Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms

Shuvra S. Bhattacharyya, Member, IEEE, Joseph T. Buck, Soonhoi Ha, Member, IEEE, and Edward A. Lee, Fellow, IEEE

Abstract—Synchronous dataflow (SDF) semantics are well-suited to representing and compiling multirate signal processing algorithms. A key to this match is the ability to cleanly express iteration without overspecifying the execution order of computations, thereby allowing efficient schedules to be constructed. Due to limited program memory, it is often desirable to translate the iteration in an SDF graph into groups of repetitive firing patterns so that loops can be constructed in the target code. This paper establishes fundamental topological relationships between iteration and looping in SDF graphs, and presents a scheduling framework that provably synthesizes the most compact looping structures for a large class of practical SDF graphs. By modularizing different components of the scheduling framework, and establishing their independence, we show how other scheduling objectives, such as minimizing data buffering requirements or increasing the number of data transfers that occur in registers, can be incorporated in a manner that does not conflict with the goal of code compactness.

I. INTRODUCTION

In the dataflow model of computation, pioneered by Dennis [6], a program is represented as a directed graph in which the nodes represent computations and the arcs specify the passage of data. Synchronous dataflow (SDF) [15] is a restricted form of dataflow in which the nodes, called actors, consume a fixed number of data items, called tokens or samples, per invocation and produce a fixed number of output samples per invocation. SDF and related models have been studied extensively in the context of synthesizing assembly code for signal processing applications, for example [8]–[11], [17], [19]–[21].

Fig. 1 shows a simple SDF graph with three actors, labeled A, B and C. Each arc is annotated with the number of samples produced by its source and the number of samples consumed by its sink. Thus, actor A produces two samples on its output arc each time it is invoked and B consumes one sample from its input arc. The “D” on the arc directed from B to C designates a unit delay, which we implement as an initial token on the arc.

In SDF, iteration is induced whenever the number of samples produced on an arc (per invocation of the source actor) does not match the number of samples consumed (per sink invocation) [13]. For example, in Fig. 1, actor B must be invoked two times for every invocation of actor A. Multirate applications often involve a large amount of iteration and thus subroutine calls must be used extensively, code must be replicated, or loops must be organized in the target program. The use of subroutine calls to implement repetition may reduce throughput significantly however, particularly for graphs involving small granularity. On the other hand, we have found that code duplication can quickly exhaust on-chip program memory [12]. Thus, it may be essential that we arrange loops in the target code. In this paper we develop topological relationships between iteration and looping in SDF graphs.

We emphasize that in this paper, we view dataflow as a programming model, not as a form of computer architecture [2]. Several programming languages used for DSP, such as Lucid [25], SISAL [16], and Silage [10] are based on, or include dataflow semantics. The developments in this paper are applicable to this class of languages. Compilers for such languages can easily construct a representation of the input program as a hierarchy of dataflow graphs. It is important for a compiler to recognize SDF components of this hierarchy, since in DSP applications, usually a large fraction of the computation can be expressed with SDF semantics. For example, in [7] Dennis shows how to convert recursive stream functions in SISAL-2 into SDF graphs.
In [12] How showed that we can often greatly improve looping by clustering subgraphs that operate at the same repetition rate, and scheduling such subgraphs as a single unit. Fig. 1 shows how this technique can improve looping. A naive scheduler might schedule this SDF graph as CABCBC, which offers no looping possibility within the schedule period. However, if we first group the subgraph \([B, C]\) into a hierarchical “supernode” \(\Omega\), a scheduler will generate the schedule \(\Lambda\Omega\). To highlight the repetition in a schedule, we let the notation \((nX_1X_2\cdots X_m)\) designate \(n\) successive repetitions of the firing sequence \(X_1X_2\cdots X_m\). We refer to a schedule expressed with this notation as a **looped schedule**. Using this notation, and substituting each occurrence of \(\Omega\) with a subschedule for the corresponding subgraph, our clustering of the uniform-rate set \([B, C]\) leads to either A(2BC) or A(2CB), both of which expose the full potential for looping in the SDF graph of Fig. 1.

We explored the looping problem further in [5]. First, we generalized How’s scheme to exploit looping opportunities that occur across sample-rate changes. Our approach involved constructing the subgraph hierarchy in a pairwise fashion by clustering exactly two nodes at each step. Our subgraph selection was based on frequency of occurrence—we selected the pair of adjacent nodes whose associated subgraph had the largest repetition count. The “repetition count” of a subgraph can be viewed as the number of times that a minimal schedule for the subgraph is repeated in a minimal schedule for the overall graph. We will define this concept precisely in the next section.

By not discriminating against sample-rate boundaries, our approach exposed looping more thoroughly than How’s scheme. Furthermore, by selecting subgraphs based on repetition count, we reduced data memory requirements, an aspect that How’s scheme did not address.

Clustering a subgraph must be done with care since certain groupings cause deadlock. Thus, for each candidate subgraph, we must first verify that its consolidation does not result in an unschedulable graph. One way to perform this check is to attempt to schedule the new SDF graph [14], but this approach is extremely time consuming if a large number of clustering candidates must be considered. In [5], we employed a computationally more efficient method in which we maintained the subgraph hierarchy on the acyclic precedence graph rather than the SDF graph. Thus we could verify whether or not a grouping introduced deadlock by checking whether or not it introduced a cycle in the precedence graph. Furthermore, we showed that this check can be performed quickly by applying a _reachability matrix_, which indicates for any two precedence graph nodes (invocations) \(P_1\) and \(P_2\), whether there is a precedence path from \(P_1\) to \(P_2\).

Two limitations surfaced in the approach of [5]. First, the storage cost of the reachability matrix proved prohibitive for multirate applications involving very large sample rate changes. Observe that this cost is quadratic in the number of distinct actor _invocations_ (precedence graph nodes). For example, a rasterization actor that decomposes an image into component pixels may involve a sample-rate change on the order of 250 000 to 1. If the rasterization output is connected to a homogenous block (for example, a gamma level correction), this block alone will produce on the order of \((250 \times 10^3)^2 = 6.25 \times 10^{16}\) entries in the reachability matrix! Thus very large rate changes preclude straightforward application of the reachability matrix, this is unfortunate because looping is most important precisely for such cases. The second limitation in [5] is its failure to process cyclic paths in the graph optimally. Since cyclic paths limit looping, first priority should be given to preserving the full amount of looping available within the strongly connected components [1] of the graph. As Fig. 2 illustrates, clustering subgraphs based on repetition count alone does not fully carry out this goal.

In this paper, we develop a class of uniprocessor scheduling algorithms that extract the most compact looping structure from the cyclic paths in the SDF graph. This scheduling _framework_ is based on a topological quality that we call “tight interdependence.” We show that for SDF graphs that contain no tightly interdependent subgraphs, our framework always synthesizes the most compact looping structures. Interestingly and fortunately, a large majority of practical SDF graphs seem to fall into this category. Furthermore, for this class of graphs, our technique does not require use of the reachability matrix, the precedence graph, or any other unreasonably large data structure. For graphs that contain tightly interdependent subgraphs, we show that our scheduling framework naturally isolates the minimal subgraphs that require special care. Only when analyzing these “tightly interdependent components,” do we need to apply reachability matrix-based analysis, or some other explicit deadlock-detection scheme.

An important aspect of our scheduling framework is its flexibility. By modularizing the framework into “sub-algorithms,” we allow other scheduling objectives to be integrated in a manner that does not conflict with code compactness objectives. Also, we show how decisions that a scheduler makes about grouping, or “clustering,” computations together can be formally evaluated in terms of their effects on program compactness. As an example, we demonstrate a very efficient clustering technique for increasing the amount of buffering that is done in machine registers, as opposed to memory, and we prove that this clustering strategy preserves code space compactness for a large class of SDF graphs.

## II. Background

An SDF program is normally translated into a loop, where each iteration of the loop executes one cycle of a periodic schedule for the graph. In this section we summarize important properties of such periodic schedules. Most of the terminology introduced in this and subsequent sections is summarized in the glossary at the end of the paper.

