Decidable Dataflow Models for Signal Processing: Synchronous Dataflow and Its Extensions

Soonhoi Ha and Hyunok Oh

Abstract Digital signal processing algorithms can be naturally represented by a dataflow graph where nodes represent function blocks and arcs represent the data dependency between nodes. Among various dataflow models, decidable dataflow models have restricted semantics so that we can determine the execution order of nodes at compile-time and decide if the program has the possibility of buffer overflow or deadlock. In this chapter, we explain the synchronous dataflow (SDF) model as the pioneering and representative decidable dataflow model and its decidability focusing on how the static scheduling decision can be made. In addition the cyclo-static dataflow model and a few other extended models are briefly introduced to show how they overcome the limitations of the SDF model.

1 Introduction

Digital signal processing (DSP) algorithms are often informally, but intuitively, described by block diagrams in which a block represents a function block and an arc or edge represents a dependency between function blocks. While a block diagram is not a programming model, it resembles a formal dataflow graph in appearance. Figure 1 shows a block-diagram representation of a simple DSP algorithm, which can also be regarded as a dataflow graph of the algorithm.

A dataflow graph is a graphical representation of a dataflow model of computation in which a node, or an *actor*, represents a function block that can be executed,

S. Ha (✉)
Seoul National University, Seoul, Korea
e-mail: sha@snu.ac.kr

H. Oh
Hanyang University, Seoul, Korea
e-mail: hoh@hanyang.ac.kr

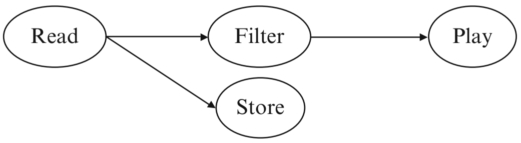


Fig. 1 Dataflow graph of a simple DSP algorithm

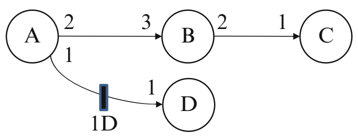
or *fired*, when enough input data are available. An arc is a FIFO channel that delivers data samples, also called *tokens*, from an output port of the source node to an input port of the destination node. If a node has no input port, the node becomes a source node that is always executable. In DSP algorithms, a source node may represent an interface block that receives triggering data from an outside source. The “Read” block in Fig. 1 is a source block that reads audio data samples from an outside source. A dataflow graph is usually assumed to be executed iteratively as long as the source blocks produce samples on the output ports.

The dataflow model of computation was first introduced as a parallel programming model for the associated computer architecture called dataflow machines [6]. While the granularity of a node is assumed as fine as a machine instruction in dataflow machine research, the node granularity can be as large as a well-defined function block such as a filter or an FFT unit in a DSP algorithm representation. The main advantage of the dataflow model as a programming model is that it specifies only the true dependency between nodes, revealing the function-level parallelism explicitly. There are many ways of executing a dataflow graph as long as data dependencies between the nodes are preserved. For example, blocks “Filter” and “Store” in Fig. 1 can be executed in any order after they receive data samples from the “Read” block. They can be executed concurrently in a parallel processing system.

To execute a dataflow graph on a target architecture, we have to determine where and when to execute the nodes, which is called *scheduling*. Scheduling decision can be made only at run-time for general dataflow graphs. A dynamic scheduler monitors the input arcs of each node to check if it is executable, and schedules the executable nodes on the appropriate processing elements. Thus dynamic scheduling incurs run-time overhead of managing the ready nodes to schedule in terms of both space and time. Another concern in executing a dataflow graph is resource management. While a dataflow graph assumes an infinite FIFO queue on each arc, a target architecture has a limited size of memory. Dynamic scheduling of nodes may incur buffer overflow or a deadlock situation if buffers are not carefully managed. A dataflow graph itself may have errors to induce deadlock or buffer overflow errors. It is not decidable for a general dataflow program whether it can be executed without buffer overflow or a deadlock problem.

On the other hand, some dataflow models have restricted semantics so that the scheduling decision can be made at compile-time. If the execution order of nodes is determined statically at compile-time, we can decide before running the program if the program has the possibility of buffer overflow or deadlock. Such dataflow graphs are called *decidable dataflow graphs*. More precisely, a dataflow is decidable if and only if a schedule of which length is finite can be constructed statically. Hence,

a



b

Periodic schedules
Σ1: ADADBCCADBCC
Σ2: 3(AD)2(B2(C))
Σ3: 3A3D2B4C

Fig. 2 (**a**) An SDF graph and (**b**) some periodic schedules for the SDF graph

in a decidable dataflow graph, the invocation number of each node is finite and computable at compile time. The *SDF (synchronous dataflow) model* proposed by Lee in [12], is a pioneering decidable model that has been widely used for DSP algorithm specification in many design environments including Ptolemy [5] and Grape II [11].

Subsequently, a number of generalizations of the SDF model have been proposed to extend the expression capability of the SDF model. The most popular extension is CSDF (cyclo-static dataflow) [4]. Three other extensions that aim to produce better software synthesis results will also be introduced in this chapter. In this chapter, we explain these decidable dataflow models and focus on their characteristics of decidability. For decidable dataflow graphs, the most important issue is to determine an optimal static schedule with respect to certain objectives since there are numerous ways to schedule the nodes.

2 Synchronous Dataflow

In a dataflow graph, the number of tokens produced (or consumed) per node firing is called the output (or the input) sample rate of the output (or the input) port. The simplest dataflow model is the single-rate dataflow (SRDF) in which all sample rates are unity. When a port may consume or produce multiple tokens, we call it multi-rate dataflow (MRDF). Among multi-rate dataflow graphs, synchronous dataflow (SDF) has a restriction that the sample rates of all ports are fixed integer values and do not vary at run time. Note that the SRDF is called sometimes the homogeneous SDF.

Figure 2a shows an SDF graph, which has the same topology as Fig. 1, where each arc is annotated with the number of samples produced and consumed by the incident nodes. There may be delay samples associated with an arc. The sample delay is represented as initial samples that are queued on the arc buffer from the beginning, and denoted as *xD* where *x* represents the number of initial samples, as shown on arc AD in Fig. 2a.

From the sample rate information on each arc, we can determine the relative execution rate of two end nodes of the arc. In order not to accumulate tokens unboundedly on an arc, the number of samples produced from the source node should be equal to the number of samples consumed by the destination node in the long run. In the example of Fig. 2a, the execution rate of node C should be twice as fast as the execution rate of node B on average. Based on this pair-wise information on the execution rates, we can determine the ratio of execution rates among all nodes. The resultant ratio of execution rates among nodes A, B, C and D in Fig. 2a becomes 3:2:4:3.

2.1 Static Analysis

The key analytical property of the SDF model is that the node execution schedule can be constructed at compile time. The number of executions of node A within a schedule is called the repetition count $x(A)$ of the node. A *valid* schedule is a finite schedule that does not reach deadlock and produces no net change in the number of samples accumulated on each arc. In a valid schedule, the ratio of repetition counts is equal to the ratio of execution rates among the nodes so that one iteration of a valid schedule does not increase the samples queued on all arcs. If there exists a valid schedule, the SDF graph is said *consistent* . We represent the repetition counts of nodes in a consistent SDF graph G by vector q_G. For the graph of Fig. 2a,

$$q_G = (x(A), x(B), x(C), x(D)) = (3, 2, 4, 3) \tag{1}$$

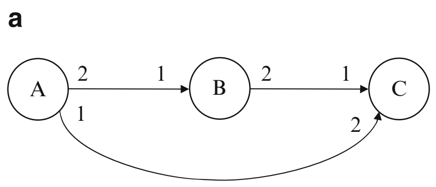
Since an SDF graph imposes only partial ordering constraints between the nodes, the order of node invocations can be determined in various ways. Figure 2b shows three possible valid schedules of the Fig. 2a graph. In Fig. 2b, each parenthesized term $n(X_1X_2 \ldots X_m)$ represents n successive executions of the sequence $X_1X_2 \ldots X_m$, which is called a looped schedule. If every block appears exactly once in the schedule such as $\Sigma 2$ and $\Sigma 3$ in Fig. 2b, the schedule is called a *single appearance (SA) schedule*. An SA-schedule that has no nested loop is called a *flat SA-schedule*. $\Sigma 3$ of Fig. 2b is a flat SA-schedule, while $\Sigma 2$ is not. Consistency analysis of an SDF graph is performed by constructing a valid schedule; no valid schedule can be found for an erroneous SDF graph.

