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

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

[12] 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.

[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] D. R. O' Hallaron, "The ASSIGN Parallel Program Generator", Technical Report, Memorandum Number CMU-CS-91-141, School of Computer Science, Carnegie Mellon University, May 1991.

[17] J. Pino, S. Ha E. A. Lee, J. T. Buck, "Software Synthesis for DSP Using Ptolemy", To appear in *Journal of VLSI Signal Processing*, 1993.

[18] 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.

[19] 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.

[20] 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.

[21] S. Ritz, M. Pankert, H. Meyr, "Optimum Vectorization of Scalable Synchronous Dataflow Graphs", Technical Report IS2/DSP93.1a, Aachen University of Technology, Germany, January 1993.

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

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

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

sink(α) The actor at the sink of SDF arc α.

source(α) The actor at the source of SDF arc α.

subgraph A *subgraph* of an SDF graph G is the graph formed by any subset Z of nodes in G together with all arcs α in G for which *source*(α), *sink*(α) $\in$ Z. We denote the subgraph corresponding to the subset of nodes Z by *subgraph*(Z, G), or simply by *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 α in G with *source*(α) $\in Z_2$ and *sink*(α) $\in Z_1$, we have *delay*(α) $\geq$ *total_consumed*(α, G). We say that Z_1 is subindependent in G if Z_1 is subindependent of (N(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(α, G) The total number of samples consumed from arc α in a minimal schedule period of the SDF graph G; that is, *total_consumed*(α, G) $= \mathbf{q}_G(sink(\alpha))c(\alpha)$.

valid schedule A schedule that is a PASS.

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, "Looped Schedules for Dataflow Descriptions of Multirate DSP Algorithms", Technical Report, Electronics Research Laboratory, College of Engineering, University of California, Berkeley CA 94720.

[4] S. S. Bhattacharyya, E. A. Lee, "Scheduling Synchronous Dataflow Graphs For Efficient Looping", To appear in *Journal of VLSI Signal Processing*, 1993.

[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, 1975.

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

[7] G. R. Gao, R. Govindarajan, P. Panangaden, "Well-Behaved Programs for DSP Computation", *ICASSP*, San Francisco, California, March 1992.

[8] 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.

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

```
end-for

** multiplier contains LCM({denom(Q(N)) | N ∈ N(G)}). **

for each node N in G
      Set Q(N) = reduced_form(multiplier × Q(N)).
end-for
Output Q(N) as qG(N) for each node N in G.
```

Glossary

$Z_1 \mid_G Z_2$ — 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 $Z_1 \mid_G Z_2$.

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

appearances(N, S) — The number of times that actor N appears in the looped schedule S.

admissable schedule — A schedule $S_1\ S_2 \ldots S_k$ 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(α) — The number of samples consumed from SDF arc α by one invocation of *sink*(α).

delay(α) — The number of delays on SDF arc α.

gcd — Greatest common divisor.

looped schedule — A schedule that has zero or more parenthesized terms of the form $(n\ \Psi_1\ \Psi_2 \ldots \Psi_k)$, where *n* is a nonnegative integer, and each Ψ_i represents either an SDF node or another parenthesized term. $(n\ \Psi_1\ \Psi_2 \ldots \Psi_k)$ represents the successive repetition *n* times of the firing sequence $\Psi_1\ \Psi_2 \ldots \Psi_k$.

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

PASS — A periodic admissable sequential schedule.

p(α) — The number of samples produced onto SDF arc α by one invocation of *source*(α).

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.

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.

$\mathbf{q}_G$ — The *repetitions vector* $\mathbf{q}_G$ of the SDF graph G is a vector that is indexed by the nodes in G. $\mathbf{q}_G$ has the property that every PASS for G invokes each node N a multiple of $\mathbf{q}_G(N)$ times.

single appearance schedule — A schedule that contains only one appearance of each actor in the associated SDF graph.

Appendix

This appendix presents an efficient algorithm for computing the repetitions vector $\mathbf{q}_G$ for an SDF graph. The time complexity of this algorithm is linear in the number of arcs in the input SDF graph. Our specification of the algorithm will use the following notation.

Notation: For a rational number χ, we denote the numerator and denominator of χ as $numer(\chi)$ and $denom(\chi)$ respectively, and we denote by $reduced\_form(\chi)$ that rational number whose numerator is $numer(\chi) / gcd(numer(\chi), denom(\chi))$ and whose denominator is $denom(\chi) / gcd(numer(\chi), denom(\chi))$. Finally, we denote the least common multiple of a set of positive integers z_1, z_2, …, z_k by $LCM(z_1, z_2, \ldots, z_k)$. For example $numer(6/9) = 6$, $denom(6/9) = 9$, $reduced\_form(6/9) = 2/3$, and $LCM(6, 9) = 18$.

Algorithm to compute the repetitions vector in linear time:

Input: a connected SDF graph G.
Output: the repetitions vector $\mathbf{q}_G$ for G.

```
     Define an array of rational numbers Q with one entry correspond-
ing to each node in G. For each N ∈ N(G), initialize Q(N) to be
zero.
     for each arc α in G
          if (Q(source(α)) ≠ 0) and (Q(sink(α)) ≠ 0))
          then
               ** check for sample-rate consistency **
               if Q(source(α))×p(α) ≠ Q(sink(α))×c(α)
               then
                    ERROR: G has inconsistent sample rates.
                    exit
               end-if
          else if Q(source(α)) = 0
          then
               Set Q(source(α)) = reduced_form(Q(sink(α))×c(α)/p(α)).
               Set multiplier = LCM(multiplier, denom(Q(source(α)))).
          else if Q(sink(α)) = 0
          then
               Set Q(sink(α)) = reduced_form(Q(source(α))×p(α)/c(α)).
               Set multiplier = LCM(multiplier, denom(Q(sink(α)))).
          end-if
```

existence of a single appearance schedule. When used as a preprocessing technique, this can sharply reduce the execution time of a loose interdependence algorithm.