For an SDF graph \(G\), we denote the set of nodes in \(G\) by \(N(G)\) and the set of arcs in \(G\) by \(A(G)\). For an SDF arc \(\alpha\), we let \(source(\alpha)\) and \(sink(\alpha)\) denote the nodes at the source and the sink of \(\alpha\); we let \(p(\alpha)\) denote the number of samples produced by \(source(\alpha), c(\alpha)\) denote the number of samples consumed by \(sink(\alpha)\), and we denote the delay on \(\alpha\) by \(delay(\alpha)\). We define a _subgraph_ of \(G\) to be that SDF graph formed by any \(Z \subseteq N(G)\) together with the set of arcs
\[
\{\alpha \in A(G) | \text{source}(\alpha), \text{sink}(\alpha) \in Z\}.
\]

We denote the subgraph associated with the subset of nodes \(Z\) by \(\text{subgraph}(Z, G)\); if \(G\) is understood, we may simply write \(\text{subgraph}(Z)\). If \(N_1\) and \(N_2\) are two nodes in an SDF graph, we say that \(N_1\) is a successor of \(N_2\) if there is an arc directed from \(N_2\) to \(N_1\); we say that \(N_1\) is a predecessor of \(N_2\) if \(N_2\) is a successor of \(N_1\); and we say that \(N_1\) and \(N_2\) are adjacent if \(N_1\) is a predecessor or successor of \(N_2\). A sequence of nodes \((N_1, N_2, \ldots, N_k)\) is a path from \(N_1\) to \(N_k\) if \(N_{i+1}\) is a successor of \(N_i\) for \(i = 1, 2, \ldots, (k - 1)\). A sequence of nodes \((N_1, N_2, \ldots, N_k)\) is a chain that joins \(N_1\) and \(N_k\) if \(N_{i+1}\) is adjacent to \(N_i\) for \(i = 1, 2, \ldots, k\).

We can think of each arc in \(G\) as having a FIFO queue that buffers the tokens that pass through the arc. Each FIFO contains an initial number of samples equal to the delay on the associated arc. Firing a node in \(G\) corresponds to removing \(c(\alpha)\) tokens from the head of the FIFO for each input arc \(\alpha\), and appending \(p(\beta)\) tokens to the FIFO for each output arc \(\beta\). After a sequence of 0 or more firings, we say that a node is fireable if there are enough tokens on each input FIFO to fire the node. An admissible sequential schedule (“sequential” is used to distinguish this type of schedule from a parallel schedule) for \(G\) is a finite sequence \(S = S_1S_2 \cdots S_k\) of nodes in \(G\) such that each \(S_i\) is fireable immediately after \(S_{i-1}\) has fired in succession.

We say that a sequential schedule \(S\) is a periodic schedule if it invokes each node at least once and produces no net change in the number of tokens on any arc’s FIFO—for each arc \(\alpha\), (the number of times \text{source}(\alpha) is fired in \(S\)) \(\times p(\alpha) = \) (the number of times \text{sink}(\alpha) is fired in \(S\)) \(\times c(\alpha)\). A periodic admissible sequential schedule (PASS) is a schedule that is both periodic and admissible. We will use the term valid schedule to describe a schedule that is a PASS, and the term consistent to describe an SDF graph that has a PASS. Except where otherwise stated, we deal only with consistent SDF graphs in this paper.

In [14], it is shown that for each connected SDF graph \(G\), there is a unique minimum number of times that each node needs to be invoked in a periodic schedule. We specify these minimum numbers of firings by a vector of positive integers \(q_G\), which is indexed by the nodes in \(G\), and we denote the component of \(q_G\) corresponding to a node \(N\) by \(q_G(N)\). Every PASS for \(G\) invokes each node \(N\) a multiple of \(q_G(N)\) times, and corresponding to each PASS \(S\), there is a positive integer \(J(S)\) called the blocking factor of \(S\), such that \(S\) invokes each \(N \in N(G)\) exactly \(J(S)q_G(N)\) times. We call \(q_G\) the repetitions vector of \(G\). For example in Fig. 2(a), \(q_G(A) = 10\), \(q_G(B) = 4\), and \(q_G(C) = 5\). The following properties of repetitions vectors are established in [14]:

**Fact 1**: The components of a repetitions vector are collectively coprime.

**Fact 2**: The balance equation \(q_G(\text{source}(\alpha)) \times p(\alpha) = q_G(\text{sink}(\alpha)) \times c(\alpha)\) is satisfied for each arc \(\alpha\) in \(G\).

Given a subset \(Z\) of nodes in a connected SDF graph \(G\), we define \(q_G(Z) = \gcd\{q_G(N)|N \in Z\}\), where \(\gcd\) denotes the greatest common divisor. We can interpret \(q_G(Z)\) as the number of times that \(G\) invokes the “subsystem” \(Z\). We will use the following property of connected subsystems which is derived in [4].

**Fact 3**: If \(G\) is a connected SDF graph, and \(Z\) is a connected subset of \(N(G)\), then for each \(N \in Z\), \(q_G(N) = q_G(Z)(q_G(\text{subgraph}(Z); N))\).

For our hierarchical scheduling approach, we will apply the concept of clustering a subgraph. This process is illustrated in Fig. 2. Here \(\text{subgraph}(A, C)\) of (a) is clustered into the hierarchical node \(\Omega_{AC}\), and the resulting SDF graph is shown in (b). Similarly, clustering \(\text{subgraph}(A, B)\) results in the graph of (c). Each input arc \(\alpha\) to a clustered subgraph \(P\) is replaced by an arc \(\alpha’\) having \(p(\alpha’) = p(\alpha)\), and \(c(\alpha’) = c(\alpha) \times q_G(\text{sink}(\alpha))/q_G(N(P))\), the number of samples consumed from \(\alpha\) in one invocation of subgraph \(P\). Similarly we replace each output arc \(\beta’\) with \(\beta\) such that \(c(\beta’) = c(\beta)\), and

\[
p(\beta) = p(\beta) \times q_G(\text{source}(\alpha)) / q_G(N(P)).
\]

The following properties of clustered subgraphs are proven in [4].

**Fact 4**: Suppose \(G\) is a connected SDF graph, \(Z\) is a subset of nodes in \(G\), \(G’\) is the SDF graph that results from clustering \(\text{subgraph}(Z)\) into the hierarchical node \(\Omega\), and \(S’\) is a PASS for \(G’\). Suppose that \(S_Z\) is a PASS for \(\text{subgraph}(Z)\) such that for each \(N \in Z, S_Z\) invokes \(N\) \((q_G(N)/q_G(Z))\) times. Let \(S^*\) denote the schedule that results from replacing each appearance of \(\Omega\) in \(S\) with \(S_Z\). Then \(S^*\) is a PASS for \(G\).

**Fact 5**: Suppose \(G\) is a connected SDF graph, \(Z\) is a subset of nodes in \(G\), and \(G’\) is the SDF graph that results from clustering \(\text{subgraph}(Z)\) into the node \(\Omega\). Then \(q_G(\Omega) = q_G(Z)\); and for any node \(N\) in \(G’\) other that \(\Omega\), \(q_G(N) = q_G(N)\).

Given a directed graph \(G\), we say that \(G\) is strongly connected if for any pair of distinct nodes \(A, B\) in \(G\), there is a path from \(A\) to \(B\) and a path from \(B\) to \(A\). We say that a strongly connected graph is nontrivial if it contains more than one node. Finally, a strongly connected component of \(G\) is a subset of nodes \(Z\) such that \(\text{subgraph}(Z, G)\) is strongly connected, and there is no strongly connected subset of \(N(G)\) that properly contains \(Z\). For example \(\{A, B\}\) and \(\{C\}\) are the strongly connected components of Fig. 2(a).

Similarly, we define a connected component of a directed graph \(G\) to be a maximal subset of nodes \(Z\) such that for any
pair of distinct members A, B of Z, there is a chain that joins A and B. For example in Fig. 3, the connected components are \{A\}, \{C, D, F\}, and \{B, E\}.

Given a connected SDF graph G, and an arc \( \alpha \) in G, we define \( \text{total\_consumed}(\alpha, G) \) to be the total number of samples consumed from \( \alpha \) in a minimal schedule period for G. Thus \( \text{total\_consumed}(\alpha, G) = q_G(\text{sink}(\alpha))c(\alpha) \). Finally, given an SDF graph G, a looped schedule S for G and a node N in G, we define \( \text{appearances}(N, S) \) to be the number of times that N appears in S, and we say that S is a single appearance schedule if for each \( N \in N(G) \), \( \text{appearances}(N, S) = 1 \). For example, consider the two schedules \( S_1 = CA(2B)C \) and \( S_2 = A(2B)(2C) \) for Fig. 1. We have \( \text{appearances}(C, S_1) = 2 \); \( \text{appearances}(C, S_2) = 1 \); \( S_1 \) is not a single appearance schedule because C appears more than once; and \( S_2 \) is a single appearance schedule. Single appearance schedules form the class of schedules that allow in line code generation without any code space or subroutine penalty.