To construct a valid schedule, we first compute the repetition counts of all nodes. For a given arc e, we denote the source node as $src(e)$ and the destination node as $snk(e)$. The output sample rate of $src(e)$ onto the arc is denoted as $prod(e)$ and the input sample rate of $snk(e)$ as $cons(e)$. Then, the following equation, called a *balance* equation, should be held for a consistent SDF graph.

$$x(src(e))prod(e) = x(snk(e))cons(e) \; for \; each \; e. \tag{2}$$

We can formulate the balance equations for all arcs compactly with the following matrix equation.

$$\Gamma q_G^T = 0 \tag{3}$$



b

$$\begin{array}{cc} & \begin{array}{ccc} node: A & B & C \end{array} \\ \Gamma = & \left(\begin{array}{ccc|c} 2 & -1 & 0 & AB \\ 0 & 2 & -1 & BC \\ 1 & 0 & -2 & AC \end{array}\right) \end{array} \quad arc:$$

Fig. 3 (**a**) An SDF graph that is sample rate inconsistent and (**b**) the associated topology matrix

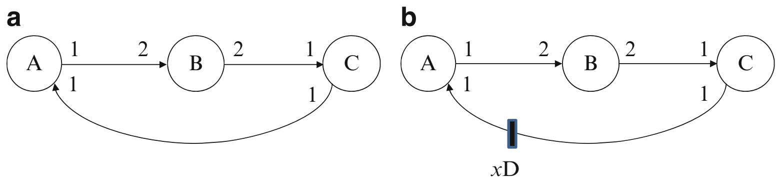


Fig. 4 (**a**) An SDF graph that is deadlocked and (**b**) the modified graph with initial samples on the feedback arc

where Γ, called the topology matrix of G, is a matrix of which rows are indexed by the arcs in G and columns are indexed by the nodes in G, An entry of the topology matrix is defined by

$$\Gamma(e,A) = \begin{cases} prod(e), & \text{if } A = src(e) \\ -cons(e), & \text{if } A = snk(e) \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

A valid schedule exists only if Eq. (3) has a non-zero solution of repetition vector q_G. Mathematically, this condition is satisfied when the rank of the topology matrix Γ is $n-1$ [12], where n is the number of nodes in G. In case no non-zero solution exists, the SDF graph is called *sample rate inconsistent*. Figure 3a shows a simple SDF graph that is sample rate inconsistent, and its associated topology matrix. Note that the rank of the topology matrix is 3, not 2.

Sample rate consistency does not guarantee that a valid schedule exists. A sample rate consistent SDF graph can be deadlocked as illustrated in Fig. 4a if the SDF graph has a cycle with insufficient amount of initial samples. The repetition vector, $q_G = (x(A), x(B), x(C))$, is (2,1,2). However, there is no fireable node since all nodes wait for input samples from each other. So we modify the graph by adding initial samples on arc CA in Fig. 4b. Suppose that there is an initial sample on arc CA, or $x = 1$. Then node A is fireable initially. After node A is fired, one sample is produced and queued into the FIFO channel of arc AB. But the graph is deadlocked again since no node becomes fireable afterwards. In this example, the minimum number of initial samples is two in order to rescue the graph from the deadlock condition.

The simplest method to detect deadlock is to construct a static SDF schedule by simulating the SDF graph as follows:

1. At first, make an empty schedule list that will contain the execution sequence of nodes, and initialize the set of fireable nodes.
2. Select one of the fireable nodes and put it in the schedule list. If the set of fireable nodes is empty, exit the procedure.
3. Simulate the execution of the selected node by consuming the input samples from the input arcs and producing the output samples to the output arcs.
4. Examine each destination node of the output arcs, and add it to the set of fireable nodes only if it becomes fireable and its execution count during the simulation is smaller than its repetition count.
5. Go back to step 2 to repeat this procedure.

When we complete this procedure, we can determine if the graph is deadlocked by examining the schedule list. If there is any node that is scheduled fewer times than its repetition count in the schedule list, the graph is deadlocked. Otherwise, the graph is deadlock-free. In summary, an SDF graph is consistent if it is sample rate consistent and it is deadlock-free. Therefore, the consistency of an SDF graph can be statically verified by computing the repetition counts of all nodes (sample rate consistency) and by constructing a static schedule.

2.2 *Software Synthesis from SDF Graph*

An SDF graph can be used as a graphical representation of a DSP algorithm, from which target codes are automatically generated. Software synthesis from an SDF graph includes determination of an appropriate schedule and a coding style for each dataflow node, both of which affect the memory requirements of the generated software. One of the main scheduling objectives for software synthesis is to minimize the total (sum of code and data) memory requirements.

For software synthesis, the kernel code of each node (function block) is assumed already optimized and provided from a predefined block library. Then the target software is synthesized by putting the function blocks into the scheduled position once a schedule is determined. There are two coding styles, *inline* and *function*, depending on how to put a function block into the target code. The former is to generate an inline code for each node at the scheduled position, and the latter is to define a separate function that contains the kernel of each node. Figure 5 shows three programs based on the same schedule $\Sigma 2$ of Fig. 2b. The first two use the inline coding style, and the third the function coding style.

If we use function calls, we have to pay, at run-time, the function-call overhead which can be significant if there are many function blocks of small granularity. If inlining is used, however, there is a danger of large code size if a node is instantiated multiple times. For the example of Fig. 2, schedule $\Sigma 1$ is not adequate for inlining unlike SA-schedules, $\Sigma 2$ and $\Sigma 3$. Figure 5b shows an alternative code that uses

a

```
main() {
for () {
 for (3) {
  code block of A;
  code block of D;
 }
 for (2) {
  code block of B;
  for (2) {
    code block of C;
  }
 }
} /* end of for */
} /* end of main */
```

b

```
main() {
 for () {
   switch (i) {
   case 1-3:
      code block of A;
      code block of D;
      break;
    case 4-5:
      code block of B;
      for (2) {code block of C;}
      break;
    default: i = 1; break;
   }
   i++;
 } /* end of for */
} /* end of main */
```

c

```
A() { code block of A; }
B() { code block of B; }
C() { code block of C; }
D() { code block of D; }

main() {
for() {
 for (3) { A() ; D(); }
 for (2) {
   B();
   for (2) { C(); }
 }
} /* end of for */
} /* end of main */
```

Fig. 5 Three programs based on the same schedule $\Sigma 2$ of Fig. 2b

Table 1 Buffer requirements for three schedules of Fig. 2b

Schedule	Arc AB	Arc AD	Arc BC	Total
$\Sigma 1$: ADADBCCADBCC	4	2	2	8
$\Sigma 2$: 3(AD)2(B2(C))	6	2	2	10
$\Sigma 3$: 3A3D2B4C	6	4	4	14

inlining without a proportional increase of code size to the number of instantiations of nodes. The basic idea is to make a simple run-time system that executes the nodes according to the schedule sequence. It pays the run-time overhead of switch-statements and code overhead for schedule sequence management. Hence an appropriate coding style should be selected considering the node granularity and the schedule.

For each schedule, the buffer size requirement can be computed. If we assume that a separate buffer is allocated on each arc, as is usually the case, the minimum buffer requirement of an arc becomes the maximum number of samples accumulated on the arc during an iteration of the schedule. For the example of Fig. 2, we can compare the buffer size requirements of three schedules as shown in Table 1.

From Table 1, we can observe that the SA schedules usually require larger buffers while they guarantee the minimum code size for inline code generation. In multimedia applications, frame-based algorithms are common where the size of a unit sample may be as large as a video frame or an audio frame. In these applications minimizing the buffer size is as important as minimizing the code size. In general, both code size and buffer size should be considered when we construct a memory-optimal schedule.

Buffering requirements can be reduced if we use buffer sharing. Arc buffers can be shared if their life-times are not overlapped with each other during an iteration of the schedule. The life-time of an arc buffer is defined by a set of durations from the source node invocation that starts producing a sample to the buffer to the completion of the destination node that empties the buffer. Consider schedule $\Sigma 1$ of Fig. 2. The buffer life-time of arc BC consists of two durations, $\{$BCC, BCC$\}$, in the schedule. Since the buffer of arc AD is never empty, the buffer life-time of arc AD is the entire duration of the schedule. If we remove the initial sample on arc AD, the buffer life-time of arc AD consists of three durations, $\{$AD, AD, AD$\}$. Then we can share the two arc buffers of arc AD and arc BC since their life-times are not overlapped. A more aggressive buffer sharing technique has been developed by separating global sample buffers and local pointer buffers in case the sample size is large in frame-based applications [13]. The key idea is to allocate a global buffer whose size is large enough to store the maximum amount of live samples during an iteration of the schedule. Each arc is assigned a pointer buffer that stores pointers to the global buffer.

