SINGLE APPEARANCE SCHEDULES FOR SYNCHRONOUS DATAFLOW PROGRAMS

by

Shuvra S. Bhattacharyya, Soonhoi Ha, and Edward A. Lee

Memorandum No. UCB/ERL M93/4

10 January 1993

SINGLE APPEARANCE SCHEDULES FOR SYNCHRONOUS DATAFLOW PROGRAMS

by

Shuvra S. Bhattacharyya, Soonhoi Ha, and Edward A. Lee

Memorandum No. UCB/ERL M93/4

10 January 1993

Title Page

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SINGLE APPEARANCE SCHEDULES FOR SYNCHRONOUS DATAFLOW PROGRAMS

by

Shuvra S. Bhattacharyya, Soonhoi Ha, and Edward A. Lee

Memorandum No. UCB/ERL M93/4

10 January 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SINGLE APPEARANCE SCHEDULES FOR SYNCHRONOUS DATAFLOW PROGRAMS

Shuvra S. Bhattacharyya
Soonhoi Ha
Edward A. Lee

Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

Synchronous Dataflow (SDF) provides block-diagram semantics that are well-suited to compiling multirate signal processing algorithms onto programmable signal processors. A key to this match is the ability to cleanly express iteration without overspecifying the execution order of blocks, 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 hierarchical clustering strategy that provably synthesizes the most compact nested loop structure for a large class of applications.

1 INTRODUCTION

In the Dataflow model of computation, pioneered by Dennis [5], a program is managed as a directed graph in which the nodes represent computations and the arcs specify the passage of data. Synchronous Dataflow (SDF) [14] 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 used extensively to synthesize assembly code for signal processing applications, for example [7, 8, 9, 17, 18, 19].

In SDF, *iteration* is defined as the repetition induced when 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) [11]. For example, in figure 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, assembly 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 [10]. Thus, it is often 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]. Many programming languages used for DSP, such as Lucid[22], SISAL[15], and Silage[8] 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 [6] Dennis shows how convert recursive stream functions in SISAL-2 into SDF graphs.

In [10], How evaluated augmenting schedulers that did not consider looping with a post-processing phase that detects successively occurring repetitive firing patterns, and concluded that such simple tactics were ineffective for generating compact programs. To synthesize loops effectively, the scheduler must exploit specific topological properties in the SDF graph. How demonstrated such a property by showing that we can often greatly improve looping by consolidating subgraphs that operate at the same sample rate, and scheduling such subgraphs as a single unit. Figure 1 shows how this technique can improve looping. A naive scheduler might schedule this SDF graph as CABCB, which offers no looping possibility within the schedule period. However, if we first group the subgraph induced by {B,C} into a hierarchical "supernode" Γ, a scheduler will generate the schedule A$\Gamma\Gamma$. To highlight the repetition in a schedule, we let the notation $(NX_1X_2...X_m)$ designate N successive repetitions of the firing sequence $X_1X_2...X_m$. We refer to a

schedule expressed with this notation as a **looped schedule**, and we refer to each term of the form $(N\ X_1X_2...X_m)$ as a **schedule loop**. Using this notation, and substituting each occurrence of Γ with a subschedule for the corresponding subgraph, our consolidation of the uniform-rate set {B,C} leads to either A(2BC) or A(2CB), both of which expose the full amount of looping in the SDF graph of figure 1.

We explored the looping problem further in [3]. 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 consolidating 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 invocation count. By not discriminating against sample-rate boundaries, our approach exposed looping more thoroughly than How's scheme. Furthermore, by selecting subgraphs based on repetition rate, we reduced data memory requirements, an aspect that How's scheme did not consider.

Consolidating a subgraph must be done with care since certain groupings cause deadlock. For example, combining C and D in figure 2 results in a graph for which no periodic schedule

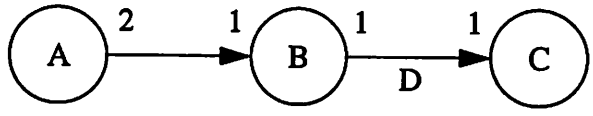

Fig 1. An example that illustrates the benefits of consolidating uniform sample-rate subgraphs. Each arc is annotated with the number of samples produced by its source and the number of samples consumed by its sink. The "D" designates a unit delay.

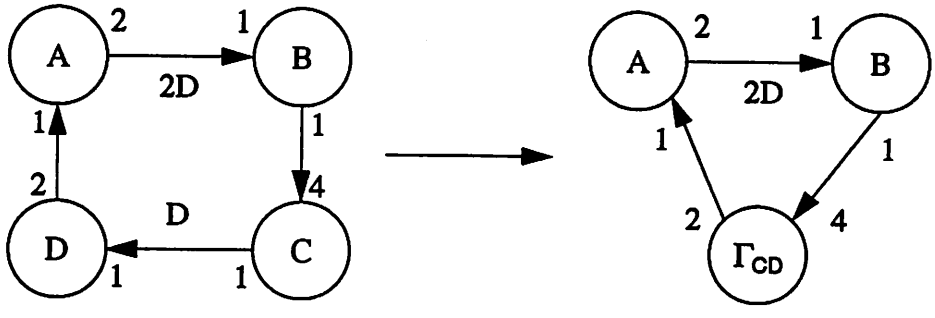

Fig 2. An example of how consolidating a subgraph in an SDF graph can result in deadlock.

exists because the grouping "hides" a critical delay. Similarly, deadlock can be introduced when a grouping encapsulates a source actor. 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 [13], however this approach is extremely time consuming if a large number of consolidations must be considered. In [3], 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 [3]. 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 often involves a sample-rate change on the order of 250000 to 1. If the rasterization output is connected to an actor that consumes only one token per invocation (for example, a gamma level correction), this actor alone will produce on the order of $(250000)^2 = 6.25 \times 10^{10}$ 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 [3] 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 figure 3 illustrates, this goal can conflict with consolidating subgraphs based on repetition count.

In this paper, we develop an efficient method for extracting the most compact looping structure from the cyclic paths in the SDF graph. This technique is based on a topological quality that we call "loose interdependence". We show that for SDF graphs that are loosely interdependent, our method is optimal. Interestingly and fortunately, a large majority of practical SDF graphs seem to fall into this category. Furthermore, for this class of graphs, our algorithm does not

require use of the reachability matrix, or any other unreasonably large data structure. For graphs that are not loosely interdependent, we show that our algorithm naturally isolates the minimal subgraphs which 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. We emphasize that the techniques developed in this paper extend the developments of [3] by improving the analysis of cyclic subgraphs. In particular, our earlier method still applies to acyclic subgraphs for organizing looping while keeping buffering requirements low. However, when it is used only for acyclic graphs, deadlock is not an issue, and the reachability matrix is no longer required.

Because we focus on the fundamental limits of looping, the methods developed in this paper cannot be directly applied to the general parallel processing case. However, we believe that these techniques will be helpful to understanding problems that combine parallelization and looping objectives, and we are currently investigating such problems. The techniques of this paper do apply to target systems that exploit instruction-level parallelism, such as superscalar and pipelined architectures.

