Multithreaded Simulation for Synchronous Dataflow Graphs

Chia-Jui Hsu
Agilent Technologies, Inc.
Westlake Village, CA 91362
jerry_hsu@agilent.com

José Luis Pino
Agilent Technologies, Inc.
Westlake Village, CA 91362
jpino@agilent.com

Shuvra S. Bhattacharyya
Dept. of ECE and UMIACS
University of Maryland
College Park, MD 20742
ssb@umd.edu

ABSTRACT

Synchronous dataflow (SDF) has been successfully used in design tools for system-level simulation of wireless communication systems. Modern wireless communication standards involve large complexity and highly-multirate behavior, and typically result in long simulation time. The traditional approach for simulating SDF graphs is to compute and execute static single-processor schedules. Nowadays, multi-core 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 multithreading capability, current design tools gain only minimal performance improvements. In this paper, we present a new *multithreaded simulation scheduler*, called MSS, to provide simulation runtime speed-up for executing SDF graphs on multi-core processors. 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 speed-up 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.

Keywords: Synchronous dataflow, Multithreaded simulation, Scheduling.

1. INTRODUCTION

System-level modeling and simulation using electronic design automation (EDA) and electronic system-level (ESL) tools are key steps in the design process for communication and signal processing systems. The synchronous dataflow (SDF) [11] and timed synchronous dataflow (TSDF) [13] models of computation are widely used in design tools for these purposes, including ADS from Agilent [13], DIF from University of Maryland [6], and many others listed in [8].

According to [8], modern wireless communication systems involve large complexity and highly multirate behavior, and SDF representations of such systems typically result in large memory requirement and long simulation time. Nowadays, multi-core processors are prevalent for their potential speed-up through on-chip, thread-level parallelism, e.g., dual-core and quad-core CPUs from Intel and AMD. However, current design tools gain only minimal performance improvements due to the underlying sequential (single-thread) SDF execution semantics.

In this paper, we focus on multithreaded simulation of SDF graphs, and our objective is to *speed up* simulation runtime. 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 actors in the SDF graph into multiple clusters such that actors in the same cluster are executed sequentially by a single 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 implementations, it is necessary to impose such bounds on buffer sizes.
4. *Assignment* — Allocating threads and assigning clusters to threads for concurrent execution, under the constraint that each cluster can only be executed by one thread at any given time.
5. *Synchronization* — Determining when a cluster is executed by a thread, and synchronizing between multiple concurrent threads such that all data precedence and buffer bound constraints are satisfied.

We develop the *multithreaded simulation scheduler* (MSS) to systematically exploit multithreading capabilities in SDF graphs. The compile-time scheduling in MSS strategically integrates graph clustering, actor vectorization, intra-cluster scheduling, and inter-cluster buffering techniques to jointly perform static clustering, ordering, and buffering for trading off between throughput, synchronization overhead, and buffer requirements. From this compile-time scheduling, *inter-thread communication* (ITC) SDF graphs are constructed for multithreaded execution. The runtime scheduling in MSS then applies the *self-scheduled multithreaded execution model* to perform dynamic assignment and synchronize multiple threads for executing ITC graphs at runtime.

The organization of this paper is as follows: We review

DAC 2008, June 8–13, 2008, Anaheim, California, USA

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. Due to page limitations, we refer the readers to [5] for detailed descriptions and proofs.

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 $snk(e)$. A dataflow edge e can have a non-negative integer *delay* $del(e)$, and 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 tokens on all of its input edges $in(v)$. When firing, v consumes certain numbers of tokens from its input edges 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 $prd(e) = cns(e)$; a *multirate* edge if $prd(e) \neq cns(e)$.

Before execution, a *schedule* of a dataflow graph is computed. 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 there is a positive integer solution to the *balance equations*:

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

When it exists, the minimum positive integer solution for the vector $\boldsymbol{x}$ is called the *repetitions vector* of G [11], 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* 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 [1] is an important operation in scheduling SDF graphs. Given an SDF graph $G = (V, E)$, clustering a subset $Z \subseteq V$ into a *supernode* α means transforming G into a smaller graph $G' = (V - Z + \{\alpha\}, E')$ such that executing α in G' corresponds to executing one iteration of a minimal periodic schedule for the subgraph associated with Z.