III. SUBINDEPENDENCE

Our scheduling framework for synthesizing compact nested loop structures is based on a form of precedence independence, which we call subindependence.

Definition 1: Suppose that G is a connected SDF graph. If \( Z_1 \) and \( Z_2 \) are disjoint, nonempty subsets of \( N(G) \) we say that “\( Z_1 \) is subindependent of \( Z_2 \) in G” if for every arc \( \alpha \) in G such that source(\( \alpha \)) \( \in Z_2 \) and sink(\( \alpha \)) \( \in Z_1 \), we have delay(\( \alpha \)) \( \geq \text{total\_consumed}(\alpha, G) \). We occasionally drop the “in G” qualification if G is understood from context. If \( \{ Z_1 \} \) is subindependent of \( \{ Z_2 \} \) and \( \{ Z_1 \cup Z_2 = N(G) \} \), then we write \( \{ Z_1 \} \text{G} \{ Z_2 \} \), and we say that \( \{ Z_1 \} \) is subindependent in G.

Thus \( Z_1 \) is subindependent of \( Z_2 \) if no samples produced from \( Z_2 \) are consumed by \( Z_1 \) in the same schedule period that they are produced; and \( Z_1 \cup Z_2 \) is subindependent of \( Z_2 \), and \( Z_1 \) and \( Z_2 \) form a partition of the nodes in G. For example, consider Fig. 2(a). Here \( q_G(A, B, C) = \{10, 4, 5\} \), and the complete set of subindependence relationships is (1) \{A\} is subindependent of \{C\}; (2) \{B\} is subindependent of \{C\}; (3) \{A, B\} \{G\} \{C\}; and \{C\} is subindependent of \{B\}.

The following property of subindependence follows immediately from definition 1.

Fact 6: If G is a strongly connected SDF graph and X, Y, and Z are disjoint subsets of N(G), then (a) (X is subindependent of Z) \( \Rightarrow \) (Y \( \cup \) Y is subindependent of \( \{ X \cup Y \} \) is subindependent of \( \{ Z \} \) \( \Rightarrow \) X is subindependent of \( \{ Y \cup Z \} \).

Our scheduling framework is based on the following condition for the existence of a single appearance schedule, which is developed in [4].

Fact 7: An SDF graph has a valid single appearance schedule iff for each nontrivial strongly connected component Z,
a single appearance schedule is equivalent to the absence of tightly interdependent subgraphs. However, these properties are useful even when tightly interdependent subgraphs are present. The following definition specifies how to use loose interdependence to guide the looping process.

Definition 3: Let \( A_1 \) be any algorithm that takes as input a nontrivial strongly connected SDF graph \( G \), determines whether \( G \) is loosely interdependent, and if so, finds a subindependent subset of \( N(G) \). Let \( A_2 \) be any algorithm that finds the strongly connected components of a directed graph. Let \( A_3 \) be any algorithm that takes an acyclic SDF graph and generates a valid single appearance schedule. Finally, let \( A_4 \) be any algorithm that takes a tightly interdependent SDF graph, and generates a valid looped schedule of blocking factor 1. We define the algorithm \( L(A_1, A_2, A_3, A_4) \) as follows:

**Input:** a connected SDF graph \( G \).

**Output:** a valid unit-blocking-factor looped schedule \( S_L(G) \) for \( G \).

**Step 1:** Use \( A_2 \) to determine the nontrivial strongly connected components \( Z_1, Z_2, \ldots, Z_t \) of \( G \).

**Step 2:** Cluster \( Z_1, Z_2, \ldots, Z_t \) into nodes \( \Omega_1, \Omega_2, \ldots, \Omega_n \), respectively, and call the resulting graph \( G' \). This is an acyclic SDF graph.

**Step 3:** Apply \( A_3 \) to \( G' \); denote the resulting schedule \( S'(G') \).

**Step 4:** (Shown at the bottom of the page.)

The for-loop replaces each "\( \Omega_i \)" in \( S'(G') \) with a valid looped schedule for \( subgraph(Z_i) \). From repeated application of fact 4, we know that these replacements yield a valid looped schedule \( S_L(G) \) for \( G \). We output \( S_L(G) \).

Remark 1: Observe that step 4 does not insert or delete appearances of actors that are not contained in a nontrivial strongly connected component \( Z_i \). Since \( A_3 \) generates a single appearance schedule for \( G' \), we have that for every node \( N \) that is contained in a nontrivial strongly connected component of \( G \), \( appearances(N, S_L(G)) = 1 \).

Remark 2: If \( C \) is a nontrivial strongly connected component of \( G \) and \( N \in C \), then since \( S_L(G) \) is derived from \( S'(G) \) by replacing the single appearance of each \( \Omega_i \), we have \( appearances(N, S_G(G)) = appearances(N, S_L(subgraph(G))) \).

Remark 3: For each strongly connected component \( Z_k \) whose subgraph is loosely interdependent, \( L \) partitions \( Z_k \) into \( X \) and \( Y \) such that \( X \cup Y = Z_k \) and \( X \cup Y = Z_k \). If \( X \) is a member of the connected component \( X_i \), then \( N \neq Y \), so \( appearances(N, S_x, S_y) = appearances(N, S_L(subgraph(X_i))) \). Also since \( N \) cannot be in any other strongly connected component besides \( Z_k \), and since \( S'(G) \) contains only one appearance of \( \Omega_k \), we have \( appearances(N, S_L(G)) = appearances(N, S_L(subgraph(X_i))) \).

The same argument can be used to show that for \( i = 1, 2, \ldots, n \), \( N \in X_i \Rightarrow appearances(N, S_L(G)) = appearances(N, S_L(subgraph(Y_i))) \).

\( L(\bullet, \bullet, \bullet, \bullet) \) defines a family of algorithms, which we call loose interdependence algorithms because they exploit loose interdependence to decompose the input SDF graph. Since nested recursive calls decompose a graph into finer and finer strongly connected components, it is easy to verify that any loose interdependence algorithm always terminates. Each loose interdependence algorithm \( \lambda = L(A_1, A_2, A_3, A_4) \) involves the "sub-algorithms" \( A_1, A_2, A_3 \), and \( A_4 \), which we call, respectively, the subindependence partitioning algorithm of \( \lambda \), the strongly connected components algorithm of \( \lambda \), the acyclic scheduling algorithm of \( \lambda \), and the tight scheduling algorithm of \( \lambda \).

We will apply a loose interdependence algorithm to derive a nonrecursive necessary and sufficient condition for the existence of a single appearance schedule. First, we introduce two lemmas.

**Lemma 1:** Suppose \( G \) is a connected SDF graph; \( N \) is a node in \( G \) that is not contained in any tightly interdependent subgraph of \( G \); and \( \lambda \) is a loose interdependence algorithm. Then \( N \) appears only once in \( S_\lambda(G) \), the schedule generated by \( \lambda \).

---

Step 4:

\[
\text{for } i = 1, 2, \ldots, s \\
\text{let } SZ \text{ denote subgraph}(Z_i). \\
\text{apply } A_1 \text{ to } SZ. \\
\text{if } X, Y \subseteq Z_i \text{ are found such that } X \mid SZ \text{ Y, then} \\
\text{\hspace{1cm} \bullet \text{ determine the connected components } X_1, X_2, \ldots, X_s \text{ of subgraph}(X), \text{ and the connected components } Y_1, Y_2, \ldots, Y_w \text{ of subgraph}(Y).} \\
\text{\hspace{1cm} \bullet \text{ recursively apply algorithm } L \text{ to construct the schedules } } \\
\text{\hspace{1.5cm} } S_x = \text{eqsz}(X_1)S_L(subgraph(X_1)) \cdots \text{eqsz}(X_s)S_L(subgraph(X_s)), \\
\text{\hspace{1.5cm} } S_y = \text{eqsz}(Y_1)S_L(subgraph(Y_1)) \cdots \text{eqsz}(Y_w)S_L(subgraph(Y_w)). \\
\text{\hspace{1cm} \bullet \text{ replace the (single) appearance of } \Omega_i \text{ in } S'(G) \text{ with } S_xS_y,} \\
\text{else (} SZ \text{ is tightly interdependent) } \\
\text{\hspace{1cm} \bullet \text{ apply } A_4 \text{ to obtain a valid schedule } S_i \text{ for } SZ.} \\
\text{\hspace{1cm} \bullet \text{ replace the single appearance of } \Omega_i \text{ in } S'(G) \text{ with } S_i.} \\
\text{end – if} \\
\text{end – for}
The proof of lemma 1 can be found in the appendix.