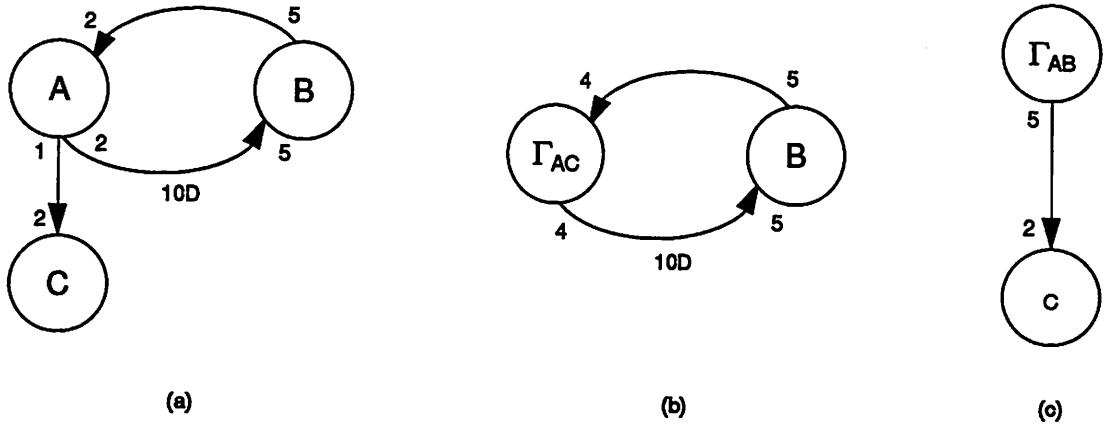

Fig 3. This example illustrates how consolidating subgraphs based on repetition count alone can conceal looping opportunities that occur within cyclic paths. Part (a) depicts a multirate SDF graph. Two pairwise subgraphs exist — {A, B}, having repetition count 2, and {A, C}, having repetition count 5. Consolidating the subgraph with the highest repetition count yields the hierarchical topology in (b), for which the most compact schedule is $(2B)(2\Gamma_{AC})B\Gamma_{AC}B(2\Gamma_{AC}) \Rightarrow$ (2B)(2(2A)C)B(2A)CB(2(2A)C). Consolidating the subgraph {A,B} of lower repetition rate, as depicted in part (c), yields the more compact schedule $(2\Gamma_{AB})(5C) \Rightarrow$ (2(2B)(5A))(5C).

2 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 periodic schedules.

For an SDF graph G, we denote the set of nodes in G by $N(G)$. If G' is a subgraph of G, then we can obtain another subgraph of G by removing from G all nodes in G' and all arcs that have one or both endpoints in G'. We call this subgraph the complement of G' in G, and we denote it by G – G'. Also, if $N_s \subseteq N(G)$, then we write $S(N_s, G)$ to denote the subgraph induced by N_s in G. For an SDF arc α, we let "source(α)" and "sink(α)" denote the nodes at the source and sink of α; we let "p(α)" denote the number of samples produced by source(α), "c(α)" denote the number of samples consumed by sink(α), and we denote the delay on α by "delay(α)". Finally, if x and y are two nodes in and SDF graph, we say that x is a **successor** of y if there is an arc directed from y to x, and we say that x is a **predecessor** of y if y is a successor of x.

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(α) tokens from the head of the FIFO for each input arc α, and appending p(β) tokens to the FIFO for each output arc β. 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 $S_1\ S_2\ \ldots\ S_N$ of nodes in G such that each S_i is fireable immediately after $S_1, S_2, \ldots, S_{i-1}$ have fired in succession. If some S_i is not fireable immediately after its antecedents, then the schedule is not admissable, and we say that the schedule *deadlocks just prior to* S_i. Finally, we say that an admissible sequential schedule S is a *periodic admissible sequential schedule* (PASS) if it invokes each node at least once, and it produces no net change in the number of tokens on a FIFO — for each arc α, (the number of times source(α) is fired in S) × p(α) = (the number of times sink(α) is fired in S) × c(α). We will use the term *valid schedule* to describe a schedule that is a PASS.

For a given periodic schedule, we denote the *i*th *firing*, or *invocation*, of actor N by N_i, and if *f* is a firing in some schedule, we denote the actor associated with *f* by actor(*f*) (e.g. actor(N_i) = N).

In [13], it is shown that for each SDF graph G that has a PASS, there is a mapping $q_G : N_G \rightarrow \{1, 2, 3, \ldots\}$ such that every PASS for G invokes each node *n* a multiple of $q_G(n)$ times. More specifically, corresponding to each PASS S, there is a positive integer *J* called the *blocking factor* of S, such that S invokes each $n \in N(G)$ exactly $Jq_G(n)$ times. We call this mapping q_G the *repetitions vector* of G. The following properties of repetitions vectors are proved in [13]:

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

Fact 2: If S is an admissible schedule for G, and there is a positive integer *J* such that S invokes each $n \in N(G)$ exactly $Jq_G(n)$ times, then S is a PASS.

Fact 3: For each arc α in G, $q_G(\text{source}(\alpha)) \times p(\alpha) = q_G(\text{sink}(\alpha)) \times c(\alpha)$.

We will also use the following property, which is derived in [4]:

Fact 4: If G_s is a subgraph of G, and *n* is a node in G_s, then $kq_{G_s}(n) = q_G(n)$, where $k = \gcd\{q_G(m) \mid m \in N(G_s)\}$.

For our hierarchical scheduling approach, we will apply the concept *of consolidating a subgraph*, which was introduced in [12]. This process is illustrated in figure 3. Here the subgraph {A, C} of (a) is consolidated into the hierarchical node Γ_{AC}, and the resulting SDF graph is shown in (b). Similarly, consolidating subgraph {A, B} results in the graph of (c). Each input arc α to a consolidated subgraph Γ is replaced by an arc α' having $p(\alpha') = p(\alpha)$, and $c(\alpha') = c(\alpha) \times q_\Gamma(\text{sink}(\alpha))$, the number of samples consumed from α in one *invocation of subgraph* Γ. Similarly we replace each output arc β with β' such that $c(\beta') = c(\beta)$, and $p(\beta') = p(\beta) \times q_\Gamma(\text{source}(\beta))$. The following property of consolidated subgraphs is proven in [4].

Fact 5: Suppose G is an SDF graph, G' is the SDF graph that results from consolidating a connected subgraph Γ of G, S is a PASS for G', and S_Γ is a pass for Γ. Then replacing each appearance of Γ in S with S_Γ results in a PASS for G.

The possibility of a self-loop, an arc whose source node is the same as its sink, introduces minor technical complications in our development. However, without loss of generality, we can assume that self-loops do not exist, and doing so, we can formulate our results more concisely. This assumption is valid because a self-loop either introduces deadlock or imposes no sequencing constraints on the construction of a PASS.

Unless otherwise stated, we assume that an SDF graph contains no self-loops. For example, when we say "Let G be an SDF graph ...", we mean "Let G be an SDF graph with all self-loops removed ...". From an implementation standpoint, this means that the input SDF graph is first preprocessed to remove all self -loops, which, as discussed above, does not affect the subsequent scheduling process. An important consequence of our assumption is that every strongly connected subgraph contains at least two nodes.

3 SINGLE APPEARANCE SCHEDULES

To determine the limits of looping for a general SDF graph, we have found it instructive to determine the topological conditions required for a the existence of a looped schedule that contains only a single appearance for each actor. We refer to such a schedule as a **single appearance schedule**. For example, the schedule CA(2B)C for figure 1 is not a single appearance schedule since C appears twice. Thus, either C must be implemented with a subroutine, or we must insert two versions of C's code block into program memory. In the schedule A(2CB) however, no actor appears more than once, so it is a single appearance schedule, and it translates into the most compact program for the given SDF graph.

Since single appearance schedules implement the full repetition inherent in an SDF graph without requiring subroutines or code duplication, we examine the topological conditions required for such a schedule to exist. First suppose that G is an *acyclic* SDF graph containing N

nodes. Then we can take some root node r_1 of G and fire all $q_G(r_1)$ invocations of r_1 in succession. After all invocations of r_1 have fired, we can remove r_1 from G, pick a root node r_2 of the new acyclic graph, and schedule its $q_G(r_2)$ repetitions in succession. Clearly, we can repeat this process until no nodes are left to obtain the single appearance schedule $(q_G(r_1)\ r_1)\ (q_G(r_2)\ r_2)\ \ldots\ (q_G[r_N]\ r_N)$ for G. Thus we see that any acyclic graph has a single appearance schedule.