3. RELATED WORK

Various scheduling algorithms have been developed for different applications of SDF graphs. For software synthesis onto embedded single-processor implementations, Bhattacharyya et al. [1] have developed algorithms for joint code and memory minimization; Ko et al. [10] and Ritz et al. [14] have developed actor vectorization (or block processing) techniques to improve execution performance.

For simulation of SDF graphs in single-processor environments, the *cluster-loop scheduler* has been developed based on [2] in the Ptolemy environment 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 from large runtimes and buffer requirements in heavily multirate systems [8]. Hsu et al. have developed the *simulation-oriented scheduler* [8, 7], which strategically integrates several techniques for graph decomposition and SDF scheduling to provide effective, joint minimization of time and memory requirements for simulating SDF graphs.

Heuristics for minimum buffer scheduling have been developed in, e.g., [1]. Various methods have been developed to analyze throughput in SDF graphs, e.g., [16, 3], and to explore tradeoffs between buffer sizes and throughput, e.g., [4, 17]. These approaches are useful for certain forms of synthesis. However, the complexities of these approaches are not polynomially bounded in the graph size.

Multiprocessor scheduling for dataflow graphs and related models has been extensively studied in the literature, e.g., see [15, 16]. Sriram and Bhattacharyya [16] reviewed an abundant set of scheduling and synchronization techniques for embedded multiprocessors.

Regarding SDF scheduling specific to multithreaded simulation, the only previous work that we are aware of is the *thread cluster scheduler* developed by Kin and Pino [9] 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 recursive two-way clusters with multiple threads in a pipelined fashion. Experimental results in [9] 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.

4. Ω-SCHEDULING

As discussed in Section 1, the problem of scheduling SDF graphs for multithreaded execution is highly complex. Our first step is to develop solutions to achieve maximal throughput assuming unbounded processing resources.

4.1 Definition and Throughput Analysis

In dataflow-related tools, actors may have internal state that prevents executing multiple invocations of the actors in parallel, e.g., FIR filters. Furthermore, whether or not an actor has internal state may be a lower level detail that is not visible to the scheduler. This is, for example, the case in Agilent ADS. Thus, exploring data-level parallelism by duplicating actors onto multiple processors is out of the scope of this paper.

In pure dataflow semantics, data-driven execution assumes that edge buffers have infinite capacity. For practical implementations, it is necessary to impose bounds on buffer sizes. Given an SDF graph $G = (V, E)$, we denote the number of tokens on an edge $e \in E$ at some particular instant in time by $tok(e)$ (where the time instant t is to be understood from context and therefore suppressed from the notation). Let Z^+ denote the set of positive integers. We denote the buffer size of an edge e by $buf(e)$, and denote $f_B : E \rightarrow Z^+$ as the associated buffer size function. Also, we denote the execution time of an actor v by $t(v)$, and denote $f_T : V \rightarrow Z^+$ as the associated actor execution time function.

In the following definition, we specify conditions that must be met before firing an actor in the bounded-buffer context.

Definition 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* if 1) v is fireable — i.e., v has sufficient numbers of tokens on all of its input edges (data-driven property) — $\forall e \in in(v)$, $tok(e) \geq cns(e)$, and 2) v has sufficient numbers of spaces on all of its output edges (bounded-buffer property) — $\forall e \in out(v)$, $buf(e) - tok(e) \geq prd(e)$.

Recall that the task of synchronization is to maintain data precedence and bounded-buffer constraints. As a result, the most intuitive scheduling strategy for maximal throughput is to fire an actor as soon as it is bounded-buffer fireable.

Definition 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.

THEOREM 4.1. *[5] 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*

$$\frac{1}{\max_{v \in V}(\boldsymbol{q}_G[v] \times t(v))}. \quad (2)$$

THEOREM 4.2. *[5] Given a consistent SDF graph $G = (V, E)$ and an actor execution time function f_T, Equation (2) is the throughput upper bound in Ω-scheduling.*

4.2 Buffering for Maximum Throughput

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

In this development, it is useful to employ the following notions. Given a connected graph $G = (V, E)$, a *parallel edge set* $[u, v]$ is a set of edges $\{e \in E | src(e) = u \text{ and } snk(e) = v\}$ that connect from the same vertex u to the same vertex v. Here, by considering each parallel edge set $[u, v]$ as a single edge (u, v), a *biconnected component* is a maximal set of edges A such that any pair of edges in A lies in a simple undirected cycle. A *bridge* is then an edge (parallel edge set) that does not belong to any biconnected component. Finally, a *biconnected-component-free (BCF) partition* is a partition of V such that clustering the partition does not introduce biconnected components.