Lemma 2: Suppose that G is a strongly connected SDF graph, \( P \subseteq N(G) \) is subindependent in G, and C is a strongly connected subset of \( N(G) \) such that \( C \cap P \neq C \) and \( C \cap P \neq \emptyset \). Then \( C \cap P \) is subindependent in \( subgraph(C) \).

Proof: Suppose that \( \alpha \) is an arc directed from a member of \( (C - (C \cap P)) \) to a member of \( (C \cap P) \). By the subindependence of P in G, \( delay(\alpha) \geq c(\alpha) \times q_{inds}(sink(\alpha)) \), and by fact 3, \( q_{inds}(sink(\alpha)) \geq q_{subgraph(C)}(sink(\alpha)) \). Thus, \( delay(\alpha) \geq c(\alpha) \times q_{subgraph(C)}(sink(\alpha)) \). Since this holds for any \( \alpha \) directed from \( (C - (C \cap P)) \) to \( (C \cap P) \), we conclude that \( (C \cap P) \) is subindependent in C.

QED

Corollary 1: Suppose that G is a strongly connected SDF graph, \( Z_1 \) and \( Z_2 \) are subsets of \( N(G) \) such that \( Z_1 \cap GZ_2 \) and \( T \) is a tightly interdependent subgraph of G. Then \( N(T) \subseteq Z_1 \) or \( N(T) \subseteq Z_2 \).

Proof: (By contraposition.) If \( N(T) \) has nonempty intersection with both \( Z_1 \) and \( Z_2 \), then from lemma 2, \( N(T) \cap Z_1 \) is subindependent in T, so T is loosely interdependent. QED

Theorem 1: Suppose that G is a strongly connected SDF graph. Then G has a single appearance schedule iff every nontrivial strongly connected subgraph of G is loosely interdependent.

Proof: Suppose every nontrivial strongly connected subgraph of G is loosely interdependent, and let \( \lambda \) be any loose interdependence algorithm. Since no node in G is contained in a tightly interdependent subgraph, lemma 1 guarantees that \( S_\lambda(G) \) is a single appearance schedule for G.

⇒ Suppose that G has a single appearance schedule and that C is a strongly connected subset of \( N(G) \). Set \( Z_0 = G \). From fact 7, there exist \( X_0, Y_0 \subseteq Z_0 \) such that \( X_0, subgraph(X_0) \), and \( subgraph(Y_0) \) both have single appearance schedules. If \( X_0 \) and \( Y_0 \) do not both intersect C then C is completely contained in some strongly connected component \( Z^*_1 \) of \( subgraph(X_0) \) or \( subgraph(Y_0) \). We can then apply fact 7 to partition \( Z^*_1 \) into \( X_1, Y_1 \), and continue recursively in this manner until we obtain a strongly connected \( Z_k \subseteq N(G) \), with the following properties: \( Z_k \) can be partitioned into \( X_k \) and \( Y_k \) such that \( X_k, subgraph(Z_k), Y_k, Z_k \); \( Y_k \cap C \) is nonempty. From \( X_k \subseteq X_1 \), \( X_1 \subseteq subgraph(C) \), so C must be loosely interdependent.

QED

Corollary 2: Given a connected SDF graph G, any loose interdependence algorithm will obtain a single appearance schedule if one exists.

Proof: If a single appearance schedule for G exists, then from theorem 1, G contains no tightly interdependent subgraphs. In other words, no node in G is contained in a tightly interdependent subgraph of G. From lemma 1, the schedule resulting from any loose interdependence algorithm contains only one appearance for each actor in G.

QED

Thus, a loose interdependence algorithm always obtains an optimally compact solution when a single appearance schedule exists. When a single appearance schedule does not exist, strongly connected graphs are repeatedly decomposed until tightly interdependent subgraphs are found. In general, however, there may be more than one way to decompose \( N(G) \) into two parts so that one of the parts is subindependent of the other. Thus, it is natural to ask the following question: Given two distinct partitions \( \{Z_1, Z_2\} \) and \( \{Z'_1, Z'_2\} \) such that \( Z_1 \cap GZ_2 \) and \( Z'_1 \cap GZ'_2 \), is it possible that one of these partitions leads to a more compact schedule than the other? Fortunately, as we will show in the remainder of this section, the answer to this question is “No.” In other words, any two loose interdependence algorithms that use the same tight scheduling algorithm always lead to equally compact schedules. The key reason is that tight interdependence is an additive property.

Lemma 3: Suppose that G is a connected SDF graph, \( Y \) and \( Z \) are subsets of \( N(G) \) such that \( Y \cap Z \neq \emptyset \), and \( subgraph(Y) \) and \( subgraph(Z) \) are both tightly interdependent. Then \( subgraph(Y \cup Z) \) is tightly interdependent.

Proof: (By contraposition.) Let \( H = Y \cup Z \), and suppose that \( subgraph(H) \) is loosely interdependent. Then there exist \( H_1 \) and \( H_2 \) such that \( H = H_1 \cup H_2 \) and \( subgraph(H_1) \cup subgraph(H_2) \). From \( H_1 \cup H_2 = Y \cup Z \), and \( Y \cap Z \neq \emptyset \), it is easily seen that \( H_1 \) and \( H_2 \) both have a nonempty intersection with \( Y \), or they both have a nonempty intersection with \( Z \). Without loss of generality, assume that \( H_1 \cap Y \neq \emptyset \) and \( H_2 \cap Y \neq \emptyset \). From lemma 2, \( (H_1 \cap Y) \) is subindependent in \( subgraph(Y) \), and thus \( subgraph(Y) \) is not tightly interdependent.

QED

Lemma 3 implies that each SDF graph G has a unique set \( \{C_1, C_2, \ldots, C_n\} \) of maximal tightly interdependent subgraphs such that \( i \neq j \Rightarrow N(C_i) \cap N(C_j) = \emptyset \), and every tightly interdependent subgraph in G is contained in some \( C_i \). We call each \( N(C_i) \) a tightly interdependent component of G. It follows from theorem 1 that G has a single appearance schedule if G has no tightly interdependent components. Furthermore, since the tightly interdependent components are unique, the performance of a loose interdependence algorithm, with regards to schedule compactness, is not dependent on the particular subindependence partitioning algorithm, the sub-algorithm used to partition the loosely interdependent components. The following theorem develops this result.

Theorem 2: Suppose G is an SDF graph that has a PASS, \( N \) is a node in G, and \( \lambda \) is a loose interdependence algorithm. If \( N \) is not contained in a tightly interdependent component of G, then \( N \) appears only once in \( S_\lambda(G) \).

On the other hand, if \( N \) is contained in a tightly interdependent component \( T \) then \( appearances(N, S_\lambda(G)) = appearances(N, S_\lambda(subgraph(T))) \)—the number of appearances of \( N \) is determined entirely by the tight scheduling algorithm of \( \lambda \).

Proof: If \( N \) is not contained in a tightly interdependent component of G, then \( N \) is not contained in any tightly interdependent subgraph. Then from lemma 1, \( appearances(N, S_\lambda(G)) = 1 \).

Now suppose that \( N \) is contained in some tightly interdependent component \( T \) of G. If \( T = N(G) \) we are done. Otherwise we set \( M_0 = N(G) \), and thus \( T \neq M_0 \) by definition, tightly interdependent graphs are strongly connected, so \( T \) is contained in some strongly connected component \( C \) of \( subgraph(M_0) \).