Also, observe that if G is an arbitrary SDF graph, then we can consolidate the subgraphs associated with each strongly connected component of G. Consolidating a strongly connected component into a single block never results in deadlock since there can be no directed loop containing the consolidated block. Since consolidating connected components yields an acyclic graph, it follows from fact 5 that *G has a valid single appearance schedule if and only if each strongly connected component has a valid single appearance schedule.*

Observe that we must, in general, analyze a strongly connected component G_c as a separate entity, since G may have a single appearance schedule even if there is a node n in G_c for which we cannot fire all $q_G[n]$ invocations in succession.The key is that q_{G_c} may be less than q_G, so we may be able to generate a single appearance subschedule for G_c (e.g. we may be able to schedule n $q_{G_c}(n)$ times in succession). Since we can schedule G so that G_c's subschedule appears only once, this will translate to a single appearance schedule for G. For example, in figure 3, it is can be verified that $q_G(A) = 10$ and $q_G(B) = 4$; but so many invocations of A or B cannot be fired in succession. However, consider the strongly connected component Γ_{AB} consisting of nodes A and B. Then we obtain $q_{\Gamma_{AB}}(A) = 5$ and $q_{\Gamma_{AB}}(B) = 2$, and we immediately see that $q_{\Gamma_{AB}}(B)$ invocations of B can be scheduled in succession to obtain a subschedule for Γ_{AB}. This leads to the single appearance schedule given in the caption of figure 3.

In this section, we develop important properties of single appearance schedules. In section 4, we will use these properties to develop our looping techniques and prove their optimality of our looping techniques. We begin with a lemma. The terminology introduced in this lemma will be use throughout the rest of this section.

Lemma 1: Suppose that G is an SDF graph, S is a valid looped schedule for G, and L is a schedule loop within S. Let A(L) denote the set of actors that appear in L, and let M be any maximal

connected subset of A(L) (the subgraph associated with M is connected and no node in A(L) – M is adjacent to a node in M). Remove from L all actors that are not in M, remove any empty loops that result, and call the resulting schedule loop L_1 — we call L_1 *the* **restriction** *of the schedule loop* L *to the set of actors* M. Similarly, let L_2 denote the restriction of L to A(L) – M. Finally, let S' denote the schedule obtained by replacing L in S with L_1 L_2. Then S' is a valid schedule for G.

For example, suppose that G is the SDF graph in Figure 4, and suppose we are given the schedule S = M(3Y(2AB)CZ) for G. Let L denote the outer loop in this schedule, (3Y(2AB)CZ). Then M_1 = {A, B, C} and M_2 = {Y, Z} are two maximal connected subgraphs that partition A(L). Now we remove the members of M_1 from L to obtain (3Y(2)Z), and from this we remove the empty loop "(2)" to obtain L_1 = (3YZ). Similarly, we remove Y and Z from L to obtain L_2 = (3(2AB)C). Lemma 1 states that if M(3Y(2AB)CZ) is a valid schedule for the graph in figure 4, then so are M(3YZ)(3(2AB)C) and M(3(2AB)C)(3YZ).

Proof of lemma 1: Suppose that S' deadlocks just prior to some invocation *i* of actor X. If we define P(*x*, *y*, *s*) to be the number of firings of actor *x* that precede invocation *y* in schedule/subschedule *s*, then clearly there exists an arc α such that

(1) $\text{sink}(\alpha) = X_i$ and $P(\text{source}(\alpha), X_i, S') < P(\text{source}(\alpha), X_i, S)$.

Now the sequence of invocations fired in S can be divided into $(s_1\ l_1\ s_2\ l_2\ \ldots\ l_N\ s_{N+1})$, where l_i is the sequence of firings associated with the *i*th invocation of loop L, and s_i is the

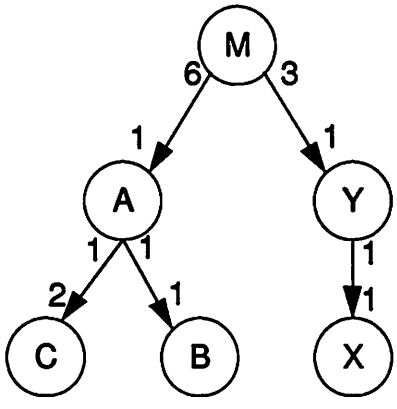

Fig 4. An illustration of lemma 1. Note that the repetition counts for A, B, C, M, X, Y are respectively 6, 6, 3, 1, 3, 3.

sequence of firings between the $(i-1)$th and ith invocations of L. Since S' is derived by rearranging the firings in L, we can express it similarly as $(s_1\ l_1'\ s_2\ l_2'\ \ldots\ l_N'\ s_{N+1})$, where l_i' corresponds to the ith invocation of $(L_1\ L_2)$ in S'.

Now, the firing sequence generated by L_1 (or L_2) is simply the firing sequence generated by L with the invocations associated with nodes in $A(L_1)$ (or $A(L_2)$) removed. Thus,

(2) $P(N, f, S) = P(N, f, S')$ if $(N, \text{actor}(f) \in A(L_1))$ or $(N, \text{actor}(f) \in A(L_2))$.

Also, the number of firings of each actor in l_i' is the same as the corresponding number in l_i, so

(3) for $i = 1, 2, \ldots, N+1$ and for any node N in G, $f \in s_i \Rightarrow P(N, f, S) = P(N, f, S')$; and

(4) for $i = 1, 2, \ldots, N$ and for any node $N \notin A(L)$, $f \in l_i \Rightarrow P(N, f, S) = P(N, f, S')$.

It follows from (1) and (3) that S' can not deadlock in an s_i; i.e. $X_i \in l_j'$ for some j, and then from (4) it follows that source(α) $\in$ A(L). Then $P(\text{source}(\alpha), X_i, l_j') < P(\text{source}(\alpha), X_i, l_j)$, so from (2), either source(α) $\in A(L_1)$ and $X \in A(L_2)$, or source(α) $\in A(L_2)$ and $X \in A(L_1)$. I.e. *either* source(α) $\in A(L) - M$ and $X \in M$ *or* source(α) $\in M$ and $X \in A(L) - M$. Since M is a maximally connected subset of A(L), this contradicts the adjacency of source(α) and X = sink(α). Thus our assumption that S' deadlocks cannot hold. *QED*.

Repeated application of lemma 1 to each maximally connected subgraph immediately yields the following consequence.

Corollary 1: Suppose that G is an SDF graph, S is a valid looped schedule for G and L is a schedule loop in S. Let $M_1, M_2, \ldots, M_n$ denote the set of maximally connected subgraphs of $S(A(L), G)$, and for $i = 1, 2, \ldots, n$, let L_i denote the restriction of L to M_i. Then the schedule obtained by replacing L in S with $L_1\ L_2 \ldots L_n$ is a valid schedule for G.

Definition 1: We define the **nesting degree** *of a schedule loop* L, denoted ND(L), to be the maximum loop nesting depth within L. To be precise, ND(L) = 1 if no loops are nested within L; otherwise, $ND(L) = 1 + \max\{ND(L') \mid L' \text{ is a loop that is nested within } L\}$. Similarly we define the

nesting degree *of a looped schedule* S, denoted ND(S), to be zero if S contains no schedule loops; if S contains at least one schedule loop the we define ND(S) to be max{ND(L) | L is a schedule loop in S} — in other words ND(S) is the maximum nesting degree over all loops in S, which is equivalent to the maximum nesting degree over all outermost loops in S. For example, the schedule loop (3 AB(2BC)D(2A(2B))) has nesting degree 3, and the looped schedule AB(2C(3A)B)C has nesting degree 2.