In the theorem below, we present buffer analysis for acyclic SDF graphs that contain no biconnected components.

THEOREM 4.3. *[5] Suppose that we are given a consistent SDF graph $G = (V, E)$, and suppose 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 setting buffer sizes for each $[u, v] \in E$ according to Equation (3):*

$$\forall e_i \in [u, v], buf(e_i) = \begin{cases} (p_i + c_i - g_i) \times 2 + d_i - d^* \times g_i \\ \quad \text{if } 0 \leq d^* \leq (p^* + c^* - 1) \times 2, \\ d_i, \text{ otherwise.} \end{cases} \quad (3)$$

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

```
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   apply Equation (3)   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} = BCF-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 Theorem 4.3 on G_i'
8       for each 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
```

Figure 1: Ω-Acyclic-Buffering algorithm.

In Theorem 4.3, gcd stands for greatest common divisor. Applying Theorem 4.3 to general acyclic SDF graphs may cause deadlock in Ω-scheduling. In order to allocate buffers for general acyclic SDF graphs, we have developed the Ω-*Acyclic-Buffering* algorithm as shown in Figure 1, and we state the validity of the algorithm in Theorem 4.4.

THEOREM 4.4. *[5] 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

In this section, we present compile-time scheduling techniques based on the concept of Ω-scheduling to construct inter-thread 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, the available resources on current multi-core processors is limited — usually 2 or 4 processing units are available. As a result, threads are competing for processing units for both execution and synchronization (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 intra-cluster scheduling is one of our strategies to reduce synchronization overhead. We formalize this scheduling strategy in Definition 3.

Definition 3. Given a consistent SDF graph $G = (V, E)$, Π-*scheduling* is defined as the SDF scheduling strategy that 1) clusters a *consistent partition* $P = (Z_1, Z_2, \ldots, Z_{|P|})$ of V such that G is transformed into a smaller consistent SDF graph $G_P = (V_P = \{v_1, v_2, \ldots, v_{|P|}\}, E_P)$; 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.

After clustering, the synchronization overhead is reduced from Q_G to $Q_{G_P} = \sum_{v_i \in V_P} \boldsymbol{q}_{G_P}[v_i]$, and the repetition count

of v_i becomes $\boldsymbol{q}_{G_P}[v_i] = \gcd_{v \in Z_i} \boldsymbol{q}_G[v]$ [1]. Clustering may increase buffer requirements because the interface production and consumption rates of the resulting supernodes are scaled by positive-integer factors in order to preserve multirate consistency [1]. Clustering also decreases throughput in our model because it generally reduces the amount of available parallelism. In the following theorem, we analyze the effect of clustering on throughput under Ω-scheduling.

THEOREM 5.1. *[5] 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 consistent 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*

$$\frac{1}{\max_{Z_i \in P}(\sum_{v \in Z_i}(\boldsymbol{q}_G[v] \times t(v)))}. \tag{4}$$

In addition, if G_P is acyclic, Equation (4) gives the maximum achievable throughput.

Theorem 5.1 tells us that 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 in a cluster. 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 5.1, the cluster with the largest SRTP value dominates the overall throughput.

In single-processing-unit environments, the ideal iteration period for executing a consistent SDF graph $G = (V, E)$ is $SRTP(G) = \sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$. Now considering an N-processing-units processor, the ideal throughput can be expressed as $N / \sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))$. In the clustering process, by imposing Equation (5) as a constraint for each cluster (partition) Z_i, the ideal N times speed-up can be achieved theoretically only when the SRTP threshold factor M in Equation (5) is larger than or equal to N, where the right hand side of Equation (5) is called the *SRTP threshold*.

$$\sum_{v \in Z_i}(\boldsymbol{q}_G[v] \times t(v)) \leq \sum_{v \in V}(\boldsymbol{q}_G[v] \times t(v))/M \tag{5}$$

In practice, M is usually set larger than N to tolerate estimation and variation in actor execution time — that is, by having more small (in terms of estimated 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, and depends on the graph size and other instance-specific factors.

