions also results in poor performance. In our approach, we perform *actor execution-time profiling* to collect estimates of actor execution times before scheduling. The profiling process repeatedly runs an actor for a short time and takes an average.

We have implemented and integrated the Multithreaded Simulation Scheduler (MSS) in the Advanced Design System (ADS) from Agilent Technologies [Pino and Kalbasi 1998]. However, the design of MSS is not specific to ADS, and the techniques presented in this article can be generally implemented in any simulation tool that incorporates SDF semantics and works in multithreaded environments. Indeed, the definitions, theoretical results, and algorithms have been carefully presented in this article in a manner that is not specific to ADS.

Our experimental platform is an Intel dual-core hyperthreading 3.46 GHz processor with 1GB memory running the Windows XP operating system. The processor contains

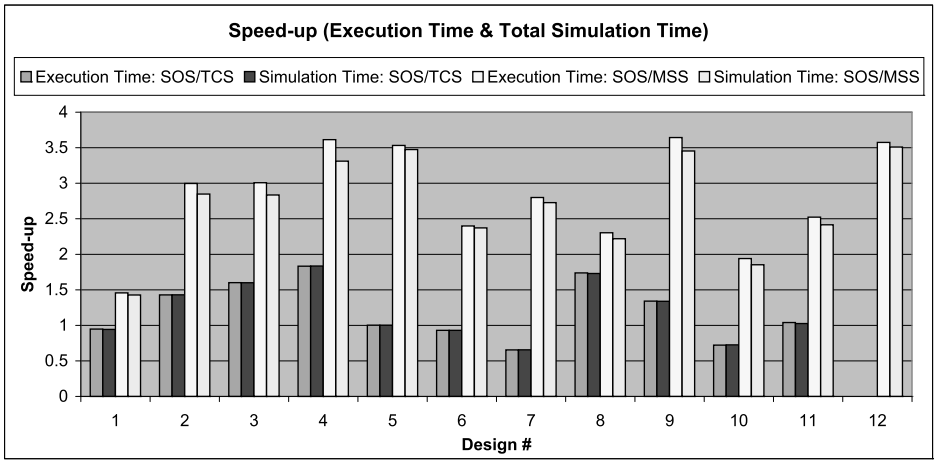

Fig. 7. Speedup: execution time and total simulation time.

4 processing units capable of executing 4 threads in parallel. We use the NSPR API [Mozilla.org 2011] as the targeted multithread library. In the experiments, we use the following three schedulers: (1) our Multithreaded Simulation Scheduler (MSS), (2) the Thread Cluster Scheduler (TCS) [Kin and Pino 2003] in ADS, and (3) our Simulation-Oriented Scheduler (SOS) [Hsu et al. 2007]. As discussed in Section 3, TCS was developed previously in ADS for simulation speedup using multithreaded execution, and it is the only prior work that we are aware of for multithreaded SDF simulation. Also, as presented in Hsu et al. [2007], SOS was developed for joint minimization of time and memory requirements when simulating large-scale and highly multirate SDF graphs in single-processor environments (single-thread execution semantics). We use SOS as the single-thread benchmark scheduler for comparing TCS and MSS to state-of-the-art, single-thread SDF execution methods.

In our experiment with MSS, the parameter M for the SRTP threshold (Eq. (9)) is set to 32, and the buffer upper bound is set to 4,500,000 tokens. We include 12 wireless communication designs from Agilent Technologies in the following standards: WCDMA3G (3GPP), CDMA 2000, WLAN (802.11a and 802.11g), WiMax (WMAN, 802.16e), Digital TV, and EDGE. The numbers of nodes and edges in the SDF graphs of these designs are in the range of hundreds. After compile-time scheduling in MSS, the numbers of nodes and edges in the resulting ITC graphs are effectively reduced to within a hundred. In the experiment, we collect both execution time and total simulation time results: here, *execution time* refers to the time spent in executing the graph, and this is the component that can be sped up by multithreaded execution; *total simulation time* refers to the time spent in overall simulation, including actor profiling, scheduling, buffer allocation, and execution. Figure 7 presents execution time and total simulation time speedup for TCS over SOS (SOS/TCS) and for MSS over SOS (SOS/MSS).

As shown in Figure 7, MSS outperforms TCS in all designs. MSS can achieve around 3.5 times execution time speedup on designs 4, 5, 9, 12, and around 2 to 3 times execution time speedup on designs 2, 3, 6, 7, 8, 11. The speedup from MSS is provided by not only the multicore capability but also the novel clustering and actor vectorization techniques. TCS performs worse than single-thread SOS in designs 1, 6, 7, and 10 due to its unbalanced partitioning, which takes numbers of firings into account rather than SRTP values. TCS also encounters out-of-memory problems in design 12 due to its

heavy dependence on the cluster loop scheduler, which cannot reliably handle highly multirate SDF graphs (see Hsu et al. [2006]).