7 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 structures for a large class of applications. Furthermore, we have extended the developments of [4] 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 [21]. 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 [17]. 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. Another important trade-off to further examine is that involving buffering costs.

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 ∈ (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. *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 α is directed from A to B then condition (3) is equivalent to $p(\alpha) = kc(\alpha)$. 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 the

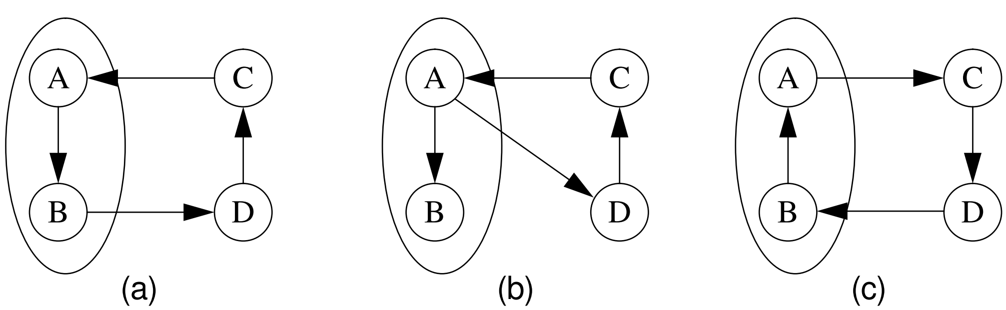

Fig. 9. An illustration of how a directed circuit involving Ω originates in G' for theorem 4. The two possible scenarios are shown in (a) and (b); (c) will not occur due to condition (4). SDF parameters on the arcs have not been assigned because they are irrelevant to the introduction of directed cycles.

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 EQ 1 it follows that $Z \mid_G (X_1 - Z + \{B\})$. 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 $Y_1 \mid_{subgraph(X_1)} Y_2$. From EQ 2 and lemma 6, Y_1 is subindependent of Y_2 in G. Now if $A \in Y_1$, then from condition (3), B is subindependent of Y_2 in G, so from fact 6(a), $(Y_1 \cup \{B\}) \mid_G Y_2$. Applying lemma 4, we see that G' is loosely interdependent. On the other hand, suppose that $A \in Y_2$. From EQ 1, we know that Y_1 is subindependent of $\{B\}$ in G. From fact 6(b), it follows that Y_1 is subindependent of $(Y_2 \cup \{B\})$, so again we can apply lemma 4 to conclude that G' is loosely interdependent. *QED*.

Theorem 4: Suppose G is a connected SDF graph, 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) $\mathbf{q}_G(B) = k\mathbf{q}_G(A)$ for some positive integer k.

(4) B has no predecessors other than A and B.

Proof. It suffices to show that all strongly connected subgraphs in G' that contain Ω are loosely interdependent. So we suppose that Z' is a strongly connected subset of $N(G')$ that contains Ω, and we let Z denote the "corresponding" subset in G; that is, $Z = Z' - \{\Omega\} + \{A, B\}$. Now in Z', suppose that there is a directed circuit $(C \rightarrow \Omega \rightarrow D \rightarrow C)$ containing the node Ω. 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 Ω in G' are illustrated in figure 9(a) and (b); the situation in (c) cannot arise because of condition (4).

Now in Z', if one or more of the circuits involving Ω corresponds to figure 9(a), then Z must be strongly connected. Otherwise, all of the circuits involving Ω correspond to figure 9(b),

Proof. For each arc α directed from a member of Z_2 to a member of Z_1, we have $delay(\alpha) \geq total\_consumed(\alpha, subgraph(Z))$. From fact 3, $\mathbf{q}_{subgraph(Z)}(N) = \mathbf{q}_G(N)$ for all $N \in Z$. Thus, for all arcs α in $subgraph(Z)$, $total\_consumed(\alpha, subgraph(Z)) = total\_consumed(\alpha, G)$, and we conclude that Z_1 is subindependent of Z_2 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) $\mathbf{q}_G(B) = k\mathbf{q}_G(C)$ for some $C \in N(G)$, $C \neq B$.

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 X_1, X_2 of $N(G)$ such that $X_1 |_G X_2$. If $A, B \in X_1$ or $A, B \in X_2$, then from lemma 4, we are done. Now condition (2) precludes the scenario ($B \in X_1$, $A \in X_2$), so the only remaining possibility is ($A \in X_1$, $B \in X_2$). There are two cases to consider here:

(i) B is not the only member of X_2. Then from (3), $(X_1 + \{B\}) |_G (X_2 - \{B\})$. But $A, B \in (X_1 + \{B\})$, so lemma 4 again guarantees that G' is loosely interdependent.

(ii) A is not the only member of X_1 and $X_2 = \{B\}$. Thus we have $X_1 |_G \{B\}$, so

$$\forall \alpha \in A(G), (source(\alpha) = B) \Rightarrow delay(\alpha) \geq total\_consumed(\alpha, G). \qquad \textbf{(EQ 1)}$$

Also, since $C \in X_1$ we have from (4) that $q_G(X_1) = gcd(\{\mathbf{q}_G(N) \mid N \in X_1\}) = gcd(\{\mathbf{q}_G(N) \mid N \in X_1\} \cup \{k\mathbf{q}_G(C)\}) = gcd(\{\mathbf{q}_G(N) \mid N \in X_1\} \cup \{\mathbf{q}_G(B)\}) = gcd(\{\mathbf{q}_G(N) \mid N \in N(G)\}) = 1$. That is,

$$q_G(X_1) = 1. \qquad \textbf{(EQ 2)}$$

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), $A \notin Z$. This is true because if Z contained A, then no member of $(X_1 - Z)$ would have a directed path

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

Fact 8: If G_1 and G_2 are two isomorphic SDF graphs and G_1 is loosely interdependent then G_2 is loosely interdependent.