If \( T \) is a proper subset of \( C \), then \( subgraph(C) \) must be loosely interdependent, since otherwise \( subgraph(T) \) would not be a maximal tightly interdependent subgraph.
Thus, $\lambda$ partitions $\text{subgraph}(C)$ into $X$ and $Y$ such that $X$ (\text{subgraph}(C) Y). We set $M_1$ to be that connected component of $\text{subgraph}(X)$ or $\text{subgraph}(Y)$ that contains $N$. Since $X$, $Y$ partition $C$, $M_1$ is a proper subset of $M_0$. Also, from remark 3, \text{appearances}(N, S_{\lambda}(\text{subgraph}(M_0))) = \text{appearances}(N, S_{\lambda}(\text{subgraph}(M_1)))$, and from corollary 1, $N(T) \subseteq M_1$.

On the other hand, if $T = C$, then we set $M_1 = T$. Since $T \neq M_0$, $M_1$ is a proper subset of $M_0$; from remark 2, \text{appearances}(N, S_{\lambda}(\text{subgraph}(M_0))) = \text{appearances}(N, S_{\lambda}(\text{subgraph}(M_1))) and trivially, $T \subseteq M_1$.

If $T \neq M_1$, then we can repeat the above procedure to obtain a proper subset $M_2$ of $M_1$ such that \text{appearances}(N, S_{\lambda}(\text{subgraph}(M_1))) = \text{appearances}(N, S_{\lambda}(\text{subgraph}(M_2)))$, and $N(T) \subseteq M_2$. Continuing this process, we get a sequence $M_1, M_2, \ldots$. Since each $M_k$ is a proper subset of its predecessor, we cannot repeat this process indefinitely—eventually, for some $k \geq 0$, we will have $N(T) = M_k$. But, by construction, \text{appearances}(N, S_{\lambda}(G)) = \text{appearances}(N, S_{\lambda}(\text{subgraph}(M_0))) = \text{appearances}(N, S_{\lambda}(\text{subgraph}(M_1))) = \cdots = \text{appearances}(N, S_{\lambda}(\text{subgraph}(T)))$. QED.

Theorem 2 states that the tight scheduling algorithm is independent of the subdependence partitioning algorithm, and vice-versa. Any subdependence partitioning algorithm makes sure that there is only one appearance for each actor outside the tightly interdependent components, and the tight scheduling algorithm completely determines the number of appearances for actors inside the tightly interdependent components. For example, if we develop a new subdependence partitioning algorithm that is more efficient in some way (e.g., it is faster or minimizes data memory requirements), we can replace it for any existing subdependence partitioning algorithm without changing the “compactness” of the resulting schedules—we don’t need to analyze its interaction with the rest of the loose interdependence algorithm. Similarly, if we develop a new tight scheduling algorithm that schedules any tightly interdependent graph more compactly than the existing tight scheduling algorithm, we are guaranteed that using the new algorithm instead of the old one will lead to more compact schedules overall.

V. COMPUTATIONAL EFFICIENCY

The complexity of a loose interdependence algorithm $\lambda$ depends on its subdependence partitioning algorithm $\lambda_{sp}$, strongly connected components algorithm $\lambda_{sc}$, acyclic scheduling algorithm $\lambda_{as}$, and tight scheduling algorithm $\lambda_{ts}$. From the proof of theorem 2, we see that $\lambda_{ts}$ is applied exactly once for each tightly interdependent component. For example, the simplest solution for a tight scheduling algorithm would be to apply an algorithm from the family of class-S scheduling algorithms that are defined in [14]; class-S algorithms exist whose complexity is linear in the number of actor firings (assuming that the number of input and output edges for a given actor is bounded) [3]. Alternatively, a more elaborate technique such as that presented in [5] can be employed.

As mentioned earlier, one drawback of the technique of [5] is that it requires a reachability matrix, which has a storage cost that is quadratic in the number of actor firings. However, we greatly reduce this drawback by restricting application of the algorithm to only the tightly interdependent components. We are currently investigating other alternatives to scheduling tightly interdependent SDF graphs.

The other subalgorithms, $\lambda_{sc}$, $\lambda_{as}$, and $\lambda_{ts}$, are successively applied to decompose an SDF graph, and the process is repeated until all tightly interdependent components are found. In the worst case, each decomposition step isolates a single node from the current $n$-node subgraph, and the decomposition must be recursively applied to the remaining $(n-1)$-node subgraph. Thus, if the original program has $n$ nodes, $n$ decomposition steps are required in the worst case. Tarjan [24] first showed that the strongly connected components of a graph can be found in $O(m)$ time, where $m = max(number$ of nodes, number of arcs). Hence $\lambda_{sc}$ can be chosen to be linear, and since at most $n \leq m$ decomposition steps are required, the total time that such a $\lambda_{sc}$ accounts for in $\lambda$ is $O(n^2)$. In Section III we presented a simple linear-time algorithm that constructs a single appearance schedule for an acyclic SDF graph. Thus $\lambda_{as}$ can be chosen such that its total time is also $O(n^2)$.

The following theorem presents a simple topological condition for loose interdependence that leads to a linear subdependence partitioning algorithm $\lambda_{sp}$.

Theorem 3: Suppose that $G$ is a nontrivial strongly connected SDF graph. From $G$, remove all arcs $\alpha$ for which $\text{delay}(\alpha) \geq c(\alpha) \times q_{G}(\text{sink}(\alpha))$, and call the resulting SDF graph $G'$. Then $G$ is tightly interdependent if and only if $G'$ is strongly connected.

For example, suppose that $G$ is the strongly connected SDF graph in Fig. 5(a). The repetitions vector for $G$ is $q_{G}(A, B, C, D) = (1, 2, 2, 4)$. This graph is loosely interdependent if $d_1 \geq 2$, which corresponds to $\{C, D\} \nsubseteq \{A, B\}$, or if $d_2 \geq 4$, which corresponds to $\{A, B\} \nsubseteq \{C, D\}$. The corresponding $G'$'s are depicted at the bottom of Fig. 5: Fig. 5(b) shows $G'$ when $d_1 \geq 2$ and $d_2 < 4$, and Fig. 5(c) shows $G'$ when $d_2 \geq 4$ and $d_1 < 2$. Observe that in both of these cases, $G'$ is not strongly connected.
Proof: We prove both directions by contraposition.

⇒ Suppose that \( G' \) is not strongly connected. Then \( N(G) \) can be partitioned into \( Z_1 \) and \( Z_2 \) such that there is no arc directed from a member of \( Z_2 \) to a member of \( Z_1 \) in \( G' \). Since no nodes were removed in constructing \( G' \), \( Z_1 \) and \( Z_2 \), in \( G \) occur in \( G' \). Thus, by the construction of \( G' \), for each arc \( \alpha \) in \( G \) directed from a member of \( Z_2 \) to a member of \( Z_1 \), we have \( \text{delay}(\alpha) \geq c(\alpha) \times q_G(\text{sink}(\alpha)) \). It follows that \( Z_1(GZ_2) \), so \( G \) is loosely interdependent.

⇐ Suppose that \( G \) is loosely interdependent. Then \( N(G) \) can be partitioned into \( Z_1 \) and \( Z_2 \) such that \( Z_1(GZ_2) \). By construction of \( G' \), there are no arcs in \( G' \) directed from a member of \( Z_2 \) to a member of \( Z_1 \), so \( G' \) is not strongly connected.

Thus, \( \lambda_{\text{sp}} \) can be constructed as follows: (1) Determine \( q_G(N) \) for each node \( N \); (2) Remove each arc \( \alpha \) whose delay is at least \( c(\alpha) \times q_G(\text{sink}(\alpha)) \); (3) Determine the strongly connected components of the resulting graph; (4) If the entire graph is the only strongly connected component, then \( G \) is tightly interdependent; otherwise (5) cluster the strongly connected components—the resulting graph is acyclic and has at least two nodes. The strongly connected component corresponding to any root node of this graph is subcomponent of the rest of the graph. An algorithm (first used in the Gabriel system [11]) that performs (1) in time \( O(m) \) is described in [3]; it is obvious that (2) is \( O(m) \); Tarjan’s algorithm allows \( O(m) \) for (3); and the checks in (4) and (5) are clearly \( O(m) \) as well. Thus, we have a linear \( \lambda_{\text{sp}} \) and the total time that \( \lambda \) spends in \( \lambda_{\text{sp}} \) is \( O(m^2) \).