Definition 2: Let S be a looped schedule or a subschedule for an SDF graph G. We say that S is **regular** if for every schedule loop L in S, A(L) forms a connected subgraph of G. If n is a nonnegative integer, we say that S is ***n*-regular** if for every loop L in S whose nesting degree is less than n, A(L) is connected. Thus, S is regular $\Leftrightarrow$ S is (ND(S) + 1)-regular.

Definition 3: Let S be a looped schedule for an SDF graph G and let n be a node in G, then we define #appearances(n, S) to be the number of times that n appears in S. For example, #appearances(C, CA(2B)C) = 2, and S is a single appearance schedule $\Leftrightarrow$ #appearances(n, S) = 1 $\forall$ n.

Lemma 2: Suppose that G is an SDF graph and suppose that there exists a valid n-regular looped schedule S for G. If S is not regular then there exists an $(n + 1)$-regular valid looped schedule S' for G such that ND(S') = ND(S), and for every actor m in G, #appearances(m, S') = #appearances(m, S).

Note that it is trivial to construct an $(n + 1)$-regular S' if we do not require ND(S') = ND(S). We can do this simply by replacing each loop λ of nesting degree n by the loop (1 λ). To increase the "degree of regularity" without constructing a more deeply nested schedule, we repeatedly apply lemma 1.

Proof of Lemma 2: Let ω denote the schedule loops in S that are not associated with connected subgraphs of G and whose nesting degree is n: $\omega = \{\lambda \mid ND(\lambda) = n$ and $A(\lambda)$ is not connected$\}$. From corollary 1, we can replace each schedule loop $X \in \omega$ by a sequence of loops $X_1, X_2, \ldots, X_{N_x}$, where each $A(X_i)$ forms a maximal connected subgraph of A(X), and each X_i is the restriction of X to $A(X_i)$.

Now suppose that L is a loop properly contained in some X_i (X_i contains L but $X_i \neq L$). Then L is the restriction to $A(X_i)$ of some L' nested within X. Since this L' is nested in X, ND(L') < ND(X) = n, so from the n-regularity of S, we know that A(L') is connected. Since $A(X_i)$ forms a maximally connected subgraph of A(X) and A(L') forms a connected subgraph of A(X), it follows that $A(L') \subset A(X_i)$, and thus L' = L, which implies that A(L) is connected.

Since each $A(X_i)$ is connected, and each loop L properly contained in X_i has the property that A(L) is connected, it follows that each subschedule $X_1\ X_2 \ldots\ X_{N_x}$ is regular. Furthermore $ND(X_i) = ND(X)$, so replacing X by $X_1\ X_2 \ldots\ X_{N_x}$ does not increase the nesting degree of the overall schedule. We conclude that by replacing each $X \in \omega$ in S with $X_1\ X_2 \ldots\ X_{N_x}$, we obtain an (n + 1)-regular schedule S' such that ND(S') = ND(S).

Finally, since X_i is the restriction of X to $A(X_i)$, and since $A(X_1)$, $A(X_2)$, ..., $A(X_{N_x})$ are disjoint, each actor in A(X) appears in exactly one X_i, and it appears the same number of times in that X_i as it appears in X. Thus each actor appears the same number of times in S' as it does in S. *QED.*

We will apply the following extension of lemma 2.

Corollary 2: Suppose that there exists a valid single appearance schedule for G. Then there exists a valid single appearance schedule for G that is regular.

Proof: Let S be a valid single appearance schedule for G and let n = ND(S). S is trivially 1-regular, so if n=0, we are done. Otherwise, repeated application of lemma *2* guarantees the existence of valid single appearance schedule with nesting degree n that are 1-regular, 2-regular, ..., (n + 1)-regular. In particular, there exists a valid single appearance schedule S' such that ND(S') = n, and S' is (n + 1)-regular $\Rightarrow$ S' is regular. *QED.*

Lemma 3: Suppose that S is an admissible single appearance schedule for G and suppose that L = $(M\ (N_1\ S_1)\ (N_2\ S_2) \ldots\ (N_m\ S_m))$ is a schedule loop within S (of any nesting depth) such that each $A(S_i)$ forms a connected subgraph G_i of G. Let $\gamma = gcd(N_1, N_2, \ldots, N_m)$, and let L' denote the loop $(\gamma M\ (\gamma^{-1}N_1\ S_1)\ (\gamma^{-1}N_2\ S_2) \ldots\ (\gamma^{-1}N_m))$. Then replacing L with L' in S results in an admissible schedule for G.

Proof. Suppose S has blocking factor k. Clearly, each S_i is a PASS of some blocking factor v_i for G_i, and we have

$$\textbf{(5)}\ \forall x \in A(S_i),\ M \times N_i \times v_i \times q_{G_i}(x) = k \times q_G(x)$$

Let S' denote the looped schedule obtained by replacing L with L' and suppose that S' deadlocks just prior to invocation r of actor X. Since L and L' invoke each actor the same number of times, we have $\forall$ Y $\in$ N(G), and for any invocation I fired outside of L in S, P(Y, I, S) = P(Y, I, S'). Thus X $\in$ A(L) = A(L'); i.e. X $\in$ $N(G_a)$, for some $a \in \{1, 2, \ldots m\}$. Also, for any actor Z $\notin$ A(L), we have P(Z, X_r, S) = P(Z, X_r, S'). Thus there exists a predecessor X' $\in$ A(L) of X, and an arc θ directed from X' to X such that in S', delay(θ) + (*the total number of samples produced onto* θ *by* X' *prior to* X_r) < (*the total number of samples consumed from* θ *by the first* r *invocations of* X). Since each S_i is an admissible schedule for the associated G_i, X' cannot be in G_a. So X' $\in$ $N(G_b)$, $b \neq a$. The graphical relationship between X and X' is illustrated in figure 5.

Now let R denote the total number of invocations of the loop (N_a S_a) that have completed prior to the rth invocation of X (R = floor((r − 1) / ($N_a \times q_{G_a}(x) \times v_a$)). Then, if ($a > b$) — i. e. S_b lexically precedes S_a — we have

$$\text{delay}(\theta) + (R+1) \times \gamma^{-1} N_b \times q_{G_b}(x') \times v_b \times p(\theta) < (R+1) \times \gamma^{-1} N_a \times q_{G_a}(x) \times v_a \times c(\theta)$$

$$\Rightarrow N_b \times q_{G_b}(x') \times v_b \times p(\theta) < N_a \times q_{G_a}(x) \times v_a \times c(\theta).$$

Multiplying both sides by M and applying (5) gives:

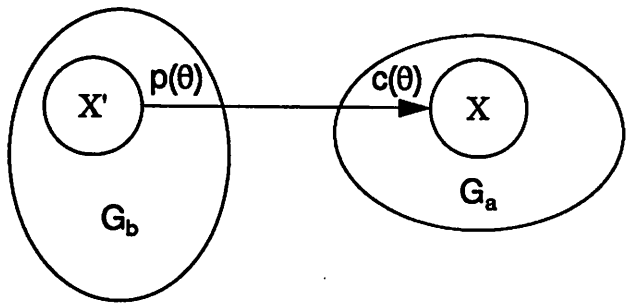

Fig 5. An illustration of X, X', and θ in the proof of lemma 3.

$$k \times q_G(x') \times p(\theta) < k \times q_G(x) \times c(\theta)$$

$$\Rightarrow q_G(x') \times p(\theta) \neq q_G(x) \times c(\theta),$$

which contradicts fact 3.