Lemma 5: Suppose that G is an SDF graph, $M \subseteq N(G)$, $A_1 \in M$, and A_2 is an SDF node that is contained in $N(G)$ but not in M such that

(1) A_2 is not adjacent to any member of $(M - \{A_1\})$, and

(2) for some positive integer k, $\mathbf{q}(A_2) = k\mathbf{q}(A_1)$.

Then if we cluster $W = \{A_1, A_2\}$ in G, then *subgraph*$(M - \{A_1\} + \{\Omega\}, G')$ is isomorphic to *subgraph*(M, G).

As a simple illustration, consider again the clustering example of figure 7(c)-(d). Let G and G' respectively denote the graphs of figures 7(c) and (d), and in figure 7(c), let $M = \{A, B\}$, $A_1 = A$, and $A_2 = C$. Then $(M - \{A_1\} + \Omega) = \{B, \Omega\}$, and clearly, *subgraph*$(\{B, \Omega\}, G')$ is isomorphic to *subgraph*$(\{A, B\}, G)$.

Proof of lemma 5. Let $C =$ *subgraph*$(M - \{A_1\} + \Omega, G')$, let Φ denote the set of arcs in *subgraph*(M, G), and let Φ' denote the set of arcs in C. From (1), every arc in C has a corresponding arc in *subgraph*(M, G) and vice-versa, and thus $\Phi' = \{\alpha' \mid \alpha \in \Phi\}$. Now from the definition of clustering a subgraph, we know that $p(\alpha') = p(\alpha)$ for any arc $\alpha \in \Phi$ such that $source(\alpha) \neq A_1$. If $source(\alpha) = A_1$, then α is replaced by α' with $source(\alpha') = \Omega$, and $p(\alpha') = p(\alpha)\mathbf{q}(A_1) / gcd(\mathbf{q}(A_1), \mathbf{q}(A_2))$. But $gcd(\mathbf{q}(A_1), \mathbf{q}(A_2)) = gcd(\mathbf{q}(A_1), k\mathbf{q}(A_1)) = \mathbf{q}(A_1)$, 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

$$f_1(N) = N \text{ if } N \neq A_1,\ f_1(A_1) = \Omega;\ \text{and } f_2(\alpha) = \alpha'$$

demonstrates that *subgraph*(M, G) is isomorphic to C. *QED.*

Lemma 6: Suppose that G is a strongly connected SDF graph, and Z is a strongly connected subset of nodes in G such that $q_G(Z) = 1$. Suppose Z_1 and Z_2 are disjoint subsets of Z such that Z_1 is subindependent of Z_2 in *subgraph*(Z). Then Z_1 is subindependent of Z_2 in G.

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 $X \subseteq N(G)$, 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 figure 7(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 figure 7(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 $X' = \{\Omega, C\}$.

Lemma 4: Suppose that G is a strongly connected SDF graph and X_1, X_2 partition $N(G)$ such that $X_1 \,|_G\, X_2$. Also suppose that A, B are nodes in G such that $A, B \in X_1$ or $A, B \in X_2$. If we cluster $W = \{A, B\}$ then the resulting SDF graph G' is loosely interdependent.[1]

Proof. Let Φ denote the set of arcs directed from a node in X_2 to a node in X_1, and let Φ' denote the set of arcs directed from a node in X_2' to a node in X_1'. Since subgraph $\{A, B\}$ does not contain any arcs in Φ, it follows that $\Phi' = \{\alpha' \mid \alpha \in \Phi\}$. From fact 5, it can easily be verified that for all α', *total_consumed*(α', G') = *total_consumed*(α, G). Now since $X_1 \,|_G\, X_2$, we have $\forall\, \alpha \in \Phi$, *delay*$(\alpha) \geq$ *total_consumed*(α, G). It follows that $\forall\, \alpha' \in \Phi'$, *delay*$(\alpha') \geq$ *total_consumed*(α', G'). We conclude that X_1' is subindependent of X_2' in G'. *QED.*

Definition 4: We say that two SDF graphs G_1 and G_2 are *isomorphic* if there exist bijective mappings f_1: $N(G_1) \rightarrow N(G_2)$ and f_2: $A(G_1) \rightarrow A(G_2)$ such that for each $\alpha \in A(G_1)$, *source*$(f_2(\alpha)) = f_1($*source*$(\alpha))$, *sink*$(f_2(\alpha)) = f_1($*sink*$(\alpha))$, *delay*$(f_2(\alpha))$ = *delay*(α), $p(f_2(\alpha)) = p(\alpha)$, and $c(f_2(\alpha)) = c(\alpha)$. Intuitively, two SDF graphs are isomorphic if they differ only by a relabeling of the nodes. For example, the SDF graph in figure 7(d) is isomorphic to *subgraph*$(\{A, B\})$ in figure 7(c).

1. However, G' may be deadlocked even if G is not. This will not be a problem in our application of lemma 4.

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 figure 8. One possible single appearance schedule for figure 8(a) is (10 X)(10 Y)ZV(10W). This is the *minimum activation* schedule preferred by Ritz et al. [21]; 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 figure 8(d); this graph leads to the single appearance schedule (10 Ω_2)Ω_1(10 W) $\Rightarrow$ (10 XY)ZV(10 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.

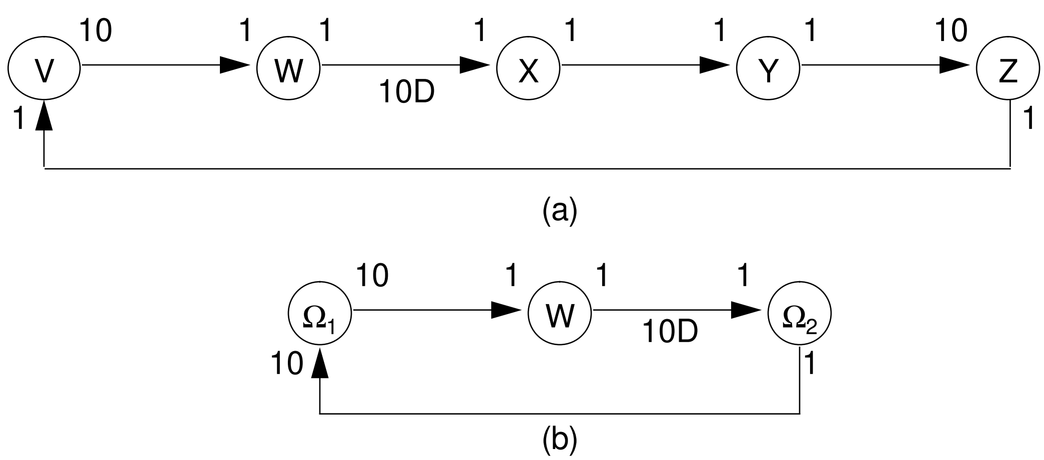

Fig. 8. An example of clustering to increase the frequency of data transfers that occur through registers rather than memory.