Code size can also be reduced by sharing the kernel of a function block when there are multiple instances of the same block [22] in a dataflow graph. Multiple instances of the same block are regarded as different blocks, and the same kernel, possibly with different local states, may appear several times in the generated code. A technique has been proposed to share the same kernel by defining a shared function. Separate state variables and buffers should be maintained for each instance, which define the *context* of each instance. The shared function is called with the context of an instance as an argument at the scheduled position of the instance. To decide whether sharing a code block is beneficial or not, the overhead and the gain of sharing should be compared. If Δ is an overhead that is incurred by function sharing, R is a code block size, and n is the number of instances of a block, the decision function for code sharing is summarized as the following inequality: $\frac{\Delta}{(n-1)R} < 1$.

For more detailed information on the code generation procedure and other issues related with software synthesis from SDF graphs, refer to [2].

2.3 Static Scheduling Techniques

Static scheduling of an SDF graph is the key technique of static analysis that checks the consistency of the graph and determines the memory requirement of the generated code. Since an SDF graph imposes only partial ordering constraints between the nodes, there exist many valid schedules and finding an optimal schedule has been actively researched.

2.3.1 Scheduling Techniques for Single Processor Implementations

The main objective for a single processor implementation is to minimize the memory requirement, considering both the code and the buffer size. Since the problem of finding a schedule with minimum buffer requirement for an acyclic graph is NP-complete, various heuristic approaches have been proposed. Since a single appearance schedule guarantees the minimum code size for inline code generation, a group of researchers have focused on finding a single appearance schedule that minimizes the buffer size. Bhattacharyya et al. developed two heuristics: APGAN and RPMC, to find an SA-schedule that minimizes the buffer requirements [3]. Ritz et al. used an ILP formulation to find a flat single appearance schedule that minimizes the buffer size [19] considering buffer sharing. Since a flat SA-schedule usually requires more data buffer than a nested SA-schedule, it is not evident which approach is better between these two approaches.

Another group of researches tries to minimize only the buffer size. Ade et al. presented an algorithm to determine the smallest possible buffer size for arbitrary SDF applications [1]. Though their work is mainly targeted for mapping an SDF application onto a Field Programmable Gate Array (FPGA) in the GRAPE environment, the computed lower bound on the buffer requirement is applicable to software synthesis. Govindarajan et al. [7] developed a rate optimal compile time schedule, which minimizes the buffer requirement by using linear programming formulation. Since the resultant schedule will not be an SA-schedule in general, a function coding style should be used to minimize the code size in the generated code.

No previous work exists that considers all design factors such as coding styles, buffer sharing, and code sharing. In spite of extensive prior research efforts, finding an optimal schedule that minimizes the total memory requirement still remains an open problem, even for single processor implementation.

2.3.2 Scheduling Techniques for Multiprocessor Implementations

A key scheduling objective for multiprocessor implementation is to reduce the execution length or to maximize the throughput of a given SDF graph. While there are numerous techniques developed for multiprocessor scheduling, they usually assume a single rate dataflow graph where each node is executed only once in a single iteration. And they primarily focus on exploiting the functional parallelism of an application to minimize the length of the schedule, called *makespan*. In stream-based applications, however, maximizing the throughput is more important than minimizing the schedule length. Pipelining is a popular way of improving the throughput of a dataflow graph. For example Hoang et al. have proposed a pipelined mapping/scheduling technique based on a list scheduling heuristic [8]. They maximize the throughput of a homogeneous dataflow graph on a homogeneous multi-processor architecture.

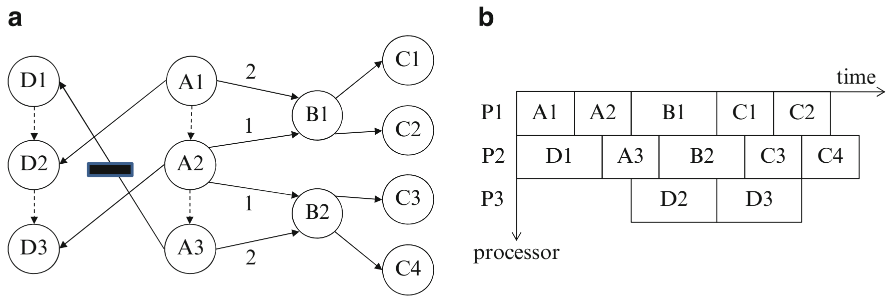


Fig. 6 (**a**) An APEG (acyclic precedence expanded graph) of the SDF graph in Fig. 2a, and (**b**) a parallel scheduling result displayed with a Gantt chart

To apply these techniques for an SDF graph directly, we need to translate an SDF graph to a single rate task graph, called an APEG (Acyclic Precedence Expanded Graph) or simply EG (Expanded Graph) [17]. A node of an SDF graph is expanded to as many nodes in the EG as the repetition counts of the node. The corresponding EG of the graph of Fig. 2a is shown in Fig. 6a where nodes A and D are expanded to three invocations respectively and node B to two and node C to four invocations. The number of samples communicated through each arc is unity unless specified otherwise; if an arc is annotated with a non-unity sample rate, such as an arc between nodes A1 and B2, the arc can be split into as many uni-rate arcs as the number to make a single-rate dataflow graph. If a node has any internal state, dependency arcs should be added between node invocations: in the figure, we assume that nodes A and D have internal states and the dependency between invocations is represented by dashed arcs. Note that the initial sample on arc AD is placed on the arc between A3 and D1. Since the initial sample breaks the execution dependency between A3 and D1 in the same iteration, D1 can be executed before A3.

One approach to schedule an SDF graph to a multiprocessor architecture is to generate an APEG and apply an existent multi-processor scheduling algorithm. Figure 6b shows a parallel scheduling result with a Gantt chart where the vertical axis represents the processing elements of the target system and the horizontal axis represents the elapsed time. There are some issues worth noting in this approach. First, loop-level data parallelism in an SDF graph is translated into functional parallelism in the EG. Nodes B and C in Fig. 2a express data-level parallelism since multiple invocations can be executed in parallel. But all invocations are translated into separate nodes that can be scheduled independently, ignoring the loop structure, in the EG. As a result, the same block can appear several times in the schedule, which may incur significant code size overhead if inline coding style is used. While it is a reasonable way to exploit the loop-level data parallelism, it may result in a very expensive solution.

Second, the total number of nodes in the EG is the sum of all repetition counts of the nodes. A simple SDF graph with non-trivial sample rate changes can be translated to a huge EG. Therefore the algorithm complexity of a multiprocessor scheduling technique should be low enough to scale well as the graph size increases.

Third, multiple invocations of the same node are likely to be mapped to different processors. If a node has internal states (for instance node D in Fig. 6b), the internal states should be transferred between invocations, which incurs significant run-time overhead. And additional code should be inserted to manage the internal states in the generated code.

Therefore several parallel scheduling techniques have been proposed that work with the SDF graph directly without APEG generation. A node with internal states is constrained to be mapped to a single processor. A recent approach considers functional parallelism, loop-level parallelism, and pipelining simultaneously to minimize the throughput of the graph [24]. In this work, a node can be mapped to multiple processors if the kernel code of a node has internal data parallelism.

Unlike single processor implementation, there is a trade-off between buffer size and throughput in multi-processor implementation of SDF graphs. Stuijk et al. have explored the trade-off and obtained the Pareto-optimal solutions in terms of the buffer size and the throughput, assuming that there is no constraint for the number of processors [21]. This work is extended to consider resource constraints in [20]. Other extensions can be found in a web-site (http://www.es.ele.tue.nl/sdf3/) that they manage to make the proposed technique open to public under the name of SDF3 (SDF for Free) [21].

While most work on multiprocessor scheduling assumes static scheduling, some recent researches consider static mapping and dynamic scheduling as a viable implementation technique of SDF task graphs. This approach maps nodes to processors at compile time and delegates the scheduling decision to a run-time scheduler on each processor. A main benefit of dynamic scheduling is that it may tolerate large variation of node execution times or communication delays. When we make a static schedule, we have to assume a fixed node execution time and a fixed communication delay; worst-case values are usually assumed for real-time systems. If the worst-case values are assumed, however, a processor may be idle while the current scheduled node is waiting for the arrival of input data from the other processors even though there are other executable nodes. Thus static scheduling may result in waste of resources while dynamic scheduling changes the execution sequence of nodes to increase resource utilization. A key issue for dynamic scheduling is how to assign priorities to the mapped nodes on each processor. There is a recent work to find an optimal static mapping and priority assignment to minimize the resource requirement under a throughput constraint [10].