We have specified \( \lambda_{\text{sp}}, \lambda_{\text{sc}}, \lambda_{\text{sa}}, \) and \( \lambda_{\text{as}} \) such that the time complexity of the corresponding loose interdependence algorithm is \( O(m^2 + f) \), where \( m = \text{max}(\text{number of nodes, number of arcs}) \), and \( f \) is the number of actor firings. Note that our worst case estimate is conservative—in practice only a few decomposition steps are required to fully schedule a strongly connected subgraph, while our estimate assumes \( n \) steps, where \( n \) is the number of nodes in the input graph.

VI. CLUSTERING TO MAKE DATA TRANSFERS MORE EFFICIENT

In this section, we present a useful clustering technique for increasing the frequency of data transfers that occur through machine registers rather than memory, and we prove that this technique does not interfere with the code compactness potential of a loose interdependence algorithm—this clustering preserves the properties of loose interdependence algorithms discussed in Section IV.

Fig. 6 illustrates two ways in which arbitrary clustering decisions can conflict with code compactness objectives. Observe that Fig. 6(a) is an acyclic graph so it must have a single appearance schedule. Fig. 6(b) is the hierarchical SDF graph that results from clustering A and B in Fig. 6(a). It is easy to verify that this is a tightly interdependent graph. In fact, the only minimal periodic schedule for Fig. 6(a) that we can derive from this clustering is \( C \cup C \Rightarrow CABC \). Thus, the clustering of A and B in Fig. 6(a) cancels the existence of a single appearance schedule.

In Fig. 6(c), \{A, B\} forms a tightly interdependent component and C is not contained in any tightly interdependent subgraph. From theorem 2, we know that any loose interdependence algorithm will schedule Fig. 6(c) in such a way that C appears only once. Now observe that the graph that results from clustering A and C, shown in Fig. 6(d), is tightly interdependent. It can be verified that the most compact minimal periodic schedule for this graph is \( (5\Omega_2) \Omega_1(5\Omega_1) \), which leads to the schedule \( (5\text{AC})\Omega_2(5\text{AC}) \) for Fig. 6(c). By increasing the “extent” of the tightly interdependent component \{A, B\} to subsume C, this clustering decision increases the minimum number of appearances of C in the final schedule.

Thus we see that a clustering decision can conflict with optimal code compactness if it introduces a new tightly interdependent component or extends an existing tightly interdependent component. In this section we present a clustering technique of great practical use and prove that it neither extends nor introduces tight interdependence. Our clustering technique and its compatibility with loose interdependence algorithms is summarized by the following claim: Clustering two adjacent nodes A and B in an SDF graph does not introduce or extend a tightly interdependent component if (a) Neither A nor B is contained in a tightly interdependent component; (b) At least one arc directed from A to B has zero delay; (c) A and B are invoked the same number of times in a periodic schedule; and (d) B has no predecessors other than A or B. The remainder of this section is devoted to proving this claim and explaining the corresponding clustering technique.

We motivate our clustering technique with the example shown in Fig. 7. One possible single appearance schedule for Fig. 7(a) is \( (10\text{X})(10\text{Y})\text{ZV}(10\text{W}) \). This is the minimum activation schedule preferred by Ritz et al. [22]; however, it is inefficient with respect to buffering. Due to the loop that specifies ten successive invocations of X, the data transfers between X and Y cannot take place in machine registers and 10 words of data-memory are required to implement the arc connecting X and Y. However, observe that conditions (a)–(d) of our above claim all hold for the adjacent pairs \{X, Y\} and \{Z, V\}. Thus, we can cluster these pairs without cancelling the existence of a single appearance schedule. The hierarchical graph that results from this clustering is shown in Fig. 7(d); this graph leads to the single appearance schedule \( (10\Omega_2)\Omega_1(10\text{W}) \Rightarrow (10\text{XY})\text{ZV}(10\text{W}) \). In this second schedule, each sample produced by X is consumed by Y in the same loop iteration, so all of the transfers between X and Y can occur through a single machine register. Thus, the clustering of X and Y saves 10 words of buffer space for the
data transfers between X and Y, and it allows these transfers to be performed through registers rather than memory, which will usually result in faster code.

We will use the following additional notation in the development of this section.

Notation: Let G be an SDF graph and suppose that we cluster a subset W of nodes in G. We will refer to the resulting hierarchical graph as G’, and we will refer to the node in G’ into which W has been clustered as Ω. For each arc α in G that is not contained in subgraph(W), we denote the corresponding arc in G’ by α’. Finally, if X ⊆ N(G), we refer to the “corresponding” subset of N(G’) as X’. That is, X’ consists of all members of X that are not in W; and if X contains a member of W, then X’ also contains Ω.

For example, if G is the SDF graph in Fig. 6(a), W = {A, B}, and α and β respectively denote the arc directed from A to C and the arc directed from C to B, then we denote the graph in Fig. 6(b) by G’, and in G’ we denote the arc directed from Ω to C by α’ and the arc denoted from C to Ω by β’. Also, if X = {A, C}, then X’ = {Ω, C}.

Lemma 4: Suppose that G is a strongly connected SDF graph and X1, X2 partition N(G) such that X1 ≥ X2. Also suppose that A, B are nodes in G such that A, B ∈ X1 or A, B ∈ X2. If we cluster W = {A, B} then the resulting SDF graph G’ is loosely interdependent.

The proof of lemma 4 can be found in the appendix.

Definition 4: We say that two SDF graphs G1 and G2 are isomorphic if there exist bijection mappings f1: N(G1) → N(G2) and f2: A(G1) → A(G2) such that for each α ∈ A(G1), source(f2(α)) = f1(source(α)), sink(f2(α)) = f1(sink(α)), delay(f2(α)) = delay(α), p(2(α)) = p(α), and c(f2(α)) = c(α). Intuitively, two SDF graphs are isomorphic if they differ only by a relabeling of the nodes. For example, the SDF graph in Fig. 6(d) is isomorphic to subgraph([{A, B}], G) in Fig. 6(c).

We will use the following obvious fact about isomorphic SDF graphs.

Fact 8: If G1 and G2 are two isomorphic SDF graphs and G1 is loosely interdependent then G2 is loosely interdependent.

Lemma 5: Suppose that G is an SDF graph, M ⊆ N(G), A1 ∈ M, and A2 is an SDF node that is contained in N(G) but not in M such that
1) A2 is not adjacent to any member of (M - {A1}), and
2) for some positive integer k, q(A2) = kq(A1).

Then if we cluster W = {A1, A2} in G, then subgraph(M - {A1} + {Ω}, G’) is isomorphic to subgraph(M, G).

As a simple illustration, consider again the clustering example of Fig. 6(c) and (d). Let G and G’ respectively denote the graphs of Fig. 6(c) and (d), and in Fig. 6(c), let M = {A, B}, A1 = A, and A2 = C. Then M - {A1} + {Ω} = {B, Ω}, and clearly, subgraph(M, G) is isomorphic to subgraph([{A, B}], G). The proof of lemma 5 can be found in the appendix.

Lemma 6: Suppose that G is a strongly connected SDF graph, and Z is a strongly connected subset of nodes in G such that q2(Z) = k. Suppose Z1 and Z2 are disjoint subsets of Z such that Z1 is a subindependent of Z2 in subgraph(Z). Then Z1 is also subindependent of Z2 in G.

Proof: For each arc α directed from a member of Z2 to a member of Z1, we have delay(α) ≥ total.consumed(α, subgraph(Z)). From fact 3, q2(Z)(N) = q2(G)(N) for all N ∈ Z. Thus, for all arcs α in subgraph(Z), total.consumed(α, subgraph(Z)) = total.consumed(α, G), and we conclude that Z1 is a subindependent of Z2 in G. QED.

Lemma 7: Suppose G is a strongly connected SDF graph, A and B are distinct nodes in G, and W = {A, B} forms a proper subset of N(G). Suppose also that the following conditions all hold:
1) Neither A nor B is contained in a tightly interdependent subgraph of G.
2) There is at least one arc directed from A to B that has no delay.
3) B has no predecessors other than A or B.
4) q2(G)(C) = kq2(C) for some C ∈ N(G), C ≠ B.

Then the SDF graph G’ that results from clustering W is loosely interdependent.