Figure 7 illustrates two ways in which arbitrary clustering decisions can conflict with code compactness objectives. Observe that figure 7(a) is an acyclic graph so it must have a single appearance schedule. Figure 7(b) is the hierarchical SDF graph that results from clustering A and B in figure 7(a). It is easy to verify that this is a tightly interdependent graph. In fact, the only minimal periodic schedule for figure 7(a) that we can derive from this clustering is $C\Omega C \Rightarrow CABC$. Thus the clustering of A and B in figure 7(a) cancels the existence of a single appearance schedule.

In figure 7(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 figure 7(c) in such a way that C appears only once. Now observe that the graph that results from clustering A and C, shown in figure 7(d), is tightly interdependent. It can be verified that the most compact minimal periodic schedule for this graph is $(5\ \Omega)B(5\ \Omega)$, which leads to the schedule (5 AC)B(5 AC) for figure 7(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*

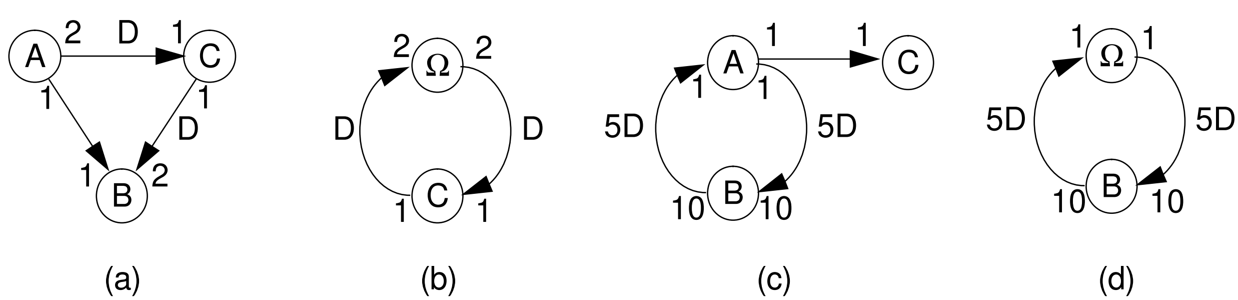

Fig. 7. Examples of how clustering can conflict with the goal of code compactness.

member of Z_2 to a member of Z_1, we have $delay(\alpha) \geq c(\alpha) \times \mathbf{q}_G(sink(\alpha))$. It follows that $Z_1 \,|_G\, Z_2$, so G is loosely interdependent.

$\Leftarrow$ Suppose that G is loosely interdependent. Then N(G) can be partitioned into Z_1 and Z_2 such that $Z_1 \,|_G\, Z_2$. 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. *QED*.

Thus, λ_{sp} can be constructed as follows: (1) Determine $\mathbf{q}_G(N)$ for each node N; (2) Remove each arc α whose delay is at least $c(\alpha) \times \mathbf{q}_G(sink(\alpha))$; (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) cluster 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. The appendix presents an algorithm that performs (1) in time $O(m)$; 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 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, where m is *max*(number of nodes, number of arcs). 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, where n is the number of nodes in the input graph. For most applications, the running time of the algorithm will scale linearly with the size of the input graph.

6 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 4.

Theorem 3: Suppose that G is a nontrivial strongly connected SDF graph. From G, remove all arcs α for which $delay(\alpha) \geq c(\alpha) \times \mathbf{q}_G(sink(\alpha))$, 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 figure 6(a). The repetitions vector for G is $\mathbf{q}_G(A, B, C, D) = (1, 2, 2, 4)$. This graph is loosely interdependent if $d_1 \geq 2$, which corresponds to $\{C, D\} |_G \{A, B\}$, or if $d_2 \geq 4$, which corresponds to $\{A, B\} |_G \{C, D\}$. The corresponding G''s are depicted at the bottom of figure 6: Figure 6(b) shows G' when $d_1 \geq 2$ and $d_2 < 4$, and figure 6(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.

$\Rightarrow$ 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 partition N(G). Also, none of the arcs directed from Z_2 to Z_1 in G occur in G'. Thus, by the construction of G', for each arc α in G directed from a