Regarding the total simulation time , MSS spends around 2 to 10 seconds more compared to execution time due to overheads in environment setup, actor profiling, scheduling, buffer allocation, and multithreading initialization and termination. In contrast, SOS only requires around 1 to 3 seconds more. Based on our experiments, scheduling time for MSS is similar or even faster than SOS. The overheads from MSS are mostly due to actor profiling, multithreading initialization/termination, and longer buffer allocation (because MSS trades off buffer requirements for synchronization reduction). However, the additional overhead from MSS is insignificant compared to the large simulation times that are observed. For long-term simulations, our results have shown that MSS is a very effective approach to speeding up overall simulation for SDF-based designs.

In theory, MSS is scalable with the advance of multicore processor technology, as long as Eq. (9) holds. However, the speedups from MSS for design 1 and 10 are clearly not as significant as for other designs, which reveals potential scalability limitation. Based on our investigation, the scalability of MSS may in general be limited by a number of inherent properties in the designs. First, a design may contain actors whose SRTP values are larger than the SRTP threshold. In other words, a heavily computational actor with large repetition count may become a bottleneck in multithreaded execution. Second, a design may involve actors that require slow or nonparallelizable external resources. For example, slow hard drive reading or writing operations may become bottlenecks. In addition, file readers and writers from parallel threads may compete for hard drive channels. Third, a design may involve Strongly Connected Components (SCCs) whose SRTP values are larger than the SRTP threshold. This limitation results from SCC clustering in MSS. However, decompositions of SCC clusters may not help in some cases. For example, in a large homogeneous cycle (i.e., a cycle in which production and consumption rates are identically equal to 1) with only a single initial delay, actors can only be executed sequentially. In these cases, clustering SCCs and computing static schedules is in general a more efficient approach.

Investigating techniques to address these limitations and further extend the power of MSS is a useful direction for further work.

8. CONCLUSION

Simulation of state-of-the-art wireless communication systems and their associated Synchronous DataFlow (SDF) models is an important area for development of parallel CAD techniques. Motivated by the increasing popularity of multicore processors that provide on-chip, thread-level parallelism, we have proposed multithreaded simulation of SDF graphs to achieve simulation runtime speedup. We have illustrated the challenges in scheduling large-scale, highly multirate SDF graphs for multithreaded execution. We have introduced Ω-scheduling and associated throughput analysis as theoretical foundations in our developments. We have then presented the novel Multithreaded Simulation Scheduler (MSS). The compile-time scheduling approach in MSS strategically integrates graph clustering, actor vectorization, intracluster scheduling, and intercluster buffering techniques to construct InterThread Communication (ITC) SDF graphs for multithreaded execution. Then the runtime scheduling approach in MSS applies self-scheduled multithreaded execution model for efficient execution of ITC graphs in multithreaded environments. Finally, on a multithreaded platform equipped with 4 processing units, we have demonstrated up to 3.5 times speedup in simulating modern wireless communication systems with MSS.

REFERENCES

ADE, M., LAUWEREINS, R., AND PEPERSTRAETE, J. A. 1997. Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets. In *Proceedings of the Design Automation Conference*.

BHATTACHARYYA, S. S. AND LEE, E. A. 1994. Memory management for dataflow programming of multirate signal processing algorithms. *IEEE Trans. Signal Process. 42,* 5, 1190–1201.

BHATTACHARYYA, S. S., MURTHY, P. K., AND LEE, E. A. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers.

BUCK, J. T. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. Thesis UCB/ERL 93/69, Department of EECS, University of California Berkeley.

BUCK, J. T. AND VAIDYANATHAN, R. 2000. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the International Workshop on Hardware/Software Codesign*.

BUCK, J. T., HA, S., LEE, E. A., AND MESSERSCHMITT, D. G. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. J. Comput. Simul. 4*, 155–182.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms* 2nd Ed. The MIT Press.

CUBRIC, M. AND PANANGADEN, P. 1993. Minimal memory schedules for dataflow networks. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'93)*. 368–383.

EKER, J., JANNECK, J. W., LEE, E. A., LIU, J., LIU, X., LUDVIG, J., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2003. Taming heterogeneity - The Ptolemy approach. *Proc. IEEE 91,* 1, 127–144.