Proof: From (1) G must be loosely interdependent, so there exist subsets X1, X2 of N(G) such that X1 ⊆ X2. If A, B ∈ X1 or A, B ∈ X2, then from lemma 4, we are done. Now condition (2) precludes the scenario (B ∈ X1, A ∈ X2), so the only remaining possibility is (A ∈ X1, B ∈ X2). There are two cases to consider here:

i) B is not the only member of X2. Then from (3), (X1 + {B}) ∪ G(X2 - {B}). But A, B ∈ (X1 + {B}), so lemma 4 again guarantees that G’ is loosely interdependent.

ii) A is not the only member of X1 and X2 = {B}. Thus we have X1 ⊆ G(B), so

∀α ∈ A(G), (source(α) = B) ⇒ delay(α) ≥ total.consumed(α, G). (1)

Also, since C ∈ X1, we have from (4) that q2(X1) = gcd(q2(G)(N) ∩ X1) = gcd(q2(G)(N)N ∈ X1) ∪ {kq2(C)} = gcd(q2(G)(N)N ∈ X1) ∪ {kq2(G)(B)} =

However, G’ may be deadlocked even if G is not. This will not be a problem in our application of lemma 4.
\textit{gcd}\left(\{q_G(N)|N \in N(G)\}\right) = 1. \text{ That is,}
\[ q_G(X_1) = 1. \tag{2} \]

Now if \( X_1 \) is not strongly connected, then it has a proper subset \( Z \) such that there are no arcs directed from a member of \( (X_1 - Z) \) to a member of \( Z \). Furthermore, from condition (3), \( A \not\in Z \). This is true because if \( Z \) contains \( A \), then no member of \( (X_1 - Z) \) would have a path to \( B \), and thus \( G \) would not be strongly connected. Thus \( A \in (X_1 - Z) \), and there are no arcs directed from \( (X_1 - Z) \) to \( Z \). So all arcs directed from \( (X_1 - Z + \{B\}) \) to \( Z \) have node \( B \) as their source. From (1) it follows that \( Z\langle G, (X_1 - Z + \{B\}) \rangle \). Now \( A, B \in (X_1 - Z + \{B\}) \), so applying lemma 4 we conclude that \( G' \) is loosely interdependent.

If \( X_1 \) is strongly connected, we know from condition (1) that there exist \( Y_1, Y_2 \) such that \( Y_1 \subseteq Y_2 \). From (2) and lemma 6, \( Y_1 \) is a subunipendent of \( Y_2 \) in \( G \). Now if \( A \in Y_1 \), then from condition (3), \( B \) is a subunipendent of \( Y_2 \) in \( G \), so from fact 6(a), \( (Y_1 \cup \{B\}) \) is a subunipendent of \( Y_2 \) in \( G \). Applying lemma 4, we see that \( G' \) is loosely interdependent. On the other hand, suppose that \( A \in Y_2 \). From (1), we know that \( Y_1 \) is a subunipendent of \( (Y_2 \cup \{B\}) \) in \( G \). From fact 6(b), it follows that \( Y_1 \) is a subunipendent of \( (Y_2 \cup \{B\}) \), so again we can apply lemma 4 to conclude that \( G' \) is loosely interdependent.

\textbf{QED}

\textbf{Theorem 4:} Suppose \( G \) is a connected SDF graph, and \( A \) and \( B \) are distinct nodes in \( G \) such that \( B \) is a successor of \( A \), and \( W = \{A, B\} \) is a proper subset of \( N(G) \). If we cluster \( W \) in \( G \) then the tightly interdependent components of \( G' \) are the same as the tightly interdependent components of \( G \) if the following conditions all hold:

1. Neither \( A \) nor \( B \) is contained in a tightly interdependent component of \( G \).
2. At least one arc directed from \( A \) to \( B \) has zero delay.
3. \( q_G(B) = kq_G(A) \) for some positive integer \( k \).
4. \( B \) has no predecessors other than \( A \) and \( B \).

\textbf{Proof:} It suffices to show that all strongly connected subgraphs in \( G' \) that contain \( \Omega \) are loosely interdependent. So we suppose that \( Z' \) is a strongly connected subset of \( N(G') \) that contains \( \Omega \), and let \( Z \) denote the "corresponding" subset in \( G \) that is, \( Z = Z' \setminus \{\Omega\} + \{A, B\} \). Now in \( Z' \), suppose that there is a directed circuit \( C \) containing the node \( \Omega \). From condition (4), this implies that there is a directed circuit in \( G \) containing \( A, C, D, \) and possibly \( B \). The two possible ways in which a directed circuit in \( G \) introduces a directed circuit involving \( \Omega \) in \( G' \) are illustrated in Fig. 8(a) and (b); the situation in (c) cannot arise because of condition (4).

Now in \( Z' \), if one or more of the circuits involving \( \Omega \) corresponds to Fig. 8(a), then \( Z \) must be strongly connected. Otherwise, all of the circuits involving \( \Omega \) correspond to Fig. 8(b), so \( (Z - \{B\}) \) is strongly connected, and from condition (4), no member of \( (Z - \{A, B\}) \) is adjacent to \( B \). In the former case, lemma 7 yields the loose interdependence of \( Z' \).

In the latter case, lemma 5 guarantees that \( (Z - \{B\}) \) is isomorphic to \( Z' \). Since \( A \in (Z - \{B\}) \), and since from condition (1), \( A \) is not contained in any tightly interdependent subgraph of \( G \), it follows that \( Z' \) is loosely interdependent.

\textbf{QED}

If we assume that the input SDF graph has a single appearance schedule then we can ignore condition (1). From our observations, this is a valid assumption for the vast majority of practical SDF graphs. Also, condition (3) can be verified by examining any single arc directed from \( A \) to \( B \); if \( \alpha \) is directed from \( A \) to \( B \) then condition (3) is equivalent to \( p(\alpha) = k(\alpha) \). In our current implementation, we consider only the case \( k = 1 \) for condition (3) because in practice, this corresponds to most of the opportunities for efficiently using registers.

We see that the clustering process defined by theorem 4—under the assumption that the original graph has a single appearance schedule—requires only local dataflow information, and thus it can be implemented very efficiently. If our assumption that a single appearance schedule exists is wrong, then we can always undo our clustering decisions. Since the assumption is frequently valid, and since it leads to a very efficient algorithm, this is the form in which we have implemented theorem 4. Finally, in addition to making data transfers more efficient, our clustering process provides a fast way to reduce the size of the graph without canceling the existence of a single appearance schedule. When used as a preprocessing technique, this can sharply reduce the execution time of a loose interdependence algorithm.

\section{Conclusions}

This paper has presented fundamental topological relationships between iteration and looping in SDF graphs, and we have shown how to exploit these relationships to synthesize the most compact looping structures for a large class of applications. Furthermore, we have extended the developments of [5] by showing how to isolate the minimal subgraphs that require explicit deadlock detection schemes, such as the reachability matrix, when organizing hierarchy.

This paper also defines a framework for evaluating different scheduling schemes having different objectives, with regard to their effect on schedule compactness. The developments of this paper apply to any scheduling algorithm that imposes hierarchy on the SDF graph. For example, by successively repeating the same block of code, we can reduce "context-switch" overhead [22]. We can identify subgraphs that use as much of the available hardware resources as possible, and these can be clustered, as the computations to be repeatedly invoked. However, the hierarchy imposed by such a scheme must be evaluated against its impact on program compactness.
For example, if a cluster introduces tight interdependence, then it may be impossible to fit the resulting program on chip, even though the original graph had a sufficiently compact schedule.

The techniques developed in this paper have been successfully incorporated into a block-diagram software synthesis environment for DSP [18]. We are currently investigating how to systematically incorporate these techniques into other scheduling objectives—for example, how to balance parallelization objectives with program compactness constraints.

APPENDIX

This appendix contains proofs of some the lemmas that were stated and used in Sections IV–VI.

Proof of Lemma 1:

From remark 1, if \( N \) is not contained in a nontrivial strongly connected component of \( G \), the result is obvious, so we assume, without loss of generality, that \( N \) is in some nontrivial strongly connected component \( H_1 \) of \( G \). From our assumptions, \( \text{subgraph}(H_1) \) must be loosely interdependent, so \( \lambda \) partitions \( H_1 \) into \( X \) and \( Y \), where \( X \cup \text{subgraph}(H_1) Y \). Let \( H' \) denote that connected component of \( \text{subgraph}(X) \) or \( \text{subgraph}(Y) \) that contains \( N \). From remark 3, \( \text{appearances}(N, S_X(G)) = \text{appearances}(N, S_Y(\text{subgraph}(H'))) \).