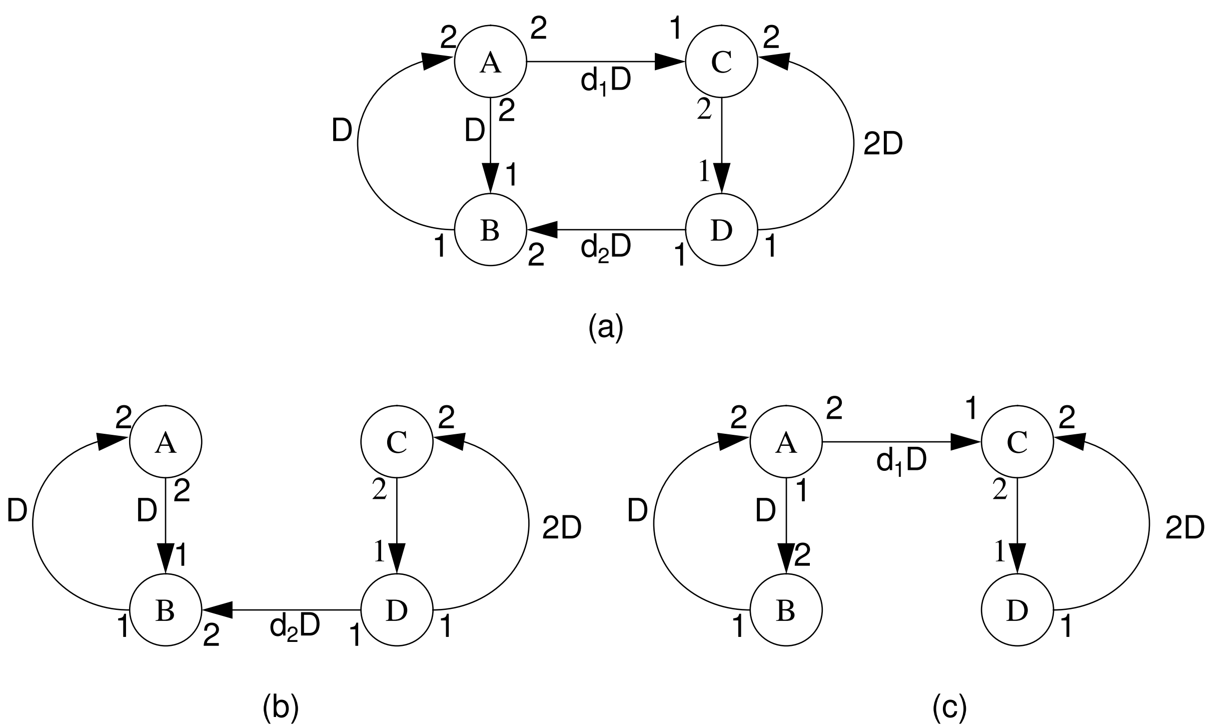

Fig. 6. An illustration of theorem 3.

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 2, we see that λ_{ts} is applied exactly once for each tightly interdependent component. For example, the technique of [4] can be applied as the tight scheduling algorithm. This technique involves a hierarchical clustering phase that has time complexity[1] O(number of arcs $\times$ 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 the original program has n nodes, n decomposition steps are required in the worst case.Tarjan [23] 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 a 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}.

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.

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, λ partitions *subgraph*(C) into X and Y such that X |*subgraph*(C) Y. We set M_1 to be that connected component of *subgraph*(X) or *subgraph*(Y) that contains N. Since X, Y partition C, M_1 is a proper subset of M_o. Also, from remark 3, *appearances*(N, S_λ(*subgraph*(M_0))) = *appearances*(N, S_λ(*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, *appearances*(N, S_λ(*subgraph*(M_0))) = *appearances*(N, S_λ(*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 *appearances*(N, S_λ(*subgraph*(M_1))) = *appearances*(N, S_λ(*subgraph*(M_2))), and $N(T) \subseteq M_2$. Continuing this process, we get a sequence M_1, M_2, …. Since each M_i 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, *appearances*(N, S_λ(G)) = *appearances*(N, S_λ(*subgraph*(M_0))) = *appearances*(N, S_λ(*subgraph*(M_1))) = … = *appearances*(N, S_λ(*subgraph*(M_k))); and thus *appearances*(N, S_λ(G)) = *appearances*(N, S_λ(*subgraph*(T))). *QED.*

Theorem 2 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 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 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*.

Lemma 3: Suppose that G is a 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 H_1 |*subgraph*(H) 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 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 2: Suppose G is an SDF graph that has a PASS, N is a node in G, and λ 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_λ(G)) = *appearances*(N, S_λ(*subgraph*(T))) — the number of appearances of N is determined entirely by the tight scheduling algorithm of λ.

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_λ(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 $T \neq M_o$; by definition, tightly interdependent graphs are strongly connected, so T is contained in some strongly connected component C of *subgraph*(M_o).

⇒ 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 $X_0 \,|subgraph(Z_0)\, Y_0$, and *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 $X_k \,|subgraph(Z_k)\, Y_k$; $C \subseteq Z_k$; and ($X_k \cap$ C) and ($Y_k \cap$ C) are both nonempty. From lemma 2, ($X_k \cap$ C) is subindependent in *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 |G Z_2 and Z_1' | G Z_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.

H_2 of H_1', then λ will partition H_2, and we will obtain a proper subset H_2' of H_1' such that *appearances*(N, S_λ(*subgraph*(H_1'))) = *appearances*(N, S_λ(*subgraph*(H_2'))). Continuing in this manner, we get a sequence H_1', H_2', … of subsets of *N*(G) such that each H_i' is a proper subset of H_{i-1}', N is contained in each H_i', and *appearances*(N, S_λ(G)) = *appearances*(N, S_λ(*subgraph*(H_1'))) = *appearances*(N, S_λ(*subgraph*(H_2'))) = …. Since each 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 *subgraph*(H_m'), and *appearances*(N, S_λ(G)) = *appearances*(N, S_λ(*subgraph*(H_m'))). But from remark 1, S_λ(*subgraph*(H_m')) contains only one appearance of N. *QED*.

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 α 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, $delay(\alpha) \geq c(\alpha) \times \mathbf{q}_G(sink(\alpha))$, and by fact 3, $\mathbf{q}_G(sink(\alpha)) \geq \mathbf{q}_{subgraph(C)}(sink(\alpha))$. Thus, $delay(\alpha) \geq c(\alpha) \times \mathbf{q}_{subgraph(C)}(sink(\alpha))$. Since this holds for any α 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 $Z_1 |_G Z_2$, 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. $\Leftarrow$ Suppose every nontrivial strongly connected subgraph of G is loosely interdependent, and let λ be any loose interdependence algorithm. Since no node in G is contained in a tightly interdependent subgraph, it follows from lemma 1 that S_λ(G) is a single appearance schedule for G.

strongly connected component besides Z_k, and since S'(G) contains only one appearance of Ω_k, we have *appearances*(N, S_L(G)) = *appearances*(N, S_x S_y). Thus, for i = 1, 2,…, v, N $\in$ X_i $\Rightarrow$ *appearances*(N, S_L(G)) = *appearances*(N, S_L(*subgraph*(X_i))). By the same argument, we can show that for i = 1, 2,…, w, N $\in$ Y_i $\Rightarrow$ *appearances*(N, S_L(G)) = *appearances*(N, S_L(*subgraph*(Y_i))).