If the execution time of each function block is fixed and known at compile-time, we can estimate the total execution length of the SDF graph. Hence the SDF graph is suitable for specifying an application that has real-time constraints on throughput or latency. When the execution time of a node varies at run-time, the worst case execution time (WCET) should be used in parallel scheduling in order to guarantee satisfaction of real-time constraints. Unfortunately, however, using WCETs is not enough to guarantee the schedulability if an application consists of multiple tasks running concurrently. If the execution time of a node becomes smaller than its WCET, the order of node firings may vary at run-time, which may lengthen the total execution time of the application. This behavior is known as "scheduling anomaly" of multiprocessor scheduling; If multiple tasks are running concurrently, resource contention may lengthen the execution of the graph when a node takes shorter time than its worst case execution time.

Since many DSP applications involve multiple tasks running concurrently, we need to consider multi-tasking in parallel scheduling. One solution is to statically schedule the multiple tasks up to their hyper-period that is defined as a least common multiple of all task periods. Since the hyper-period can be huge if the task periods are relatively prime, this approach may not be applicable. Multiprocessor scheduling of multi-tasking applications on heterogeneous multiprocessor systems still remains an open problem. With a given schedule, it is not yet possible to decide whether the schedule can satisfy the real-time constraints if some nodes have varying execution times and/or there are resource contentions during execution.

2.4 Hardware Synthesis from SDF Graph

While the target architecture is given as a constraint for software synthesis, the target hardware structure can be synthesized in hardware synthesis from an SDF graph. Therefore, we can achieve the *iteration bound* of an SDF graph in theory (see [15] to find the definition of the iteration bound of a graph) if there is no limitation on the hardware size. Since there is a trade-off between hardware cost and the throughput performance, however, architecture design and node scheduling should be considered simultaneously under given design constraints.

A key issue in hardware synthesis is to preserve the SDF semantics to maintain the correctness of the graph. In the SDF model, two samples that have the same value should be distinguished as separate samples while the same value is not identified as a new event in a hardware logic. So the arrival of an input sample should be notified somehow. And if a node has more than one input port, the node should wait until all input ports receive data samples before the node starts execution. It means that we need some control logic to perform scheduling of the nodes. There are two types of controllers: distributed controller and centralized controller. In the centralized control scheme, the execution timing of each node is controlled by a central scheduler. The execution timing can be determined at compile-time by static scheduling of the graph. In a distributed scheme, a node is associated with a control logic that monitors the input queues and triggers the node execution when all input queues have input samples to fire the node.

For hardware synthesis, a node should be specified by a hardware description language that will be synthesized by a CAD tool, or by a function block that is mapped to a pre-defined hardware IP. If an hardware IP is used, interface between the IP and the rest of the system should be designed carefully. Since the interface design is a laborious and error-prone task, extensive researches are being performed on the automatic interface synthesis.

In summary, hardware synthesis from a SDF graph involves the following problems: architecture and datapath synthesis, controller synthesis, and interface synthesis. The node granularity in a SDF graph also affects the hardware synthesis procedure. Various issues in hardware synthesis for a coarse grained graph is discussed in [9]. For FPGA synthesis from a fine-grained graph, see [23] for more detailed information.

3 Cyclo-Static Dataflow

The strict restriction of the SDF model, that all sample rates are constant, limits the expression capability of the model. Figure 7a shows a simple SDF graph that models a stereo audio processing application where the samples for the left and right channels are interleaved at the source. The interleaved samples are distributed by node D to the left (node L) and to the right (node R) channel. In this example, the distributor node (node D) waits until two samples are accumulated on its input arc to produce one sample at each output arc. A more natural implementation would be to make the distribution node route an input sample to two output ports alternatively at each arrival. A useful generalization of the SDF model, called the cyclo-static dataflow (CSDF), makes it possible [4].

In a cyclo-static dataflow graph, the sample rates of an input or output port may vary in a periodic fashion. Figure 7b shows how the CSDF model can specify the same application as Fig. 7a. To specify the periodic change of the sample rates, a tuple rather than an integer is annotated at the output ports of node D'. For instance, "$\{1{,}0\}$" on arc D'R denotes that the rate change pattern is repeated every other execution, where the rate is 1 at the first execution, and 0 at the second execution. Similarly, the periodic sample rate "$\{0{,}1\}$" means that the rate is 0 at every $(2n+1)$-th iteration and 1 at the other iterations.

Note that Fig. 7a, b represent the same application in functionality. One firing of node D in the SDF graph is broken down into two firings of node D' in the CSDF graph. Thus we have to split the behavior of node D' into phases. The number of phases is determined by the periods of the sample rate patterns of all input and output ports. In general, we can convert a CSDF graph to an *equivalent* SDF graph by merging as many firings of a CSDF node as the number of phases into a single firing of an equivalent SDF node. For instance, node D' in the CSDF graph repeats its behavior every two firings, and the number of phases becomes 2. So an equivalent

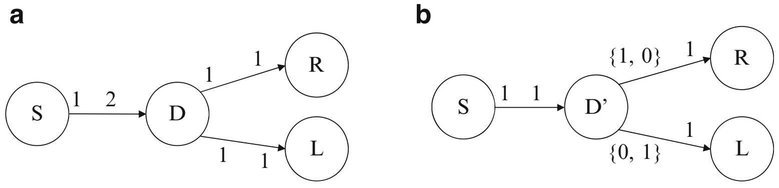


Fig. 7 (**a**) An SDF graph where node D is a distributor block and (**b**) a CSDF graph that shows the same behavior with a different version of a distributor block

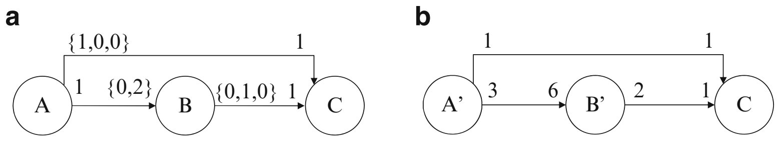


Fig. 8 (**a**) A cyclo-static dataflow graph and (**b**) its corresponding SDF graph

SDF node can be constructed by merging two firings of node D' into one, which is node D in the SDF model. The sample rates of input and output ports are adjusted accordingly by summing up the number of samples consumed and produced during one cycle of periodic behavior.

The CSDF model has a big advantage over the SDF model in that it can reduce the buffer requirement on the arcs. In the example shown in Fig. 7, the minimum size of input buffer for node D should be 2 in the SDF model while it is 1 in the CSDF model.

3.1 Static Analysis

Since we can construct an equivalent SDF graph, static analysis and scheduling algorithms developed for SDF are also applicable to CSDF. For formal treatment, we use vectors to represent the periodic sample rates in CSDF: for an arc e, the output sample rate vector of $src(e)$ and the input sample rate vector of $snk(e)$ are denoted by $\mathbf{prod}(\mathbf{e})$ and $\mathbf{cons}(\mathbf{e})$ in CSDF. Figure 8a shows another CSDF graph that has non-trivial periodic patterns of sample rates. The sample rate vectors for the graph become the following:

$$\mathbf{prod}(\mathbf{AC}) = (1,0,0), \mathbf{cons}(\mathbf{AB}) = (0,2), \mathbf{prod}(\mathbf{BC}) = (0,1,0),$$
$$\mathbf{prod}(\mathbf{AB}) = \mathbf{cons}(\mathbf{BC}) = \mathbf{cons}(\mathbf{AC}) = (1).$$

To make an equivalent SDF node for each CSDF node, we have to compute the repetition period for the phased operation of the CSDF node. First we obtain the period of the sample rate variation for each port, which is the size of the sample rate vector. Let dim($\mathbf{v}$) be the dimension of the sample rate vector $\mathbf{v}$. Then the repetition period of a CSDF node A, denoted by $p(A)$ becomes the least common multiple (lcm) value of all dim($\mathbf{v}$) values for the input and output ports of the node. For the example of Fig. 8a, the repetition periods become the following:

$$p(A) = lcm(dim(\mathbf{prod}(\mathbf{AB})), dim(\mathbf{prod}(\mathbf{AC}))) = lcm(3,1) = 3.$$
$$p(B) = lcm(dim(\mathbf{cons}(\mathbf{AB})), dim(\mathbf{prod}(\mathbf{BC}))) = lcm(2,3) = 6.$$
$$p(C) = lcm(dim(\mathbf{cons}(\mathbf{AC})), dim(\mathbf{prod}(\mathbf{BC}))) = lcm(1,1) = 1.$$