On the other hand, if $(a < b)$, then

$$\text{delay}(\theta) + R \times \gamma^{-1} N_b \times q_{G_b}(x') \times v_b \times p(\theta) < (R+1) \times \gamma^{-1} N_a \times q_{G_a}(x) \times v_a \times c(\theta).$$

Multiplying both sides by γM, applying (5) and the balance equation $q_G(x') \times p(\theta) = q_G(x) \times c(\theta)$ (fact 3) gives

$$\gamma \times \text{delay}(\theta) < N_a \times q_{G_a}(x) \times v_a \times c(\theta).$$

Since $\gamma \geq 1$, this implies that

$$\text{delay}(\theta) < N_a \times q_{G_a}(x) \times v_a \times c(\theta).$$

Now, the right side of this inequality is the number of samples that X consumes from θ in each invocation of the loop $(N_a\ S_a)$. Since $a < b$ — $(N_a\ S_a)$ lexically precedes $(N_b\ S_b)$ — it follows that S will deadlock before completing the first invocation of $(N_a\ S_a)$. This contradicts our assumption that S is an admissible schedule. *QED*

We will apply the following consequence of lemma 3.

Corollary 3: Assume the same hypotheses as in lemma 3 with the additional assumption that each S_i has blocking factor 1 with respect to the associated G_i. Then there exists a looped schedule of the form $(M^*\ S^*)$, where S^* is a single appearance schedule of *unity blocking factor* for $\bigcup_{i=1}^{m} G_i$ such that replacing L with $(M^*\ S^*)$ in S results in another valid single appearance schedule for G.

Proof. From lemma 3, replacing L with $(\gamma M\ (\gamma^{-1} N_1\ S_1)\ (\gamma^{-1} N_2\ S_2)\ \ldots\ (\gamma^{-1} N_m S_m))$ gives an admissible schedule for G. Thus $S' = (\gamma^{-1} N_1\ S_1)\ (\gamma^{-1} N_2\ S_2)\ \ldots\ (\gamma^{-1} N_m S_m)$ is PASS for $\bigcup_{i=1}^{m} G_i$. Let z denote the blocking factor for S' and suppose that $z > 1$. For each node x in G, let $\rho(x)$ denote the number of times x is invoked in one period of S'. Then clearly $\rho(x) = z q_G(x)$, $\forall\, x \in N(G)$. Since each S_i has blocking factor 1, we also have

(6) for $j = 1, 2, \ldots m, x \in A(S_j) \Rightarrow \rho(x) = \gamma^{-1}N_j\, q_{G_j}(x)$.

Now since $\gcd\{\gamma^{-1}N_1, \gamma^{-1}N_2, \ldots, \gamma^{-1}N_m\} = 1, \exists\, i$ such that z does not divide $\gamma^{-1}N_i$. Then from (6), there exists a nontrivial factor of z that divides $q_{G_i}(x)$ — i.e. there exists an integer $z' > 1$ such that z' divides z, and z' divides every component of $q_{G_i}(x)$. But this contradicts fact 1. Thus our assumption that $z > 1$ cannot hold, and we conclude that S' is a valid single appearance schedule of blocking factor 1 for $\bigcup_{i=1}^{m} G_i$. Furthermore, replacing L with (γM S') gives a valid single appearance schedule for G. *QED.*

Corollary 3 shows that we can replace several looped unit blocking factor subschedules with a loop across a single unit blocking factor subschedule. Starting at the innermost loops, and repeatedly applying corollary 3, we can show that from any valid single appearance schedule, we can generate a valid single appearance schedule of unit blocking factor. The following definition helps to prove this result concisely.

Definition 4: By a **simple loop**, we mean a schedule loop of the form (N A), where A is an actor appearance. For example (3 (2AB(3C))(2D)) contains two simple loops — (3C) and (2D). We say that a looped schedule is **simple** if every actor appearance is surrounded by a simple loop.

Theorem 1: Suppose that G is a connected SDF graph that has a valid single appearance schedule (of arbitrary blocking factor). Then G has a valid single appearance schedule of blocking factor 1.

Proof. Suppose that S_0 is a valid single appearance schedule of arbitrary blocking factor for G. From corollary 2, there exists a valid single appearance schedule S_1 of the same blocking factor that is regular (each loop spans a connected subset of nodes). Substituting each actor appearance A in S_1 with (1 A) preserves regularity and does not change the firing sequence. Thus, without loss of generality, we can assume that S_1 is simple. Let L_1 be an innermost non-simple loop of (1 S_1) (If all loops in (1 S_1) are simple, then (1 S_1) is a simple loop $\Rightarrow$ G contains only one node, which trivially yields the desired result). Then L_1 has the form $(M_1\ (N_1\ Z_1)\ (N_2\ Z_2)\ \ldots\ (N_m\ Z_m))$, where each Z_i is a node in G. Each "Z_i" is clearly a valid single appearance schedule for $S(\{Z_i\}, G)$, the subgraph associated with the single node $\{Z_i\}$. Thus we can apply corollary 3 to substitute

L_1 with $(M_1' T_1)$, where T_1 is a valid single appearance schedule of blocking factor 1 for $S(A(L_1), G)$.

Let S_2 denote the schedule that results from replacing L_1 with $(M_1' T_1)$ in $(1 S_1)$. Since S_1 is regular, $S(A(M'_1 T_1), G) = S(A(L_1), G)$ is connected, so S_2 is regular; also, corollary 3 guarantees that S_2 is a valid single appearance schedule for G. Now from S_2 select a schedule loop $L_2 = (M_2 Z_{2,1} Z_{2,2} \ldots Z_{2,m2})$ such that for each i, $Z_{2,i}$ is either a simple loop, or $Z_{2,i} = (M'_1 T_1)$. Then S_2, L_2 satisfy the hypotheses of corollary 3, so we can substitute L_2 by some $(M'_2 T_2)$, where T_2 is a valid single appearance schedule of blocking factor 1 for $S(A(L_2), G)$. Let S_3 denote the resulting schedule. Then corollary 3 and the regularity of S_2 guarantee that S_3 is a valid regular single appearance schedule G.

Clearly we can repeat this process until we have visited all non simple loops in $(1 S_1)$. At step k, we select a schedule loop $L_k = (M_k L_{k,1} L_{k,2} \ldots L_{k,mk})$ from S_k such that each $L_{k,i}$ is either a simple loop or $L_{k,i} \in \{(M_1' T_1), (M_2' T_2), \ldots, (M_{k-1}' T_{k-1})\}$, and we apply corollary 3 and the regularity of S_k to obtain a replacement $(M_k' T_k)$ for L_k, such that T_k has blocking factor 1. This replacement yields a valid regular single appearance schedule S_{k+1}.

After some number R steps, we will have consolidated all loops in $(1 S_1)$ into $(M_k' T_k)$'s. Thus $S_R = (M'_R T_R)$ is a valid single appearance schedule for G, and T_R is a unit blocking factor schedule for $S(A(T_R), G)$. But $S(A(T_R), G) = S(A(M'_R T_R), G) = S(A(1 S_1), G) = S(N(G), G) = G$. So T_R is a valid single appearance schedule for G that has blocking factor 1. *QED*

Clearly, any schedule S of unity blocking factor can be converted into a schedule of arbitrary blocking factor k simply by encapsulating S inside a loop of k iterations. Thus from theorem 1, we can conclude that given an SDF graph G, and given a positive integer k, a valid single appearance schedule of blocking factor k exists for G if and only if valid single appearance schedules exist for all blocking factors.