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 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 λ is a loose interdependence algorithm. Then N appears only once in S_λ(G), the schedule generated by λ.

Proof. 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, *subgraph*(H_1) must be loosely interdependent, so λ partitions H_1 into X and Y, where X $|_{subgraph(H_1)}$ Y. Let H_1' denote that connected component of *subgraph*(X) or *subgraph*(Y) that contains N. From remark 3, *appearances*(N, S_λ(G)) = *appearances*(N, S_λ(*subgraph*(H_1'))).

From our assumptions, all nontrivial strongly connected subgraphs of H_1' that contain N are loosely interdependent. Thus, if N is contained in a nontrivial strongly connected component

```
for i = 1, 2, ..., s
    Apply A_1 to subgraph(Z_i);
    if X, Y ⊆ Z_i are found such that X |subgraph(Z_i) Y,
    then
        • Determine the connected components X_1, X_2, ..., X_v of
        subgraph(X), and the connected components
        Y_1, Y_2, ..., Y_w of subgraph(Y).
        • Recursively apply algorithm L to construct the
        schedules
        S_x = (q_G(X_1) S_L(subgraph(X_1))...(q_G(X_v)  S_L(subgraph(X_v)),
        S_y = (q_G(Y_1) S_L(subgraph(Y_1))...(q_G(Y_w)  S_L(subgraph(Y_w)).
        • Replace the (single) appearance of Ω_i in S'(G)
        with S_x S_y.
    else (subgraph(Z_i) is tightly interdependent)
        •Apply A_4 to obtain a valid schedule S_i for
        subgraph(Z_i).
        •Replace the single appearance of Ω_i in S with
        (q_G(Z_i)  S_i).
    end-if
end-for
```

The **for**-loop replaces each "Ω_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 for G. We output S_L.■

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 not 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 Ω_i, we have *appearances*(N, S_L(G)) = *appearances*(N, S_L(*subgraph*(C))).

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 |*subgraph*(Z_k) Y, and replaces the single appearance of Ω_k in S'(G) with S_x S_y. If N is a member of the connected component X_i, then N $\notin$ Y, so *appearances*(N, S_x S_y) = *appearances*(N, S_L(*subgraph*(X_i))). Also since N cannot be in any other

4 The Class of Loose Interdependence Algorithms

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 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, ..., Z_s of G.
Step 2: Cluster Z_1, Z_2, ..., Z_s into nodes Ω_1, Ω_2, ..., Ω_s 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:

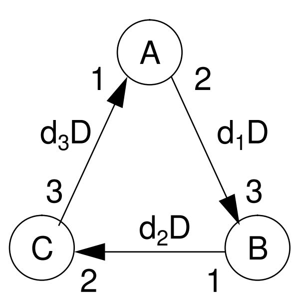

Fig. 5. An illustration of loose and tight interdependence. Here d_1, d_2, and d_3 represent the number of delays on the associated arcs. This SDF graph is tightly interdependent if and only if ($d_1 < 6$), ($d_2 < 2$), *and* ($d_3 < 3$).

A slightly different form of this condition was developed independently by Ritz et al. in [21], which discusses single appearance schedules in the context of *minimum activation schedules*. For example, the schedule A(2CB) for figure 1 results in 5 activations since invocations of C and B are interleaved. In contrast, the schedule A(2B)(2C) requires only one activation per actor, for a total of 3 activations. In the objectives of [21], the latter schedule is preferable because in that code generation framework there is a large overhead associated with each activation. However such overhead can often be avoided with careful instruction scheduling and register allocation, as [18] demonstrates. We prefer the former schedule, which has less looping overhead and requires less memory for buffering.

Fact 7 implies that for an SDF graph to have a single appearance schedule, we must be able to decompose each nontrivial strongly connected component into two subsets in such a way that one subset is subindependent of the other. Another implication of fact 7 is that every acyclic SDF graph has a single appearance schedule. We can easily construct a single appearance schedule for an acyclic SDF graph. We simply pick a root node N_1; schedule all of its invocations in succession; remove N_1 from the graph and pick a root node N_2 of the remaining graph; schedule all of N_2's invocations in succession; and so on until we have scheduled all of the nodes. By this procedure, we get a cascade of loops $(\mathbf{q}_G(N_1)\ N_1)\ (\mathbf{q}_G(N_2)\ N_2)\ \ldots\ (\mathbf{q}_G(N_k)\ N_k)$, which gives us a single appearance schedule.

Definition 2: Suppose that G is a nontrivial strongly connected SDF graph. Then we say that G is **loosely interdependent** if $N(G)$ can be partitioned into Z_1 and Z_2 such that $Z_1 |_G Z_2$. We say that G is **tightly interdependent** if it is not loosely interdependent.

For example, consider the strongly connected SDF graph in figure 5. The repetitions vector for this graph is $\mathbf{q}_G(A, B, C) = (3, 2, 1)$. Thus the graph is loosely interdependent if and only if $(d_1 \geq 6)$ or $(d_2 \geq 2)$ or $(d_3 \geq 3)$.

In this section we have introduced topological properties of SDF graphs that are related to the existence of single appearance schedules. In the following section we use these properties to develop our scheduling framework and to demonstrate some of its useful qualities.

form the class of schedules that allow in line code generation without any code space or subroutine penalty.