If $p(A)$ firings of CSDF node A are merged into a single firing, an equivalent SDF actor A' is obtained. Hence the equivalent SDF graph is obtained as shown in Fig. 8b where node B' is obtained by merging six firings of node B in the CSDF graph. We denote this equivalence relation as $B' \approx 6B$. For an equivalent SDF node, the scalar sample rate of a port should be determined. Let $\sigma(\mathbf{v})$ be the sum of elements in vector $\mathbf{v}$. The total number of samples produced or consumed on arc e of CSDF node A per the corresponding SDF node execution is given by $p(A)\frac{\sigma(\mathbf{prod}(\mathbf{e}))}{dim(\mathbf{prod}(\mathbf{e}))}$ or $p(A)\frac{\sigma(\mathbf{cons}(\mathbf{e}))}{dim(\mathbf{cons}(\mathbf{e}))}$. So, we can construct a topology matrix for the equivalent SDF graph as follows:

$$\varGamma(e,A) = \begin{cases} p(A)\frac{\sigma(\mathbf{prod}(\mathbf{e}))}{dim(\mathbf{prod}(\mathbf{e}))}, & \text{if } A = src(e) \\ -p(A)\frac{\sigma(\mathbf{cons}(\mathbf{e}))}{dim(\mathbf{cons}(\mathbf{e}))}, & \text{if } A = snk(e) \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

We can check the sample rate consistency with this topology matrix. For the graph in Fig. 8b, the topology matrix and the repetition vector become:

$$\varGamma = \begin{pmatrix} 3 & -6 & 0 \\ 1 & 0 & -1 \\ 0 & 2 & -1 \end{pmatrix}$$

$$\mathbf{q_G} = (2, 1, 2)$$

Since rank of $\varGamma$ is 2, the CSDF graph is sample rate consistent. Moreover, a valid schedule includes two invocations of nodes A' and C, and one invocation of node B'. This means that a valid CSDF schedule contains 6A, 6B and 2C since $A' \approx 3A$ and $B' \approx 6B$. The deadlock detection algorithm for an SDF graph in Sect. 2.1 is applicable for a CSDF graph, which is to construct a static schedule by simulating the graph.

3.2 Static Scheduling and Buffer Size Reduction

One strategy of scheduling a CSDF graph is to schedule the equivalent SDF graph and replace the execution of the equivalent node with the multiple invocations of the original CSDF node. We can obtain the following schedule for the graph in Fig. 8: $\varSigma 1 = 2A'B'2C = 6A6B2C$. Then the minimum buffer requirement on arc AB becomes 6. We can construct better schedules in terms of buffer requirements by utilizing the phased operation of a CSDF node. For the case of CSDF graph of Fig. 8a, we can construct a better schedule as follows.

1. Initially nodes A and B are fireable, so schedule nodes A and B: $\varSigma 2$ = "AB".
2. Since node A is the only fireable node, we schedule node A again: $\varSigma 2$ = "ABA"

3. Now two samples are accumulated on arc AB and the second phased of node B can start. So schedule node B: $\Sigma 2$ = “ABAB”.
4. Node C becomes fireable. Schedule node C for the first time: $\Sigma 2$ = “ABABC”.
5. We can schedule nodes A and B twice: $\Sigma 2$ = “ABABCABAB”
6. At this moment, only one sample is stored on arc AC and we can fire nodes A and B. We choose to schedule the fifth invocation of node B to produce one sample on arc BC. $\Sigma 2$ = “ABABCABABB”
7. Then, node C becomes fireable. Schedule node C: $\Sigma 2$ = “ABABCABABBC”
8. Finally we schedule node A twice and node B once to complete one iteration of the schedule: $\Sigma 2$ = “ABABCABABBCAAB”
9. Since schedule $\Sigma 2$ contains 6A, 6B and 2C, scheduling is finished and no deadlock is detected.

Schedule $\Sigma 2$ requires two buffers on arc AB, which is three times better than schedule $\Sigma 1$. Generally, as sample rates vary more, the buffer size reduction becomes more significant. This gain is obtained by splitting the CSDF node into multiple phases. But we have to pay the overhead of code size since a single appearance schedule is given up. In general, there are more valid schedules for a CSDF graph than for the equivalent SDF model. Therefore, discussion on the SDF scheduling can be applied to the CSDF model, but with increased complexity of scheduling problems.

3.3 *Hierarchical Composition*

Another advantage of CSDF is that it offers more opportunities of clustering when constructing a hierarchical graph. It also allows a seemingly delay-free cycle of nodes, while no delay-free cycle is allowed in SDF. Figure 9a shows an SDF graph with four nodes A, B, C and D. All sample rates are unity since no sample rate is annotated on any arc. The graph can be scheduled without deadlock since there is an initial delay sample between nodes A and B. One unique valid schedule of this graph is “BCDA”. Suppose we cluster nodes A and B into an hierarchical node W in CSDF and W’ in SDF as illustrated in Fig. 9a and b respectively. Since CSDF node W fires node B at every $(2n+1)$-th cycle and node A at every $2n$-th cycle, the input and the output sample rate vectors of node W become “$\{0,1\}$” and “$\{1,0\}$” respectively. Therefore, the CSDF graph can be scheduled without deadlock and a valid static schedule is “WCDW” where node B is fired at the first invocation of node W and node A is fired at the second invocation. On the other hand, SDF node W’ should execute both nodes A and B when it is fired. Therefore, the SDF graph as shown in Fig. 9b is deadlocked since nodes W’, D, and C wait for each other.

Clustering of nodes may cause deadlock in SDF even though the original SDF graph is consistent. On the other hand, a CSDF graph that includes a cyclic loop without an initial delay can be scheduled without deadlock if the periodic rates are carefully determined. Therefore, the CSDF model allows more freedom of hierarchical composition of the graph.

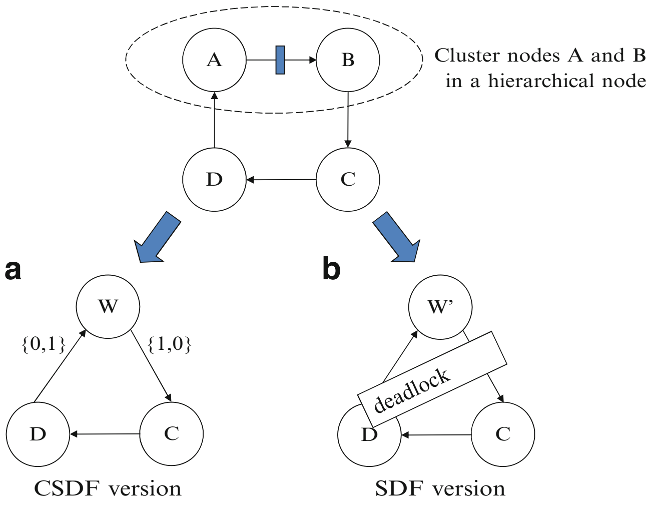


Fig. 9 Clustering of nodes A and B into an hierarchical node in (**a**) CSDF and (**b**) in SDF

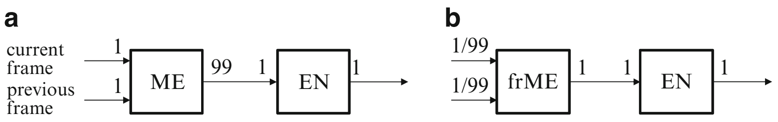


Fig. 10 A subgraph of an H.263 encoder graph (**a**) in SDF and (**b**) in FRDF

4 Other Decidable Dataflow Models

4.1 Fractional Rate Dataflow

The SDF model does not make any assumption on the data types of samples as long as the same data types are used between two communicating nodes. To specify multimedia applications or frame-based signal processing applications, it is natural to use composite data types such as video frames or network packets. If a composite data type is assumed, the buffer requirement for a single data sample can be huge, which amounts to 176×144 pixels for a QCIF video frame for instance. Then reducing the buffer requirement becomes more important than reducing the code size when we make a schedule.

Figure 10a shows an SDF subgraph of an H.263 encoder algorithm for QCIF video frames. A QCIF video frame consists of 11×9 macroblocks whose size is 16×16 pixels. Node ME that performs motion estimation consumes the current and the previous frames as inputs. Internally, the ME block divides the current frame into 99 macroblocks and computes the motion vectors and the pixel differences from the previous frame. And it produces 99 output samples at once where each output

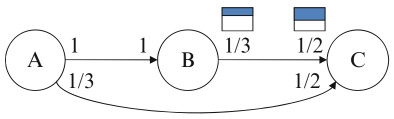


Fig. 11 An FRDF graph in which sample types on arc BC and AC are a composite and a primitive type respectively

sample is a macroblock-size data that represents a 16×16 array of pixel differences. Node EN performs macroblock encoding by consuming one macroblock at a time and produces one encoded macroblock as its output sample.