From our assumptions, all nontrivial strongly connected subgraphs of \( H' \) that contain \( N \) are loosely interdependent. Thus, if \( N \) is contained in a nontrivial strongly connected component \( H_2 \) of \( H_1 \), then \( \lambda \) will partition \( H_2 \) and we will obtain a proper subset \( H_2' \) of \( H_1' \) such that \( \text{appearances}(N, S_X(\text{subgraph}(H'])) = \text{appearances}(N, S_X(\text{subgraph}(H'))) \). Continuing in this manner, we get a sequence \( H_1', H_2', \ldots \) of subsets of \( N(G) \) such that each \( H_i' \) is a proper subset of \( H_{i-1}' \) and \( N \) is contained in each \( H_i' \). Since \( H_i' \) is a strict subset of its predecessor, we can continue this process only a finite number, say \( m \), of times. Then \( N \in H_m', N \) is not contained in a nontrivial strongly connected component of \( \text{subgraph}(H_m') \), and \( \text{appearances}(N, S_X(G)) = \text{appearances}(N, S_X(\text{subgraph}(H_m'))) \). But from remark 1, \( S_X(\text{subgraph}(H_m')) \) contains only one appearance of \( N \).

QED

Proof of Lemma 5:

Let \( C = \text{subgraph}(M - \{ A_1 \} + \{ \Omega \}, G') \), let \( \Phi \) denote the set of arcs in \( \text{subgraph}(M, G) \), and let \( \Phi' \) denote the set of arcs in \( C \). From (1), every arc in \( C \) has a corresponding arc in \( \text{subgraph}(M, G) \) and vice-versa, and thus \( \Phi' = \{ \alpha' | \alpha \in \Phi \} \). Now from the definition of clustering a subgraph, we know that \( p(\alpha') = p(\alpha) \) for any arc \( \alpha \in \Phi \) such that \( \text{source}(\alpha) \neq A_1 \). If \( \text{source}(\alpha) = A_1 \), then \( \alpha \) is replaced by \( \alpha' \) with \( \text{source}(\alpha') = \Omega \), and \( p(\alpha') = p(\alpha)q(\text{fa}(\alpha))/\text{gcd}(q(\text{fa}(\alpha), q(A_1)) \).

But \( \text{gcd}(q(\text{fa}(\alpha), q(A_2)) = \text{gcd}(q(A_1), kq(A_1)) = q(A_1) \), so \( p(\alpha') = p(\alpha) \). Thus \( p(\alpha') = p(\alpha) \) for all \( \alpha \in \Phi \). Similarly, we can show that \( c(\alpha') = c(\alpha) \) for all \( \alpha \in \Phi \). Thus, the mappings \( f_1: M \rightarrow N(C) \) and \( f_2: \Phi \rightarrow \Phi' \) defined by

\[
\begin{align*}
    f_1(N) &= N \text{ if } N \neq A_1, \\
    f_1(A_1) &= \Omega, \text{ and } f_2(\alpha) = \alpha'
\end{align*}
\]

demonstrate that \( \text{subgraph}(M, G) \) is isomorphic to \( C \). QED

GLOSSARY

\( Z_1 \cap GZ \): If \( G \) is an SDF graph and \( Z_1 \) and \( Z_2 \) form a partition of the nodes in \( G \) such that \( Z_1 \) is subindependent of \( Z_2 \) in \( G \), then we write \( Z_1 \cap GZ \).

\( A(G) \): The set of arcs in the SDF graph \( G \).

\( \text{appearances}(N, S) \): The number of times that actor \( N \) appears in the looped schedule \( S \).

\( \text{admissible schedule} \): A schedule \( S_1, S_2, \ldots, S_n \) such that each \( S_i \) has sufficient input data to fire immediately after its antecedents \( S_1, S_2, \ldots, S_{i-1} \) have fired.

\( c(\alpha) \): The number of samples consumed from SDF arc \( \alpha \) by one invocation of \( \text{sink}(\alpha) \).

\( \text{delay}(\alpha) \): The number of delays on SDF arc \( \alpha \).

\( \text{gcd} \): Greatest common divisor.

\( N(G) \): The set of nodes in the SDF graph \( G \).

\( \text{PASS} \): A periodic admissible sequential schedule.

\( p(\alpha) \): The number of samples produced onto SDF arc \( \alpha \) by one invocation of \( \text{source}(\alpha) \).

\( \text{periodic schedule} \): A schedule that invokes each node at least once and produces no net change in the number of samples buffered on any arc.

\( \text{predecessor} \): Given two nodes \( A \) and \( B \) in an SDF graph, \( A \) is a predecessor of \( B \) if there is at least one arc directed from \( A \) to \( B \).

\( q_G \): The repetitions vector \( q_G \) of the SDF graph \( G \) is a vector that is indexed by the nodes in \( G \). \( q_G \) has the property that every \( \text{PASS} \) for \( G \) invokes each node \( N \) a multiple of \( q_G(N) \) times.

\( \text{single appearance schedule} \): A schedule that contains only one appearance of each actor in the associated SDF graph.

\( \text{sink}(\alpha) \): The actor at the sink of SDF arc \( \alpha \).

\( \text{source}(\alpha) \): The actor at the source of SDF arc \( \alpha \).

\( \text{subgraph} \): A subgraph of an SDF graph \( G \) is the graph formed by any subset \( Z \) of nodes in \( G \) together with all arcs \( \alpha \) in \( G \).
for which \( \text{source}(\alpha), \text{sink}(\alpha) \in Z \). We denote the subgraph corresponding to the subset of nodes \( Z \) by \( \text{subgraph}(Z, G) \), or simply by \( \text{subgraph}(Z) \) if \( G \) is understood from context.

**subindependent**: Given an SDF graph \( G \), and two disjoint subsets \( Z_1, Z_2 \) of nodes in \( G \), we say that \( Z_1 \) is subindependent of \( Z_2 \) in \( G \) if for every arc \( \alpha \) in \( G \) with \( \text{source}(\alpha) \in Z_2 \) and \( \text{sink}(\alpha) \in Z_1 \), we have \( \text{delay}(\alpha) \geq \text{total consumed}(\alpha, G) \). We say that \( Z_1 \) is subindependent in \( G \) if \( Z_1 \) is subindependent of \( (N \setminus G) - Z_1 \) in \( G \).

**successor**: Given two nodes \( A \) and \( B \) in an SDF graph, \( A \) is a successor of \( B \) if there is at least one arc directed from \( B \) to \( A \).

**total consumed**(\( \alpha, G \)): The total number of samples consumed from arc \( \alpha \) in a minimal schedule period of the SDF graph \( G \); that is, \( \text{total consumed}(\alpha, G) = q_0(\text{sink}(\alpha)) - \text{source}(\alpha) \).

**valid schedule**: A schedule that is a PASS.

---

**REFERENCES**


Edward A. Lee (S'80-M'86-SM'93-F'94) received the B.S. from Yale University in 1979, the M.S. from the Massachusetts Institute of Technology in 1981, and the Ph.D. from the University of California, Berkeley, in 1986.

From 1979 to 1982, he was a member of technical staff at Bell Telephone Laboratories in Holmdel, NJ, in the Advanced Data Communications Laboratory. At present, he is a Professor in the Electrical Engineering and Computer Science Department at the University of California, Berkeley. His research activities include real-time software, parallel computation, architecture and software techniques for signal processing, and design methodology for heterogeneous systems. He is Director of the Ptolemy project at UC Berkeley, and previously directed the Gabriel project. He is a founder of Berkeley Design Technology, Inc. and has consulted for a number of other companies. He is co-author of *Digital Communication* (Kluwer Academic Press, 1988 first ed., 1994 second ed.), and co-author of *Digital Signal Processing Experiments* (Prentice-Hall, 1989), as well as numerous technical papers.

Dr. Lee was recently Chairman of the VLSI Technical Committee of the Signal Processing Society, and Co-Program Chair of the 1992 Application Specific Array Processor Conference. He is an Associate Editor of *Design Automation for Embedded Systems* and is on the editorial board of the *Journal on VLSI Signal Processing*. As a Fellow of the IEEE, he has received the citation “For contributions to design methodologies and programming techniques for real-time digital signal processing systems.” He was a recipient of a 1987 NSF Presidential Young Investigator award, an IBM faculty development award, the 1986 Sällfors Prize at U.C. Berkeley, and a paper award from the IEEE Signal Processing Society.