3 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 subsets of *N*(G) we say that "Z_1 is **subindependent** of Z_2 in G" if for every arc α in G such that *source*(α) $\in Z_2$ and *sink*(α) $\in Z_1$, we have *delay*(α) $\geq$ *total_consumed*(α, 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 ($Z_1 |_G Z_2$), 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 $Z_1 |_G Z_2$ if Z_1 is subindependent of Z_2, and Z_1 and Z_2 form a partition of the nodes in G. For example, consider figure 3(a). Here $\mathbf{q}_G$(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: Suppose that 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) and (Y is subindependent of Z) $\Rightarrow$ (X $\cup$ Y) is subindependent of Z.

(b) (X is subindependent of Y) and (X 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 [3].

Fact 7: An SDF graph has a valid single appearance schedule iff for each nontrivial strongly connected component Z, there exists a partition X, Y of Z such that X $|_{subgraph(Z)}$ Y, and *subgraph*(X) and *subgraph*(Y) each have single appearance schedules.

Let S^* denote the schedule that results from replacing each appearance of Ω 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 *subgraph*(Z) into the node Ω. Then $\mathbf{q}_{G'}(\Omega) = q_G(Z)$; and for any node N in G' other that Ω, $\mathbf{q}_{G'}(N) = \mathbf{q}_G(N)$.

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 directed path from A to B and a directed 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 *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 figure 3(a).

Similarly, we define a *connected component* of a directed graph to be a maximal subset of nodes Z such that if A and B are distinct members of Z, then there is a directed path from A to B, or there is a directed path from B to A, or both. For example in figure 4, the connected compo-

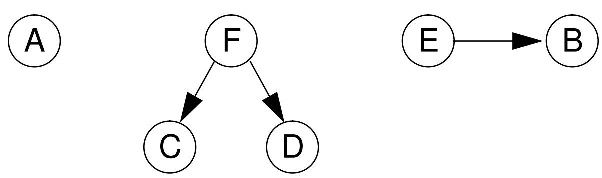

Fig. 4. A directed graph that has three connected components.

nents are {A}, {C, D, F}, and {B, E}.

Given a connected SDF graph G, and an arc α in G, we define *total_consumed*(α, G) to be the total number of samples consumed from α in a minimal schedule period for G. Thus *total_-consumed*(α, G) $= \mathbf{q}_G(sink(\alpha))c(\alpha)$. Finally, given an SDF graph G, a looped schedule S for G and a node N in G, we define *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 $N \in N(G)$, *appearances*(N, S) = 1. For example, consider the two schedules S_1 = CA(2B)C and S_2 = A(2B)(2C) for figure 1. We have *appearances*(C, S_1) = 2; *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

the component of $\mathbf{q}_G$ corresponding to a node N by $\mathbf{q}_G(N)$. Every PASS for G invokes each node N a multiple of $\mathbf{q}_G(N)$ 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 $J\mathbf{q}_G(N)$ times. We call $\mathbf{q}_G$ the *repetitions vector* of G. For example in figure 3, $\mathbf{q}_G(A) = 10$, $\mathbf{q}_G(B) = 4$, and $\mathbf{q}_G(C) = 5$. An efficient algorithm to compute $\mathbf{q}_G$ is presented in the appendix. The following properties of repetitions vectors are established in [13]:

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

Fact 2: The *balance equation* $\mathbf{q}_G(source(\alpha)) \times p(\alpha) = \mathbf{q}_G(sink(\alpha)) \times c(\alpha)$ is satisfied for each arc α in G.

Given a subset Z of nodes in a connected SDF graph G, we define $q_G(Z) = gcd(\{\mathbf{q}_G(N) \mid N \in Z\})$, 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 [3].

Fact 3: If G is a connected SDF graph, and Z is a connected subset of $N(G)$, then for each $N \in Z$, $\mathbf{q}_G(N) = q_G(Z)\mathbf{q}_{subgraph(Z)}(N)$.

For our hierarchical scheduling approach, we will apply the concept of *clustering* a subgraph. This process is illustrated in figure 3. Here *subgraph*({A, C}) of (a) is clustered into the hierarchical node Ω_{AC}, and the resulting SDF graph is shown in (b). Similarly, clustering *subgraph*({A, B}) results in the graph of (c). Each input arc α to a clustered subgraph P is replaced by an arc α' having $p(\alpha') = p(\alpha)$, and $c(\alpha') = c(\alpha) \times \mathbf{q}_G(sink(\alpha))/q_G(N(P))$, the number of samples consumed from α in one *invocation of subgraph* P. Similarly we replace each output arc β with β' such that $c(\beta') = c(\beta)$, and $p(\beta') = p(\beta) \times \mathbf{q}_G(source(\alpha))/q_G(N(P))$. The following properties of clustered subgraphs are proven in [3]

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 *subgraph*(Z) into the hierarchical node Ω, and S' is a PASS for G'. Suppose that S_Z is a PASS for *subgraph*(Z) such that for each $N \in Z$, S_Z invokes N $(\mathbf{q}_G(N)/q_G(Z))$ times.

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 α, we let *source*(α) and *sink*(α) denote the nodes at the source and the sink of α; we let $p(\alpha)$ denote the number of samples produced by *source*(α), $c(\alpha)$ denote the number of samples consumed by *sink*(α), and we denote the delay on α by *delay*(α).We define a **subgraph** of G to be that SDF graph formed by any $Z \subseteq N(G)$ together with the set of arcs $\{\alpha \in A(G) \mid source(\alpha), sink(\alpha) \in Z\}$. We denote the subgraph associated with the subset of nodes Z by *subgraph*(Z, G); if G is understood, we may simply write *subgraph*(Z). Finally, 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.

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 α, and appending $p(\beta)$ 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 *admissable sequential schedule* ("sequential" is used to distinguish this type of schedule from a parallel schedule) for G is a finite sequence $S = S_1 S_2 \ldots S_N$ 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.

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 α, (the number of times *source*(α) is fired in S) $\times\, p(\alpha)$ = (the number of times *sink*(α) is fired in S) $\times\, c(\alpha)$. A *periodic admissable sequential schedule* (PASS) is a schedule that is both periodic and admissable. 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 [13], 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 firing rates by a vector of positive integers $\mathbf{q}_G$, which is indexed by the nodes in G, and we denote

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. We emphasize that the techniques developed in this paper extend the developments of [4] 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.