We introduce the following terminology to develop the precise condition required for a strongly connected graph to have a single appearance schedule. Recall that a general SDF graph has a single appearance schedule if and only if each strongly connected component has a single

appearance schedule, so the condition for a strongly connected SDF graph specifies the condition for a general SDF graph.

Definition 5: Suppose that G is a strongly connected SDF graph. If x and y are nodes in G and x is a successor of y, then we say that x *is subindependent of* y *in* G if for every arc α directed from y to x, we have $delay(\alpha) \geq c(\alpha) \times q_G(x)$. Also we say that a proper and nonempty subgraph G_s of G is *subindependent in* G if G_s is connected and for every node x in G that is a successor of a node y in $G - G_s$, x is subindependent of y in G (we often drop the "in G" qualification if G is understood from context). In other words G_s is subindependent if no samples produced outside of G_s are consumed from G_s in the same schedule period. If G_1 and G_2 partition G and G_1 is subindependent, we say that G_1 is *subindependent of* G_2 *in* G, and we denote this by "$G_1 |_G G_2$", or "$G_1 | G_2$", if G is understood.

We are now ready to establish a recursive condition for the existence of a single appearance schedule.

Theorem 2: Suppose that G is a strongly connected SDF graph that contains more than one node. Then G has a single appearance schedule if and only if

(1) G contains a subindependent subgraph G_s; and

(2) G_s and $(G - G_s)$ both have a single appearance schedules.[1]

Proof: $\Leftarrow$ Let S_1 and S_2 denote single appearance schedules for G_s and $G - G_s$ respectively. From theorem 1, we can assume without loss of generality that S_1 and S_2 both have unit blocking factor. Let $R_1 = \gcd\{q_G(n) \mid n \in G_s\}$, let $R_2 = \gcd\{q_G(n) \mid n \in G - G_s\}$, and let S_R denote the looped schedule $(R_1\ S_1)\ (R_2\ S_2)$. Then from fact 4, it follows that S_R invokes each node $n \in N(G)$ exactly $q_G(n)$ times, and from the subindependence of G_s, S_R is an admissible schedule for G. Applying fact 2, we conclude that S_R is a PASS, and hence it is a valid single appearance schedule.

$\Rightarrow$ Suppose that S is a single appearance schedule for G. Again, from theorem 1, we can assume without loss of generality that S has blocking factor 1. Then S can be expressed as S_aS_b, where S_a and S_b are nonempty single appearance subschedules of S that are not encom-

1. Note that $(G - G_s)$ is not necessarily connected.

passed by a loop (if we could represent S as a single loop (N (...) (...) ... (...)) then $\gcd\{q_G(x) \mid x \in G\} \geq N$, so S is not of unity blocking factor — a contradiction). Furthermore, repeatedly applying corollary 1, we can separate subschedule S_a into a succession $T_1T_2 \ldots T_m$ of one or more single appearance subschedules, where the set of nodes involved in each T_i form a connected subgraph M_i and each distinct pair M_i, M_j is not connected. Thus $T_1T_2 \ldots T_mS_b$ is a valid single appearance schedule for G. In this schedule, every actor x that appears in T_1 is fired $q_G(x)$ times before any node outside of M_1 is invoked. It follows that M_1 is subindependent of $G - M_1$. Also T_1 is a single appearance schedule for M_1 and $T_2T_3 \ldots T_mS_b$ is a single appearance for $G - M_1$. *QED*.

In the following section, we will use this theorem to decompose strongly connected components in a manner that preserves the looping structure inherent in the SDF graph.

4 LOOSE INTERDEPENDENCE

Theorem 2 implies that for an SDF graph to have a single appearance schedule, we must be able to decompose it into two subgraphs, one of which is subindependent of the other. In this section we show how this topological property and its converse can be used to generate compact looped schedules. We begin with a definition.

Definition 6: Suppose that G is a strongly connected SDF graph. Then we say that G is **loosely interdependent** if G can be partitioned into subgraphs G_1 and G_2 such that $G_1 \mid_G G_2$. We say that G is **tightly interdependent** if it is not loosely interdependent.

The properties of loose/tight interdependence are important for organizing loops because, as we will show, the existence of a single appearance schedule is equivalent to the absence of a tightly interdependent subgraph. However, these properties can be used even when tightly interdependent subgraphs are present. The following definition specifies how to use loose interdependence to guide the looping process. The remainder of this paper is devoted mainly to demonstrating the effectiveness of this approach.

Definition 7: Let A_1 be any algorithm that takes as input a strongly connected SDF graph G, determines whether G is loosely interdependent, and if so, finds a subindependent subgraph in 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 as input a tightly interdependent SDF graph that has a PASS, and generates a valid looped schedule of blocking factor 1 for that graph. We define the algorithm $L(A_1, A_2, A_3, A_4)$ as follows:

Input: an SDF graph G that has a PASS.
Output: a valid unit-blocking-factor looped schedule $S_L(G)$ for G.
Step 1: Use A_2 to determine the strongly connected components $G_1, G_2, \ldots, G_s$ of G.
Step 2: Consolidate $G_1, G_2, \ldots, G_s$ into subgraphs, and call the resulting graph G'. This is an acyclic graph.
Step 3: Apply A_3 to G'; denote the resulting schedule S'(G).
Step 4:

for $i=1, 2, \ldots, s$
Apply A_1 to G_i;
if subgraphs $X=X(G_i)$, $Y=Y(G_i)$ are found such that $X|_{G_i}Y$,
then
- Recursively apply algorithm L to subgraph X; the resulting schedule is denoted $S_L(X)$.
- Recursively apply algorithm L to subgraph Y; the resulting schedule is denoted $S_L(Y)$.
- Let $r_x = \gcd\{q_G(n) \mid n \in N(X)\}$.
- Let $r_y = \gcd\{q_G(n) \mid n \in N(Y)\}$.
- Replace the (single) appearance of G_i in S'(G) with $(r_x\ S_L(X))\ (r_y\ S_L(Y))$[1].

else (G_i is tightly interdependent)
- Apply A_4 to obtain a valid schedule S_i for G_i.
- Replace the single appearance of G_i in S with S_i.

end-if
end-for

The ***for***-loop replaces each "G_i" in S'(G) with a valid looped schedule for G_i. From repeated application of fact 5, we know that these replacements yield a valid looped schedule S_L for G. We output S_L.■

1. It follows from fact 4 and the definition of loose interdependence that this is a PASS for G_i.

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

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

Remark 3: For each strongly connected component G_i that is loosely interdependent, L partitions G_i into X and Y such that $X|_{G_i} Y$, and replaces the single appearance of G_i in S'(G) with $S^* = (r_x\ S_L(X))\ (r_y\ S_L(Y))$. If $m \in N(X)$, then $m \notin N(Y)$, so #appearances(m, S^*) = #appearances(m, $S_L(X)$). Also since m cannot be in any other strongly connected component besides G_i, and since S'(G) is a single appearance schedule, we have #appearances(m, $S_L(G)$) = #appearances(m, S^*). Thus, $m \in N(X) \Rightarrow$ #appearances(m, $S_L(G)$) = #appearances(m, $S_L(X)$). By the same argument, we can show that $m \in N(Y) \Rightarrow$ #appearances(m, $S_L(G)$) = #appearances(m, $S_L(Y)$).

L(•, •, •, •) 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* λ, the *strongly connected components algorithm of* λ, the *acyclic scheduling algorithm of* λ, and the *tight scheduling algorithm of* λ.

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 need to introduce two lemmas.