Actor vectorization (*actor looping*) is our second strategy to reduce synchronization overhead. The main idea is to vectorize an actor's execution by a factor of the associated repetition count.

Definition 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, $\boldsymbol{q}_G[v^k] = \boldsymbol{q}_G[v]/k$.

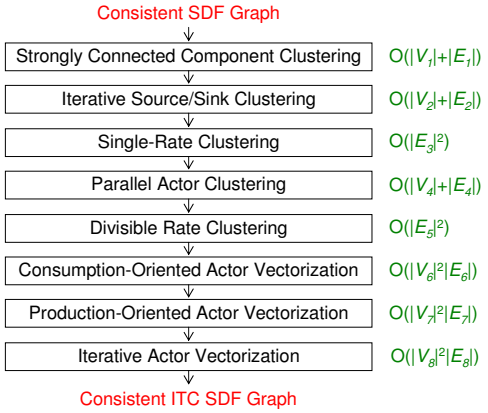

Figure 2: Compile-time scheduling framework.

In practical highly multirate SDF graphs, repetitions vectors usually consist of large, non-prime numbers. As a result, actor vectorization is suitable for synchronization reduction in this context, but at the possible expense of larger buffer requirements due to the multiplication of production and consumption rates.

5.2 Compile-time Scheduling Framework

Based on the above concepts, we develop a compile-time scheduling framework, as shown in Figure 2, to transform and schedule an input SDF graph into an *inter-thread communication* (ITC) SDF graph G_{itc}. We choose the term "ITC graph" because each node (cluster) in G_{itc} is executed by a thread. In MSS, ITC graphs are carefully constructed based on a careful assessment of trade-offs among synchronization overhead, throughput, and buffer requirements.

This framework strategically integrates 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. We also incorporate the simulation-oriented scheduler [8, 7] into this framework to compute static intra-cluster schedules and buffer sizes along the way in the bottom-up process. Finally, we apply the Ω-Acyclic-Buffering technique to compute buffer sizes for ITC graphs. All the integrated algorithms emphasize low complexity; the associated complexities are summarized in Figure 2. Note that in the input graphs for the individual algorithms, the numbers of nodes $|V_i|$ and edges $|E_i|$ decrease progressively through the bottom-up clustering process. For detailed description of the framework, we refer the reader to [5].

Given a consistent SDF graph as input, we first cluster strongly connected components (SCCs) to generate an acyclic graph. This is because existence of cycles in an ITC graph may decrease throughput, and if the SRTP value of each SCC satisfies Equation (5), clustering SCCs does not cause limitations in the maximum achievable throughput.

From acyclic graphs, we apply the following set of clustering techniques to jointly explore both topological properties and dataflow-oriented properties such that clustering does not introduce cycles nor increase buffer requirements. These techniques are effective in clustering SDF graphs because the properties that they address arise commonly in practical communication and signal processing systems. In addition, we impose Equation (5) in the clustering process to ensure sufficient throughput.

Briefly speaking, *iterative source/sink clustering* focuses on subsystems that have the form of chain- or tree-structures at the source- or sink-end of a graph, and the SDF rate behavior involved in such subsystems is successively divisible in certain directions. After that, *single-rate clustering* explores single-rate subsystems where for each edge, the associated production and consumption rates are the same.

Next, *parallel actor clustering* explores parallel actors with the same repetition count. Here, we say actors u and v are *parallel* if there is no path from u to v nor from v to u. Then, *divisible-rate clustering* iteratively searches for a candidate pair of actors such that after clustering, the increased production (consumption) rate of the supernode can divide the consumption (production) rate at the other end of the edge.

After that, we apply the following set of actor vectorization techniques to strategically trade buffer cost for synchronization overhead reductions and also to explore single-rate clustering opportunities exposed by such vectorizations. Due to highly multirate nature of our target wireless applications, the total buffer requirement is carefully kept within the given upper bound to prevent out-of-memory problems.

Consumption-oriented actor vectorization takes advantage of consumption-rate-divisible edges (i.e., $cns(e)$ is divisible by $prd(e)$) such that $src(e)$ is vectorized for $cns(e)/prd(e)$ times to match the rate of $snk(e)$. Then *production-oriented actor vectorization* is applied in a similar manner to take advantage of production-rate-divisible edges.