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 codes space compactness for a large class of SDF graphs.

Because we focus on the fundamental limits of program compactness via loops, 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.

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 such periodic schedules. Most of the terminology introduced in this and subsequent sections is summarized in the glossary at the end of the paper.

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 250000 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 $(250000)^2 = 6.25\times10^{10}$ 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 [4] 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, clustering subgraphs based on repetition count alone does not fully carry out this goal.

In this paper, we develop a class of 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

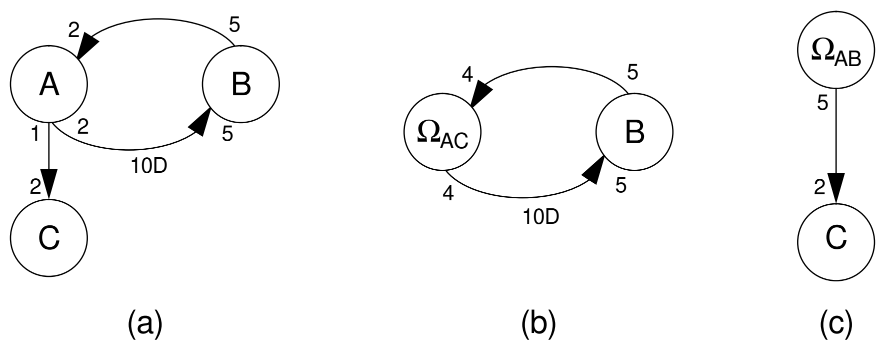

Fig. 3. This example illustrates how clustering subgraphs based on repetition count alone can conceal looping opportunities that occur within cyclic paths. Part (a) depicts a multirate SDF graph. Two pairwise clusterings lead to graphs that have schedules — {A, B}, having repetition count 2, and {A, C}, having repetition count 5 (the clustering of B and C results in deadlock). Clustering the subgraph with the highest repetition count yields the hierarchical topology in (b), for which the most compact schedule is $(2B)(2\Omega_{AC})B\Omega_{AC}B(2\Omega_{AC}) \Rightarrow (2B)(2(2A)C)B(2A)C\text{-}B(2(2A)C)$. Clustering the subgraph {A,B} of lower repetition count, as depicted in part (c), yields the more compact schedule $(2\Omega_{AB})(5C) \Rightarrow (2(2B)(5A))(5C)$.

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. For example, combining C and D in figure 2 results in a graph for which no periodic schedule 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], but this approach is extremely time consuming if a large number of clustering candidates must be considered. In [4], 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 [4]. First, the storage cost of the reachability matrix proved prohibitive for multirate applications involving very large sample rate changes.

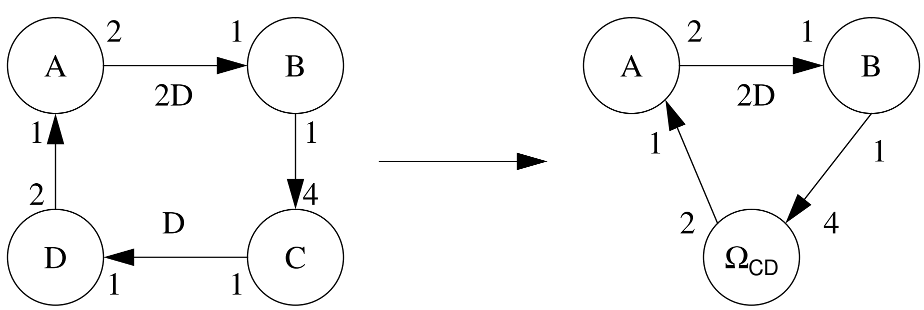

Fig. 2. An example of how clustering a subgraph in an SDF graph can result in deadlock.

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 to convert recursive stream functions in SISAL-2 into SDF graphs.

In [11], How evaluated a scheme in which existing schedulers that did not consider looping were augmented with a post-processing phase that detected successively occurring repetitive firing patterns, and concluded that such simple tactics were ineffective for generating the most compact programs. To synthesize lean looping structures, 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 clustering 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 {B,C} into a hierarchical "supernode" Ω, a scheduler will generate the schedule AΩΩ. To highlight the repetition in a schedule, we let the notation (n $X_1X_2 \ldots X_m$) designate n successive repetitions of the firing sequence $X_1X_2 \ldots X_m$. We refer to a schedule expressed with this notation as a **looped schedule**. Using this notation, and substituting each occurrence of Ω 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 figure 1.

We explored the looping problem further in [4]. 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"

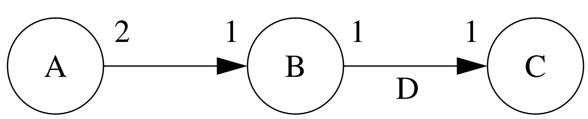

Fig. 1. A simple SDF graph.

1 Introduction

In the dataflow model of computation, pioneered by Dennis [5], 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) [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 studied extensively in the context of synthesizing assembly code for signal processing applications, for example [7, 8, 9, 10, 16, 18, 19, 20].

Figure 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) [12]. 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, 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 [11]. 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[24], SISAL[15], and Silage[9] 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

Memorandum No. UCB/ERL M93/36, Electronics Research Laboratory
College of Engineering, University of California
Berkeley, California 94720

GENERATING COMPACT CODE FROM DATAFLOW SPECIFICATIONS OF MULTIRATE DSP ALGORITHMS [1]

Shuvra S. Bhattacharyya
Joseph T. Buck
Soonhoi Ha
Edward A. Lee

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

May 19, 1993

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 manner that does not conflict with the goal of code compactness.

[1] This research was supported by DARPA, AT&T Bell Laboratories, Semiconductor Research Corporation, and the Office of Naval Research via the Naval Research Laboratory.