This SDF representation is not efficient in terms of buffer requirement and performance. Since node ME produces 99 macroblock-size samples at once after consuming a single frame size sample at each invocation, we need a 99-macroblock-size buffer or a frame-size buffer ($99 \times 16 \times 16 = 176 \times 144$) to store the samples on the arc between nodes ME and EN. Moreover node EN cannot start execution before node ME finishes motion estimation for the whole input frame. As this example demonstrates, the SDF model has inherent difficulty of efficiently expressing the mixture of a composite data type and its constituents: a video frame and macroblocks in this example. A video frame is regarded as a unit of data sample in integer rate dataflow graphs, and should be broken down into multiple macroblocks explicitly by consuming extra memory space.

To overcome this difficulty, the fractional rate dataflow (FRDF) model in which a fractional number of samples can be produced or consumed has been proposed [14]. In FRDF, a fraction number can be used as a sample rate as shown in Fig. 10b where the input sample rates of node frME is set to $\frac{1}{99}$. The fractional number means that the input data type of node frME is a macroblock and it corresponds to $\frac{1}{99}$ of a frame data.

Figure 11 shows an FRDF graph where the data type of arc BC is a composite type as illustrated in the figure and the data type of arc AC is a primitive type such as integer or float. A fractional sample rate has different meaning for a composite data type from a primitive type. For a composite data type, the fractional sample rate really indicates the partial production or consumption of the sample. In the example graph, one firing of node B fills $\frac{1}{3}$ of a sample on arc BC and node C reads the first half of the sample at every $(2n+1)$-th firing and the second half at every $2n$-th firing. Hence, if we consider the execution order of nodes B and C, a schedule "BBCBC" is valid since $\frac{2}{3}$ of a sample is available after node B is fired twice and node C becomes fireable.

For primitive types, partial production or consumption is not feasible. Then statistical interpretation is applied for a fractional rate. In the example graph, the output sample rate of node A is $\frac{1}{3}$ on arc AC. This means that node A produces a single sample every three executions. Similarly node C consumes one sample every two executions. Note that a fractional rate does not imply at which firings samples are produced. So node C becomes fireable only after node A is executed three times. If we are concerned about the execution order of nodes A and C only,

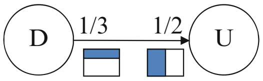


Fig. 12 If the consumer and the producer have the different access patterns for a composite type data then the type should be treated as atomic

schedule “3A2C” is valid while “2ACAC” is not. Consequently, a valid schedule for the FRDF graph of Fig. 11 is “3(AB)2C”.

Regardless of the data type, a fractional sample rate $\frac{p}{q}$ guarantees that p samples are produced or consumed after q firings of the node. Similar to the CSDF graph, we can construct an equivalent SDF graph by merging q firings of an FRDF node into an equivalent SDF node that produces or consumes p samples per firing. Therefore, static analysis techniques for the SDF model can be applied to the FRDF model. For the analysis of sample rate consistency, however, we can use fractional sample rates directly in the topology matrix. For the FRDF graph of Fig. 11, the topology matrix and repetition vector are:

$$\Gamma = \begin{pmatrix} 1 & -1 & 0 \\ 0 & \frac{1}{3} & -\frac{1}{2} \\ \frac{1}{3} & 0 & -\frac{1}{2} \end{pmatrix}$$

$$\mathbf{q_G} = (3,3,2)$$

Since the rank of Γ is 2, the graph is sample rate consistent. Deadlock can be detected by constructing a static schedule similarly to the SDF case. If there exists a valid static schedule, the graph is free from deadlock. A static schedule is simply constructed by inserting a fireable node into the schedule list and simulating its firing. An FRDF node has different firing conditions depending on the data types of input ports. An FRDF node is fireable, or executable, if all input arcs satisfy the following condition depending on the data type:

1. If the data type is primitive, there must be at least as many stored samples as the numerator value of the fractional sample rate. An integer sample rate is a special fractional rate whose denominator is 1.
2. If the data type is composite, there must be at least as large a fraction of samples stored as the fractional input sample rate.

Special care should be taken for a composite type data. If the consumer and the producer have a different interpretation on the fraction, then a composite type should be regarded as atomic like a primitive type when the firing condition is examined. Suppose that for a composite data type of a two-dimensional array, the producer regards it as an array of row vectors while the consumer regards it as an array of column vectors as shown in Fig. 12. In this case, the two-dimensional array may not be regarded as a composite type data. Therefore, schedule “DDUDU” is not valid while “3D 2U” is.

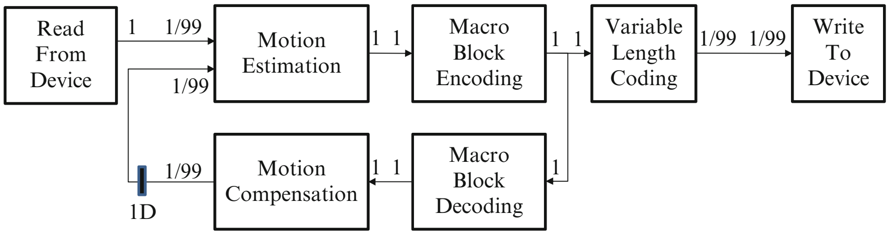
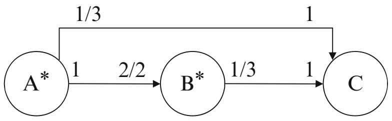


Fig. 13 H.263 encoder in FRDF

Fig. 14 An FRDF graph corresponding to Fig. 8



In general, the FRDF model results in an efficient implementation of a multimedia application in terms of buffer requirement and performance. Consider the example in Fig. 10b again. Since node frME uses a macroblock-size sample, the output arc requires only a single macroblock-size buffer. For each arrival of an input video frame, node frME is fired 99 times and consumes a macro-block size portion of the input frame per firing. Since node EN can be fired after each firing of node frME, shorter latency is experienced when compared with the SDF implementation. Figure 13 shows a whole H.263 encoding algorithm in FRDF where the sample types for "current frame" and "previous frame" are video frames. It is worth noting that the entire previous frame is needed to do motion estimation for each macroblock of the current frame while the sample rate of the bottom input port of node "Motion Estimation" is $\frac{1}{99}$. Hence, even though the previous frame is a composite data type, it should be regarded as atomic. Then node "Motion Estimation" is fireable only after the entire previous frame is available.

Figure 14 shows an FRDF graph that corresponds with Fig. 8a. Both have the same equivalent SDF graphs. Similar to the CSDF model, the FRDF model can reduce the buffer size when compared with the corresponding SDF model. Since node A^* produces a single sample and B^* consumes two samples every other execution, a valid schedule for the graph is "$2A^*$ $2B^*$ $2A^*$ B^* C B^* $2A^*$ $2B^*$ C". And the required buffer sizes on arcs A^*B^* and B^*C^* are equal to the sizes for the CSDF graph. The buffer size for arc A^*C is, however, larger than the CSDF graph since the FRDF model does not know when samples are produced and consumed, and the schedule for the FRDF model should consider the worst case behavior. For an output port, the worst case is when output samples are all produced at the last phase while it is when input samples are all consumed at the first phase for an input port. Therefore, in the FRDF model, rate $\frac{p}{q}$ for a primitive data type corresponds to "$\{(q-1) \times 0,\text{p}\}$" for an output sample rate and "$\{\text{p},(q-1) \times 0\}$" for an input sample rate in the CSDF model, where "$(q-1) \times 0$" denotes "$\underbrace{0,0,\cdots,0}_{q-1}$". Hence, the CSDF model may generate better schedules than the associated FRDF model since we can split the node execution into finer granularity of phases at compile-time scheduling; this is not possible in the FRDF if the date type is primitive. The expression capability of two models is, however, different. The CSDF model allows only a periodic pattern to express sample rate variations while the FRDF model has no such restriction as long as the average value is constant during a period. So the FRDF model has more expression power than the CSDF model since it allows dynamic behavior of an FRDF node within a period and the periodic pattern can be regarded as a special case.

4.2 Synchronous Piggybacked Dataflow

The SDF model does not allow communication through a shared global variable since the access order to the global variable can vary depending on the execution order of nodes. Suppose a source block produces the frame header in a frame-based signal processing application that is to be used by several downstream blocks. A natural way of coding in C is to define a shared data structure that the downstream blocks access by pointers. But in a dataflow model, such sharing is not allowed. As a result, redundant copies of data samples are usually introduced in the automatically generated code from the dataflow model. Such overhead is usually not tolerable for embedded systems with tight resource and/or timing constraints. To overcome this limitation, an extended SDF model, called SPDF (Synchronous Piggybacked Dataflow) is proposed [16], by introducing the notion of "controlled global states" and by coupling a data sample with a pointer to the controlled global state.