Finally, we apply *iterative actor vectorization* to explore both divisible and indivisible multirate interconnections. Here, for adjacent actors u and 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. This technique iteratively vectorizes a properly-chosen actor such that the reduction in synchronization overhead can be maximized while the penalty in buffer cost is minimal.

6. RUNTIME SCHEDULING

In this section, we develop runtime scheduling techniques for the assignment and synchronization tasks. 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 assigns 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.

In the above approach, however, when the number of fireable ITC nodes is larger than the number of processing units (which is usually very limited), multithreading APIs and operating systems take additional overheads in scheduling the usage of processing units. Motivated by this observation, we develop the *self-scheduled multithreaded execution model* to provide efficient method for executing ITC graphs in multithreaded environments.

Definition 5. 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 blocks when none of the ITC nodes are bounded-buffer fireable.

This execution model performs dynamic assignment between ITC nodes and threads, and it synchronizes threads

```
Self-Scheduled-Execution(G, L)
input: a consistent SDF graph G = (V, E) and a fireable list L
 1 while simulation is not terminated
 2   if L is not empty
 3     pop the first actor v from L
 4     n = min( min_{e∈in(v)} ⌊tok(e)/cns(e)⌋,
           min_{e∈out(v)} ⌊(buf(e) − tok(e))/prd(e)⌋ )
 5     fire v for n times
 6     for e ∈ in(v)    tok(e) = tok(e) − n × cns(e)    end
 7     for e ∈ out(v)    tok(e) = tok(e) + n × prd(e)    end
 8     if v is bounded-buffer fireable    push v in L    end
 9     for each node u ∈ adj(v)
10       if u is bounded-buffer fireable and u is not in L
             push u in L    end
11     end
12     signal state change of L
13   else
14     wait for state change of L to be signalled
15   end
16 end
```

Figure 3: Self-Scheduled-Execution function.

based on bounded-buffer fireability.

Figure 3 presents the *Self-Scheduled-Execution* function that is executed by each thread in this model. The input list L contains ITC nodes that are initially bounded-buffer fireable. If L is not empty in line 2, 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 — i.e., no ITC nodes can be executed — we force the thread to wait for a signal indicating changes in L (line 14). Back to line 6, after firing v, we update the number of tokens on input and output edges of v. Because such update only affects bounded-buffer fireability of v and its adjacent nodes $adj(v)$, we push them onto L if they become bounded-buffer fireable. Finally, we signal the possible changes in L, and if there are threads waiting for fireable ITC nodes, this will wake them up. Signal and wait are multithreading operations that are widely available in multithreading APIs, e.g., [12]. In our implementation, the lock mechanism (which is not shown in Figure 3) is applied whenever there is a change of state related to ITC nodes, ITC edges, and the fireable list L.

7. SIMULATION RESULTS

We have implemented and integrated the multithreaded simulation scheduler (MSS) in the Advanced Design System (ADS) from Agilent Technologies [13]. In our implementation, we perform *actor execution time profiling* to estimate actor execution times in order to compute the SRTP values and the SRTP threshold. The profiling process repeatedly runs an actor for a short time and takes an average.

Our experimental platform is an Intel dual-core hyper-threading (4 processing units) 3.46 GHz processor with 1GB memory running the Windows XP operating system. We use the NSPR API [12] as the multithreading library. In the experiments, we use the following three schedulers: 1) the multithreaded simulation scheduler (MSS), 2) the thread cluster scheduler (TCS) [9] in ADS, and 3) the simulation-oriented scheduler (SOS) [8, 7]. We use SOS as the single-thread benchmark scheduler for comparing TCS and MSS to single-thread SDF execution methods.

In our experiment with MSS, the SRTP threshold factor M (Equation (5)) 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 based on the following standards: WCDMA3G (3GPP), CDMA 2000, WLAN (802.11a and 802.11g), WiMax, Digital TV, and EDGE. We