Lemma 4: Suppose G is an SDF graph; n is a node in G that is not contained in any tightly interdependent subgraph of G; and λ is a loose interdependence algorithm. Then n appears only once in $S_\lambda(G)$, the schedule generated by λ.

Proof. From remark 1, if n is not contained in a strongly connected component of G, the result is obvious, so we assume, without loss of generality, that n is in some strongly connected component H_1 of G. From our assumptions, H_1 must be loosely interdependent, so λ partitions H_1 into $X(H_1)$ and $Y(H_1)$, where $X(H_1) \mid_{H_1} Y(H_1)$. Let H_1' denote that member of $\{X(H_1), Y(H_1)\}$ that contains n. From remark 3, #appearances(n, $S_\lambda(G)$) = #appearances(n, $S_\lambda(H_1')$).

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

Lemma 5: Suppose that G is a strongly connected SDF graph, P is a subindependent subgraph in G, and C is a strongly connected subgraph of G such that $C \cap P \neq C$ and $C \cap P \neq \emptyset$. Then $C \cap P$ is subindependent in C.

Proof. Suppose that α is an arc directed from a node in $C \cap (G - P)$ to a node in $C \cap P$. By the subindependence of P in G, $delay(\alpha) \geq c(\alpha) \times q_G(sink(\alpha))$, and by fact 4, $q_G(sink(\alpha)) \geq q_C(sink(\alpha))$. Thus, $delay(\alpha) \geq c(\alpha) \times q_C(sink(\alpha))$. Since this holds for any α directed from $C \cap (G - P)$ to $C \cap P$, we conclude that $C \cap P$ is subindependent in C. *QED.*

Corollary 4: Suppose that G is a strongly connected SDF graph, G_1 and G_2 are subgraphs such that $G_1 \mid_G G_2$, and T is a tightly interdependent subgraph of G. Then $T \subset G_1$ or $T \subset G_2$.

Proof. Suppose that T has nonempty intersection with both G_1 and G_2. Then from lemma 5, $T \cap G_1$ is subindependent in T. Thus T is loosely interdependent. Contradiction.

Theorem 3: Suppose that G is a strongly connected SDF graph that has an admissible schedule.Then G has a single appearance schedule iff every strongly connected subgraph of G is loosely interdependent.

Proof. $\Leftarrow$ Suppose every strongly connected subgraph of G is loosely interdependent, let λ be any loose interdependence algorithm, and let S denote the resulting schedule for G. Since no node in G is contained in a tightly interdependent subgraph, it follows from lemma 4 that $S_\lambda(G)$ is a single appearance schedule for G.

$\Rightarrow$ Suppose that G has a single appearance schedule and that G contains a tightly interdependent subgraph C. From theorem 2, we can partition G into X_0 and Y_0 such that X_0 is subindependent of Y_0 and X_0 and 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 X_0 or Y_0. We can then apply theorem 1 to partition Z_1 into X_1 and Y_1, and continue recursively in this manner until we obtain a strongly connected subgraph $Z_k \subset G$ with the following property: Z_k can be partitioned into X_k and Y_k such that $X_k \cap C$ and $Y_k \cap C$ partition C, and X_k is subindependent of Y_k in Z_k. From lemma 5, $X_k \cap C$ is subindependent of $Y_k \cap C$, and thus C is loosely interdependent. Contradiction. *QED.*

Corollary 5: Given an 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 3, G contains no tightly interdependent subgraphs. In other words, no node in G is contained in a tightly interdependent subgraph of G. From lemma 4, 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 G into two connected 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 $\{G_1, G_2\}$ and $\{G_1', G_2'\}$ into connected subgraphs such that $G_1 \mid G_2$ and $G_1' \mid G_2'$, 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 6: Suppose that G_1 and G_2 are tightly interdependent SDF graphs and $G_1 \cap G_2 \neq \emptyset$. Then $(G_1 \cup G_2)$ is tightly interdependent.

Proof. Suppose that $H = G_1 \cup G_2$ is loosely interdependent. Then there exist subgraphs H_1 and H_2 such that $H = H_1 \cup H_2$ and $H_1 \mid H_2$. From $H_1 \cup H_2 = G_1 \cup G_2$, and $G_1 \cap G_2 \neq \emptyset$, it is easily seen that H_1 and H_2 both have a nonempty intersection with G_1, or they both have a nonempty intersection with G_2. Without loss of generality, assume that $H_1 \cap G_1 \neq \emptyset$ and $H_2 \cap G_1 \neq \emptyset$. Since G_1 is tightly interdependent, there exists an arc α such that $source(\alpha) \in G_1 \cap H_2$, $sink(\alpha) \in G_1 \cap H_1$, and $delay(\alpha) < q_{G_1}(sink(\alpha)) \times c(sink(\alpha))$, Since $G_1 \subset H$, it follows from fact 4 that $q_{G_1}(sink(\alpha)) \leq q_H(sink(\alpha))$. Thus, $source(\alpha) \in H_2$, $sink(\alpha) \in H_1$, and $delay(\alpha) < q_H(sink(\alpha)) \times c(sink(\alpha))$, so H_1 is not subindependent of H_2. Contradiction.

Lemma 6 implies that each SDF graph G has a *unique* set $\{C_1, C_2, \ldots, C_n\}$ of maximal tightly interdependent subgraphs such that $i \neq j \Rightarrow C_i \cap C_j = \emptyset$, and every tightly interdependent subgraph in G is contained in some C_i. We call each C_i a *tightly interdependent component* of G. It follows from theorem 3 that G has a single appearance schedule iff 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 4: Suppose G is an SDF graph that has a PASS, m is a node in G, and λ is a loose interdependence algorithm. If m is not contained in a tightly interdependent component of G, then m appears only once in $S_\lambda(G)$. On the other hand, if m is contained in a tightly interdependent component T then #appearances(m, $S_\lambda(G)$) = #appearances(m, $S_\lambda(T)$) — the number of appearances of m is determined entirely by the tight scheduling algorithm of λ.

Proof. If m is not contained in a tightly interdependent component of G, then m is not contained in any tightly interdependent subgraph. Then from lemma 4, #appearances(m, $S_\lambda(G)$) = 1.

Now suppose that m is contained in some tightly interdependent component T of G. We set $M_0 = G$, and suppose that $T \neq M_0$. By definition, tightly interdependent graphs are strongly connected, so T is contained in some strongly connected component C of M_0.

If $T \neq C$ — i.e. T is a proper subgraph of C — then C must be loosely interdependent, since otherwise T would not be a maximal tightly interdependent subgraph. Thus, λ partitions C into $X(C)$ and $Y(C)$ such that $X(C) \mid_C Y(C)$. We set M_1 to be that member of $\{X(C), Y(C)\}$ that contains m. Since $X(C)$, $Y(C)$ partition C, M_1 is a proper subgraph of M_0. Also, from remark 3, #appearances(m, $S_\lambda(M_0)$) = #appearances(m, $S_\lambda(M_1)$), and from corollary 4, $T \subseteq M_1$.

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

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

Theorem 4 states that the tight scheduling algorithm is independent of the subindependence partitioning algorithm, and vice-versa. Any subindependence 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 subindependence partitioning algorithm that is more efficient in some way (e.g. it is faster, takes into account vectorization, or minimizes data memory requirements), we can replace it for any existing subindependence partitioning algorithm without changing the "compactness" of the resulting schedules — we don't need 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*.