The Synchronous Piggybacked Dataflow (SPDF) model was first proposed to support frame-based processing, or block-based processing, that frequently occurs in multimedia applications. In frame-based processing, the system receives input streams of frames that consist of a frame header and data samples. The frame header contains information on how to process data samples. So an intuitive implementation is to store the information in a global data structure, called *global states*, and the data processing blocks refer to the global states before processing the data samples.

Figure 15 shows a simple SPDF graph where node A reads a frame from a file and produces the header information and the data samples through different output ports. Suppose that a frame consists of 100 data samples in this example. The header information and the data samples are both connected to a special FRDF (Fractional Rate Dataflow) block, called *"Piggyback"* block, that has three parameters: *"statename"*, *"period"*, and *"offset"*. The Piggyback block updates the global state of *"statename"* with the received header information periodically with the specified *"period"* parameter. It piggybacks a pointer to the global state on each input data sample before sending it through the output port. Since it needs to receive the header information in order to update the global state only once per 100 executions, the sample rate of the input port associated with the header information

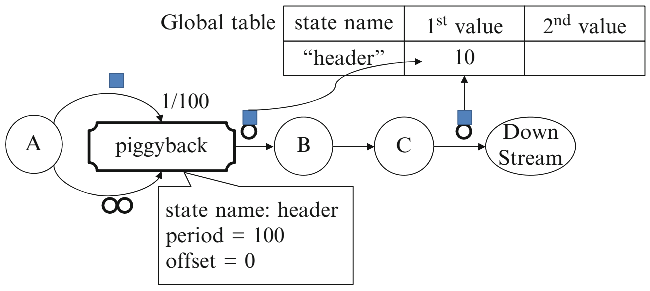


Fig. 15 An example SPDF graph that shows a typical frame-based processing: the Piggyback block writes the header information to the global state and the downstream blocks refer to the global state before processing the data samples

is set to the fractional value $\frac{1}{100}$, which means that it consumes one sample per 100 executions in the FRDF model. The input port of this fractional rate is called the *"state port"* of the Piggyback block. The sample rate of the data port, on the other hand, is unity.

The *"offset"* parameter indicates when to update the global state. The Piggyback block receives as many data samples as the *"offset"* value before updating the global state. In this example, the *"offset"* value is set to its default value, zero, which makes the Piggyback block consume the header information and update the global state before it piggybacks the data samples with a pointer to the global state.

Note that the Piggyback block with a fractional rate input port is the only extension to the SDF model. Since the sample rates of the SPDF graph are all static, the static analyzability of the SDF model is preserved even after addition of the Piggyback block. Also, piggybacking of data samples with pointers can be performed without any run-time overhead by utilizing the static schedule information of the graph. Suppose that the SDF graph in Fig. 15 has the following static schedule: "A 100(Piggyback, B, C, DownStream)". Then the pseudo code of the automatically generated code associated with the schedule is as follows:

```
code block of A
for (int i = 0; i < 100, i++) {
  if (i == offset_Piggyback)
    update the global state header
  code block of B
  code block of C
  if (i == offset_Piggyback)
    update the local state of DownSteam block
      from the global state information
  code block of DownStream
}
```

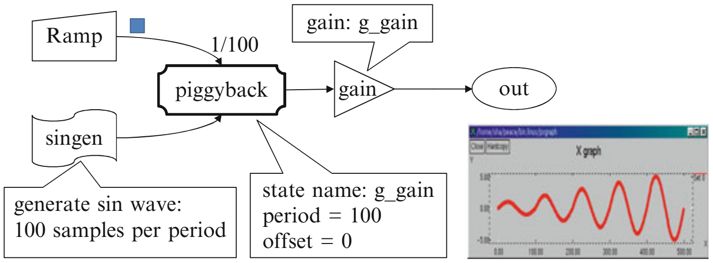


Fig. 16 An SPDF graph that produces a sinusoidal waveform with varying amplitude at run-time: the "gain" state of the "Gain" block is updated by the "Ramp" block through a global state, "global_gain"

Figure 16 shows another example that produces a sinusoidal waveform with varying amplitude at run-time. The "Singen" block generates a sinusoidal waveform (N samples per period) of unit amplitude and the "Gain" block amplifies the input samples by the "gain" parameter of the block. To control the amplitude, the graph uses a Piggyback block after the "Singen" block. Another source block, "Ramp", is connected to the state port of the Piggyback block. The "statename" of the Piggyback is named "global_gain" and the "gain" parameter of the "Gain" block is also set to "global_gain". Then, the "gain" parameter of the "Gain" block is updated with a global state named by "global_gain" whose value is determined by the "Ramp" block. In this example, the period of the Piggyback block is set to N so that the amplitude of the sinusoidal waveform is incremented by one every period as shown in Fig. 16. If we insert two initial samples on the input arc of the "Gain" block, the "offset" parameter of the Piggyback block should be 2.

Thus the SPDF model provides a safe mechanism to deliver state values through shared memory instead of message passing. Communication through shared memory is prohibited in conventional dataflow models since the access order to the shared memory may vary depending on the schedule. But the SPDF model gives another solution by observing that the side effect is caused by an implicit assumption that the global state is assigned a memory location before scheduling is performed. The SPDF model changes the order: allocate the memory for the global state after the schedule is made. Since the scheduling decision is made at compile-time, we know the access order to the variable and the life time of each global state variable. Suppose that the schedule of Fig. 16 becomes "2(100(Singen) Ramp Piggyback) 200(Gain Display)". From the static schedule, we know that we need to maintain two memory locations for the global state, "global_gain" since the Piggyback block writes the second value to the global state before the "Gain" reads the first global state.

Allowing shared memory communication without side effects gives a couple of significant benefits over conventional dataflow models. First, it can remove the unnecessary overhead of data copying of message passing communication since the global state can be shared by multiple blocks. Second, it greatly enhances the

a

```
void add()
{
   *addOut = *addIn1 + *addIn2;
}
```

b

```
void add()
{
   for(int i=0; i<Nb; i++)
     *addOut++ = *addIn1++ + *addIn2++;
}
```

Fig. 17 Code of an “Add” actor (**a**) in SDF and (**b**) in SSDF where N_b is the blocking factor

expression capability of the SDF model without breaking the static analyzability. It provides an explicit mechanism of affecting the internal behavior of a block from the outside through global states.

4.3 Scalable Synchronous Dataflow

DSP architectures have stringent on-chip memory limits and off-chip memory access is costly. They also allow vector processing of instructions and arithmetic pipelining like MAC in order to attain peak performance when the pipelining is fully utilized. Therefore, when an SDF block operates on primitive-type data and the granularity is small, it behaves inefficiently. For example, an adder block performs a single accumulation operation by reading two samples from memory and writing a sample into memory. It requires large run time overhead of off-chip memory access for two read and one write operations. In order to achieve efficient implementation, the scalable synchronous dataflow (SSDF) model is proposed [18]. The SSDF model has the same semantics as the SDF model except that a node may consume or produce any integer multiple of the fixed rates per invocation. The actual multiple, called *blocking factor*, is determined by a scheduler or an optimizer.

Figure 17a shows the code of an “Add” block in SDF. In SSDF, the code includes blocking factor N_b that is the number of iterations as shown in Fig. 17b. Since the function call overhead of “add()” is larger than the accumulation operation, the SSDF model amortizes the function call overhead by performing N_b accumulations per function call. When blocking factor N_b is 1, the SSDF graph degenerates to an SDF graph. Therefore, the SSDF model has the same sample rates as the SDF model and sample rate inconsistency can be checked using the topology matrix for the SDF model. Moreover, deadlock is detected by constructing a schedule by setting block factor $N_b = 1$. From the static analysis of the degenerated SDF graph, repetition vector q_G can be obtained, assuming N_b is 1 for all blocks. When $N_b > 1$, the repetition vector for SSDF becomes $N_b q_G$.

A straightforward scheduling technique for an SSDF graph is to increase the minimal scheduling period by an integer factor N_g where N_g is a global blocking factor. Each node A of the graph will be invoked $N_g x(A)$ times within one scheduling period, where $x(A)$ is the repetition count of node A. Increasing N_g reduces the function call overhead but requires larger buffer memory for graph execution. For

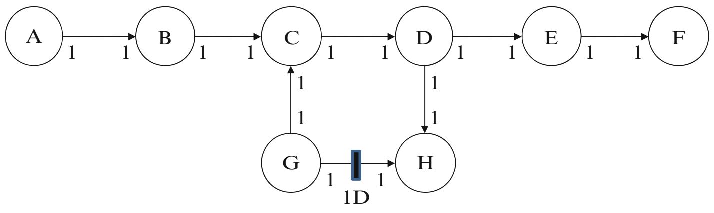


Fig. 18 A graph with feedback loop

instance, the "Add" node in Fig. 17 consumes N_b samples from each input port and produces N_b output samples, then all three buffers have size N_b while they have size 1 when the blocking factor is unity. Moreover, the increment of N_g delays the response time although it does not decrease the throughput.