GEILEN, M., BASTEN, T., AND STUIJK, S. 2005. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the Design Automation Conference*. 819–824.

GHAMARIAN, A. H., GEILEN, M. C. W., STUIJK, S., BASTEN, T., MOONEN, A. J. M., BEKOOIJ, M. J. G., THEELEN, B. D., AND MOUSAVI, M. R. 2006. Throughput analysis of synchronous data flow graphs. In *Proceedings of the International Conference on Application of Concurrency to System Design*.

GOVINDARAJAN, R., GAO, G. R., AND DESAI, P. 2002. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *J. VLSI Signal Process. 31*, 207–229.

HSU, C. 2007. Dataflow integration and simulation techniques for DSP system design tools. Ph.D. thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park.

HSU, C., KO, M., AND BHATTACHARYYA, S. S. 2005. Software synthesis from the Dataflow Interchange Format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*. 37–49.

HSU, C., KO, M., BHATTACHARYYA, S. S., RAMASUBBU, S., AND PINO, J. L. 2007. Efficient simulation of critical synchronous dataflow graphs. *ACM Trans. Des. Autom. Electron. Syst. 12,* 3, 21.

HSU, C., PINO, J. L., AND BHATTACHARYYA, S. S. 2008. Multithreaded simulation for synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*. 331–336.

HSU, C., PINO, J. L., AND HU, F. 2010. A mixed-mode vector-based dataflow approach for modeling and simulating LTE physical layer. In *Proceedings of the Design Automation Conference*.

HSU, C., RAMASUBBU, S., KO, M., PINO, J. L., AND BHATTACHARYYA, S. S. 2006. Efficient simulation of critical synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*. 893–898.

KIANZAD, V. AND BHATTACHARYYA, S. S. 2006. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Trans. Parall. Distrib. Syst. 17,* 7, 667–680.

KIM, S. J. AND BROWNE, J. C. 1988. A general approach to mapping of parallel computations upon multiprocessor architectures. In *Proceedings of the International Conference on Parallel Processing*. Vol. 3.

KIN, J. S. AND PINO, J. L. 2003. Multithreaded synchronous data flow simulation. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*.

KO, M., MURTHY, P. K., AND BHATTACHARYYA, S. S. 2004. Compact procedural implementation in DSP software synthesis through recursive graph decomposition. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*. 47–61.

KO, M., SHEN, C., AND BHATTACHARYYA, S. S. 2006. Memory-constrained block processing optimization for synthesis of DSP software. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 137–143.

LALGUDI, K. N., PAPAEFTHYMIOU, M. C., AND POTKONJAK, M. 2000. Optimizing computations for effective block-processing. *ACM Trans. Des. Autom. Electron. Syst. 5,* 3, 604–630.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Synchronous dataflow. *Proc. IEEE 75,* 9, 1235–1245.

MOZILLA.ORG. 2011. NSPR reference. http://www.mozilla.org/projects/nspr/reference/html/index.html.

MURTHY, P. K. AND BHATTACHARYYA, S. S. 2006. *Memory Management for Synthesis of DSP Software*. CRC Press.

PINO, J. L. AND KALBASI, K. 1998. Cosimulating synchronous DSP applications with analog RF circuits. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*.

REITER, R. 1968. Scheduling parallel computations. *J. Assoc. Comput. Mach. 15*, 4.

RITZ, S., PANKERT, M., ZIVOJINOVIC, V., AND MEYR, H. 1993. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application-Specific Array Processors*. 285–296.

SAHA, S., SHEN, C., HSU, C., VEERARAGHAVAN, A., SUSSMAN, A., AND BHATTACHARYYA, S. S. 2006. Model-based OpenMP implementation of a 3D facial pose tracking system. In *Proceedings of the Workshop on Parallel and Distributed Multimedia*.

SARKAR, V. 1989. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press.

SRIRAM, S. AND BHATTACHARYYA, S. S. 2009. *Embedded Multiprocessors: Scheduling and Synchronization* 2nd Ed. CRC Press.

STEFANOV, T., ZISSULESCU, C., TURJAN, A., KIENHUIS, B., AND DEPRETTERE, E. 2004. System design using Kahn process networks: The Compaan/Laura approach. In *Proceedings of the Design, Automation and Test in Europe Conference*.

STUIJK, S., GEILEN, M., AND BASTEN, T. 2006. Exploring tradeoffs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*.

SUNG, W., OH, M., IM, C., AND HA, S. 1997. Demonstration of hardware software codesign workflow in PeaCE. In *Proceedings of the International Conference on VLSI and CAD*.

Received March 2010; revised November 2010, March 2011; accepted March 2011