5 COMPUTATIONAL EFFICIENCY

The complexity of a loose interdependence algorithm λ depends on its subindependence partitioning algorithm λ_{sp}, strongly connected components algorithm λ_{sc}, acyclic scheduling algorithm λ_{as}, and tight scheduling algorithm λ_{ts}. From the proof of theorem 4, we see that λ_{ts} is applied exactly once for each tightly interdependent component. Thus an efficient tight scheduling algorithm will not contribute to intractability. For example, the technique of [3] can be applied as the tight scheduling algorithm. This technique involves a hierarchical clustering phase that has time complexity[1] O(number of arcs × number of nodes), followed by a scheduling phase that is linear in the total number of firings. One drawback of this algorithm, as mentioned in section 1, is that it requires a reachability matrix, which has quadratic storage cost. 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, λ_{sc}, λ_{as}, and λ_{sp}, 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

1. In the worst case, every arc corresponds to a cluster, and each clusterization step requires a reachability-matrix update that is linear in the number of nodes.

the original program has N nodes, N decomposition steps are required in the worst case.Tarjan [21] 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 λ_{sc} can be chosen to be linear, and since at most $N \leq M$ decomposition steps are required, the total time that such an λ_{sc} accounts for in λ is $O(M^2)$. In section 3 we presented a simple linear-time algorithm that constructs a single appearance schedule for an acyclic SDF graph. Thus λ_{as} can be chosen such that its total time is also $O(M^2)$.

The following theorem presents a simple topological condition for loose interdependence that leads to a linear subindependence partitioning algorithm λ_{sp}.

Theorem 5: Suppose that G is a strongly connected SDF graph. From G, remove all arcs α for which $delay(\alpha) \geq c(\alpha) \times q_G(sink(\alpha))$, and call the resulting SDF graph G'. Then G is tightly interdependent if and only if G' is strongly connected.

Proof. $\Rightarrow$ Suppose that G' is not strongly connected. Then G' can be partitioned into G_1' and G_2' such that there are no arcs directed from G_2' to G_1'. Since no nodes were removed in constructing G', $N(G_1')$ and $N(G_2')$ partition $N(G)$. Also, none of the arcs directed from $S(N(G_2'), G)$ to $S(N(G_1'), G)$ in G occur in G'. Thus, by the construction of G', for each arc α directed from a node in $S(N(G_2'), G)$ to a node in $S(N(G_1'), G)$, we have $delay(\alpha) \geq c(\alpha) \times q_G(sink(\alpha))$. It follows that G is loosely interdependent.

$\Leftarrow$ Suppose that G is loosely interdependent. Then G can be partitioned into G_1 and G_2 such that $G_1 |_G G_2$. By construction of G', $N(G_1)$ and $N(G_2)$ partition $N(G')$, and there are no arcs in G' directed from $S(N(G_2), G')$ to $S(N(G_1), G')$. Thus G' is not strongly connected. *QED.*

Thus, λ_{sp} can be constructed as follows: (1) Determine $q_G(n)$ for each node n; (2) Remove each arc α whose delay is at least $c(\alpha) \times q_G(sink(\alpha))$; (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) consolidate the strongly connected components — the resulting graph is acyclic and has at least two nodes. Any root node of this graph is subindependent of the rest of the graph. It is easily seen that (1) and (2) can be performed in time $O(M)$;

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 λ_{sp}, and the total time that λ spends in λ_{sp} is $O(M^2)$.

We have specified λ_{sp}, λ_{sc}, λ_{as}, and λ_{ts} such that each accounts for $O(M^2)$ time. The resulting loose interdependence algorithm is thus of quadratic worst-case complexity. 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. For most applications, the running time of the algorithm will scale linearly with the size of the input graph.

6 CONCLUSION

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 structure for a large class of applications. Furthermore, we have extended the developments of [3] by showing how to isolate the 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 repeatedly the same block of code, we can reduce "context-switch" overhead, and thus improve throughput [19]. We can identify subgraphs that use as much of the available hardware resources as possible, and these can be consolidated or "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.

We have incorporated the techniques of this paper into a block-diagram-based software synthesis environment that has been developed in our research group [16]. 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. Other important tradeoffs to examine include vectorization, as discussed above, and data memory requirements.

ACKNOWLEDGEMENT

The authors gratefully acknowledge enlightening discussions with Joseph Buck, which helped to put this work in better perspective.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, Mass., 1974.

[2] Arvind, L. Bic, T. Ungerer, "Evolution of Data-Flow Computers", Chapter 1 in *Advanced Topics In Data-Flow Computing*, edited by J. L. Gaudiot and L. Bic, Prentice Hall, 1991.

[3] S. S. Bhattacharyya, E. A. Lee, "Scheduling Synchronous Dataflow Graphs For Efficient Looping", To appear in *Journal of VLSI Signal Processing*, 1992.

[4] S. S. Bhattacharyya, "Clustering Formalism for Synchronous Dataflow", Technical Report, Memorandum No. UCB/ERLM92/30, University of California at Berkeley, April 1992.

[5] J. B. Dennis, "First Version of a Dataflow Procedure Language", *MIT/LCS/TM-61*, Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge MA 02139.

[6] J. B. Dennis, "Stream Data Types for Signal Processing", Technical Report, September 1992.

[7] D. Genin, J. De Moortel, D. Desmet, E. Van de Velde, "System Design, Optimization, and Intelligent Code Generation for Standard Digital Signal Processors", *ISCAS*, Portland, Oregon, May 1989.

[8] P. N. Hilfinger, "Silage Reference Manual, Draft Release 2.0", Computer Science Division, EECS Dept., University of California at Berkeley, July 1989.

[9] W. H. Ho, E. A. Lee, D. G. Messerschmitt, "High Level Dataflow Programming for Digital Signal Processing", *VLSI Signal Processing III*, IEEE Press 1988.

[10] S. How, "Code Generation for Multirate DSP Systems in GABRIEL", Master's Degree Report, U. C. Berkeley, May 1988.

[11] E. A. Lee, "Static Scheduling of Dataflow Programs for DSP", *Advanced Topics in Dataflow Computing*, edited by J. L. Gaudiot and L. Bic, Prentice-Hall, 1991.

[12] E. A. Lee, "A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors", Ph.D. Thesis, University of California at Berkeley, May 1986.

[13] E. A. Lee, D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing", *IEEE Transactions on Computers, January 1987*.

[14] E. A. Lee, D. G. Messerschmitt, "Synchronous Dataflow", *Proceedings of the IEEE*, September 1987.

[15] J. R. McGraw, S. K. Skedzielewski, S. Allan, D. Grit, R. Oldehoft, J. Glauert, I. Dobes, P. Hohensee, "SISAL: Streams and Iteration in a Single Assignment Language", Language Reference Manual, Version 1.1., July 1983

[16] J. Pino, S. Ha E. A. Lee, J. T. Buck, "Software Synthesis for DSP Using Ptolemy", To appear in *Journal of VLSI Signal Processing*.

[17] D. B. Powell, E. A. Lee, W. C. Newmann, "Direct Synthesis of Optimized DSP Assembly Code From Signal Flow Block Diagrams", *ICASSP*, San Francisco, California, March 1992.

[18] H. Printz, "Automatic Mapping of Large Signal Processing Systems to a Parallel Machine", Memorandum CMU-CS-91-101, School of Computer Science, Carnegie-Mellon University, May 1991, PhD Thesis.

[19] S. Ritz, M. Pankert, H. Meyr, "High Level Software Synthesis for Signal Processing Systems", Proceedings of the International Conference on Application Specific Array Processors, Berkeley, CA, August 1992.

[20] G. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication", PhD Thesis, University of California at Berkeley, 1991.

[21] R. E. Tarjan, "Depth First Search and Linear Graph Algorithms", *SIAM J. Computing*, June 1972.

[22] W. W. Wadge, E. A. Ashcroft, "Lucid, the Dataflow Language", Academic Press, 1985.