Another major obstacle to increase the blocking factor is related with feedback loops. Vector processing is restricted to the number of initial delays on the feedback loop. If the number is smaller than N_g, the vector processing capability cannot be fully utilized. For example, the scheduling result for a graph shown in Fig. 18 is "A B G C D H E F" when the blocking factor is 1. If blocking factor N_g becomes 5 then the scheduling becomes "5A 5B 5(GCDH) 5E 5F" in which nodes G, C, D and H are repeated five times sequentially. Therefore, a scheduling algorithm for SSDF should consider the graph topology to minimize the program code size.

In case feedback loops exist, strongly-connected components are first clustered into a strong component. A *strong component* of graph G is defined as a subgraph $F \subset G$ if for all pairs of nodes $u, v \in F$ there exist paths p_{uv}(from u to v) and p_{vu}(from v to u). This clustering is performed in a hierarchical fashion until the top graph does not have any feedback loop. Then, a valid schedule for an SSDF graph can be constructed using the SDF scheduling algorithms. Each node is scheduled by applying the global blocking factor N_g. For the SSDF graph in Fig. 18, the top graph consists of five nodes "A B (CDGH) E F" where nodes C, D, G and H are merged into a clustered-node. When blocking factor N_g is set to 5, a schedule for the top graph becomes "5A5B5(clustered-node)5E5F".

Next, the strong components are scheduled. The blocking factor depends on the number of initial delay samples on a feedback loop. Let $N_l(L)$ denote the maximum bound of the blocking factor on feedback loop L. Since feedback loops can be nested, a feedback loop with the largest maximum bound $N_l(L)$ should be selected first. Subsequently, feedback loops are selected in a descending order of $N_l(L)$. Scheduling of the clustered subgraph starts with a node that has many initial delay samples on its input ports and allows a large blocking factor. When a strong component "(CDHG)" is scheduled in the SSDF graph, actor G should be fired since it has an initial delay sample.

For a selected strong component, we schedule the internal nodes as follows, depending on the number of delays on the feedback loop.

Case 1: N_g is an integer multiple of $N_l(L)$. The scheduling order needs to be repeated $N_g/N_l(L)$ times using $N_b = N_l(L)$ for the internal nodes. In the example of Fig. 18, since $N_l(L) = 1$, $N_g = 5$, and $N_g/N_l(L)$ is an integer, schedule of "GCDH" is repeated five times. Moreover, the blocking factor for each node N_b is 1. Hence, the final schedule is "5A 5B 5(GCDH) 5E 5F".

Case 2: $N_g \leq N_l(L)$. Blocking factor $N_b = N_g$ is applied for all actors in the strong component. For example, if the number of delay samples increases to 5 in Fig. 18, then blocking factor $N_l(L)$ is 5 which is equal to N_g, and the schedule becomes "5A 5B (5G 5C 5D 5H) 5E 5F". Therefore, the blocking factor can be fully utilized.

Case 3: If $N_g > N_l(L)$ but not an integer multiple. One of two scheduling strategies can be applied:

1. The schedule for the strong component is repeated N_g times using $N_b = 1$ internally, which produces the smallest code at the cost of throughput.
2. The schedule is repeated with blocking factor $N_b = N_l(L)$, and then once more for the remainder to N_g. This improves throughput but also enlarges the code size.

When $N_l(L) = 2$ by increasing the number of delay samples to 2, a valid schedule is "5(GCDH)" if the first strategy is followed or "2(2G 2C 2D 2H) GCDH" if the second strategy is followed. Consequently, the final schedule is either "5A5B 5(GCDH) 5E 5F" or "5A 5B 2(2G 2C 2D 2H) GCDH 5E 5F".

Although the SSDF model is proposed to allow large blocking factors to utilize vector processing of simple operations in a node, the scheduling algorithm for SSDF is also applicable to an SDF graph in which every node has an inline style code specification. Without the modification of the SDF actor, the blocking factor can be applied to the SDF graph and the SDF schedule. For instance, when block factor $N_g = 3$ is applied to Fig. 2, a valid schedule is "9A 9D 6B 12C". For the given schedule with the blocking factor, programs can be synthesized as shown in Fig. 5 where each loop value in the codes will be multiplied by blocking factor N_g (=3).

References

1. Ade, M., Lauwereins, R., Peperstraete, J.A.: Implementing DSP applications on heterogeneous targets using minimal size data buffers. In: Proceedings of RSP'96, pp. 166–172 (1996)
2. Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: Software Synthesis from Dataflow Graphs. Kluwer Academic Publisher, Norwell MA (1996)
3. Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. Journal of Design Automation for Embedded Systems **2**, 33–60 (1997)
4. Bilsen, G., Engles, M., Lauwereins, R., Peperstraete, J.A.: Cyclo-static dataflow. IEEE Trans. Signal Processing **44**, 397–408 (1996)

5. Buck, J.T., Ha, S., Lee, E.A., Messerschimitt, D.G.: Ptolemy: A framework for simulating and prototyping heterogeneous systems. Int. Journal of Computer Simulation, Special issue on Simulation Software Development **4**, 155–182 (1994)
6. Dennis, J.B.: Dataflow supercomputers. IEEE Computer Magazine **13** (1980)
7. Govindarajan, R., Gao, G., Desai, P.: Minimizing memory requirements in rate-optimal schedules. In: Proceedings of the International Conference on Application Specific Array Processors, pp. 75–86 (1993)
8. Hoang, P.D., Rabaey, J.M.: Scheduling of DSP programs onto multiprocessors for maximum throughput. IEEE Transactions on Signal Processing pp. 2225–2235 (1993)
9. Jung, H., Yang, H., Ha, S.: Optimized RTL code generation from coarse-grain dataflow specification for fast HW/SW cosynthesis. Journal of Signal Processing Systems **52**, 13–34 (2008)
10. Kim, J., Shin, T., Ha, S., Oh, H.: Resource minimized static mapping and dynamic scheduling of SDF graphs. In: ESTIMedia (2011)
11. Lauwereins, R., Engels, M., Peperstraete, J.A., Steegmans, E., Ginderdeuren, J.V.: GRAPE: A CASE tool for digital signal parallel processing. IEEE ASSP Magazine **7**, 32–43 (1990)
12. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous dataflow programs for digital signal processing. IEEE Transactions on Computer **C-36**, 24–35 (1987)
13. Oh, H., Ha, S.: Memory-optimized software synthesis from dataflow program graphs with large size data samples. EURASIP Journal on Applied Signal Processing **2003**, 514–529 (2003)
14. Oh, H., Ha, S.: Fractional rate dataflow model for memory efficient synthesis. Journal of VLSI Signal Processing **37**, 41–51 (2004)
15. Parhi, K.K., Chen, Y.: Signal flow graphs and data flow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
16. Park, C., Chung, J., Ha, S.: Extended synchronous dataflow for efficient DSP system prototyping. Design Automation for Embedded Systems **3**, 295–322 (2002)
17. Pino, J., Ha, S., Lee, E.A., Buck, J.T.: Software synthesis for DSP using Ptolemy. Journal of VLSI Signal Processing **9**, 7–21 (1995)
18. Ritz, S., Pankert, M., Meyr, H.: High level software synthesis for signal processing systems. In: Proceedings of the International Conference on Application Specific Array Processors (1992)
19. Ritz, S., Willems, M., Meyr, H.: Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In: Proceedings of the ICASSP 95 (1995)
20. S. Stuijk, T.B., Geilen, M.C.W., Corporaal, H.: Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In: DAC, pp. 777–782 (2007)
21. Stuijk, S., Geilen, M.C.W., Basten, T.: Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In: DAC, pp. 899–904 (2006)
22. Sung, W., Ha, S.: Memory efficient software synthesis using mixed coding style from dataflow graph. IEEE Transactions on VLSI Systems **8**, 522–526 (2000)
23. Woods, R.: Mapping decidable signal processing graphs into FPGA implementations. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
24. Yang, H., Ha, S.: Pipelined data parallel task mapping/scheduling technique for MPSoC. In: DATE (Design Automation and Test in Europe) (2009)