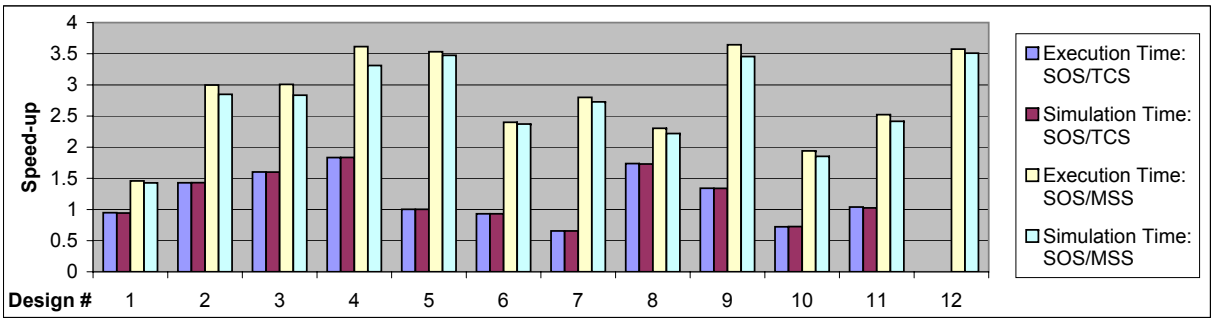

Figure 4: Speed-up (execution time and total simulation time).

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 speed-up by multithreaded execution; *total simulation time* refers to the time spent in overall simulation, including actor profiling, scheduling, buffer allocation, execution, and multithreading-related operations. Figure 4 presents the execution time and total simulation time speed-up for TCS over SOS (SOS/TCS) and for MSS over SOS (SOS/MSS).

As shown in Figure 4, MSS outperforms TCS in all designs. MSS can achieve around 3.5 times execution time speed-up on designs 4, 5, 9, 12, and around 2 to 3 times execution time speed-up on designs 2, 3, 6, 7, 8, 11. TCS performs worse than SOS in designs 1, 6, 7, and 10 due to its un-balanced partitioning, which takes numbers of firings into account rather than SRTP values. Furthermore, TCS 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 [8]).

Regarding the total simulation time, MSS spends around 2 to 10 seconds more compared to execution time. 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 and termination, and longer buffer allocation (because MSS trades off buffer requirements for synchronization overhead 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.

8. CONCLUSION

Motivated by the increasing popularity of multi-core processors, we have developed the multithreaded simulation scheduler (MSS) to achieve speed-up in simulating synchronous dataflow (SDF) based designs. We have introduced Ω-scheduling as the core theoretical foundations in our developments. The compile-time scheduling approach in MSS strategically integrates graph clustering, actor vectorization, intra-cluster scheduling, and inter-cluster buffering techniques to construct inter-thread communication (ITC) SDF graphs. Then the runtime scheduling approach in MSS applies the 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 speed-up in simulating modern wireless communication systems with MSS in the Advanced Design System (ADS) from Agilent Technologies.

9. REFERENCES

[1] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[2] J. T. Buck. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. Thesis UCB/ERL 93/69, Dept. of EECS, U. C. Berkeley, 1993.

[3] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput analysis of synchronous data flow graphs. In *Proceedings of the International Conference on Application of Concurrency to System Design*, Turku, Finland, June 2006.

[4] R. Govindarajan, G. R. Gao, and P. Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Jour. of VLSI Signal Processing*, 31:207–229, 2002.

[5] C. Hsu. *Dataflow Integration and Simulation Techniques for DSP System Design Tools*. PhD thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, April 2007.

[6] 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*, pages 37–49, Dallas, TX, Sept. 2005.

[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), August 2007.

[8] C. Hsu, S. Ramasubbu, M. Ko, J. L. Pino, and S. S. Bhattacharyya. Efficient simulation of critical synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*, pages 893–898, San Francisco, CA, July 2006.

[9] J. S. Kin and J. L. Pino. Multithreaded synchronous data flow simulation. In *Proc. of the Design, Automation and Test in Europe Conf. and Exhibition*, Munich, Germany, Mar. 2003.

[10] M. Ko, C. Shen, and S. S. Bhattacharyya. Memory-constrained block processing optimization for synthesis of DSP software. In *Proc. of the Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Greece, July 2006.

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

[12] Mozilla.org. *NSPR Reference*. Available: http://mozilla.org/projects/nspr/reference/html/index.html.

[13] 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*, Pacific Grove, CA, Nov. 1998.

[14] S. Ritz, M. Pankert, V. Zivojinovic, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application Specific Array Processors*, Venice, Italy, Oct. 1993.

[15] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, 1989.

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

[17] S. Stuijk, M. Geilen, and T. Basten. Exploring tradeoffs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*, pages 899–904, San Francisco, CA, July 2006.