[4] 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.

[5] L. J. Hendren, G. R. Gao, E. R. Altman, C. Mukherjee, “A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs”, *Lecture Notes in Computer Science*, February 1992.

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

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

[8] S. How, “Code Generation for Multirate DSP Systems in Gabriel”, Masters Degree Report, Dept. of EECS, U. C. Berkeley, May 1988.

[9] E. A. Lee, “Recurrences, Iteration, and Conditionals in Statically Scheduled Block-Diagram Languages”, *VLSI Signal Processing III*, IEEE Press 1988.

[10] E. A. Lee, D. G. Messerschmitt, “Synchronous Dataflow”, *Proceedings of the IEEE*, September 1987.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

menting other scheduling objectives, such as the minimization of buffering requirements, in a manner that is guaranteed not to interfere with code compaction goals. We have defined a class of SDF graphs called *tightly interdependent* graphs. Schedules for arbitrary SDF graphs can be constructed such that each block that is not contained in a tightly interdependent subgraph appears only once, and thus requires only one instance of its code block to appear in the target program. Our framework defines a broad class of scheduling algorithms that construct such schedules.

Our observations suggest that the vast majority of practical SDF graphs do not contain any tightly interdependent subgraphs, and thus that our scheduling framework guarantees optimal program compactness for most cases. However, we are also investigating how to schedule general tightly interdependent subgraphs compactly. New techniques that are developed for tightly interdependent graphs can easily be incorporated within our scheduling framework, since the framework modularizes the scheduling of tightly interdependent subgraphs: the algorithm used to schedule tightly interdependent subgraphs, called the “tight interdependence algorithm” never interacts with other parts of the overall scheduling algorithm, and the tight interdependence algorithm completely determines the amount of program memory required for the tightly interdependent subgraphs.

Acknowledgment

The authors are grateful to Sebastian Ritz of Aachen University, and to Thomas Parks of U. C. Berkeley for their helpful comments.

References

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

[2] S. S. Bhattacharyya, S. Ha, E. A. Lee, “Single Appearance Schedules for Synchronous Dataflow Programs”, Technical Report, Memorandum No. UCB/ERL M93/4, Dept. of EECS, U. C. Berkeley.

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

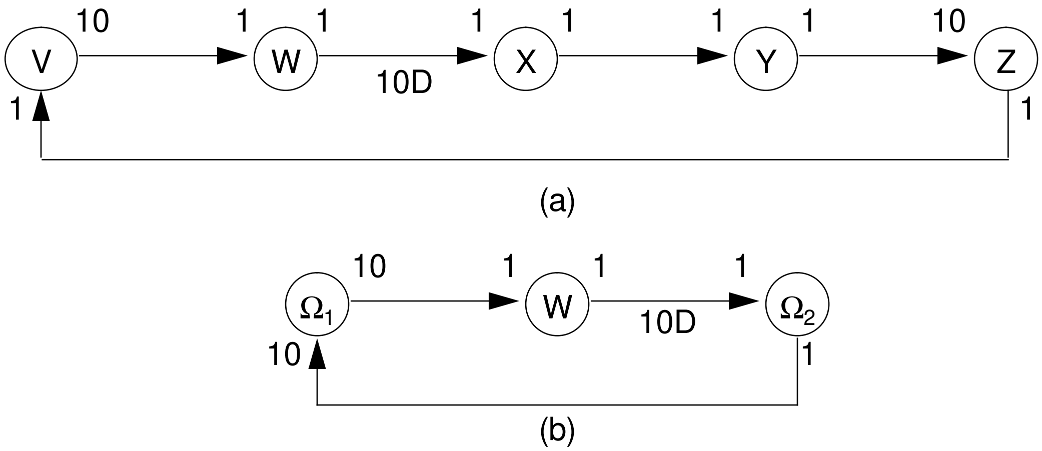


Fig 9. An example of clustering to make data transfers more efficient.

schedule, each sample produced by X is consumed by Y in the same loop iteration, so these data transfers can occur through a single machine register. Thus, the clustering of X and Y saves 10 words of memory for the data transfers between X and Y, and it allows these transfers to be performed through machine registers, which will usually result in faster code.

We have implemented a clustering process based on conditions 1-4. This clustering algorithm can be performed very efficiently since it requires only *local* dataflow information — it uses only the production, consumption and delay parameters of the arcs between the two candidate adjacent nodes. Since many practical dataflow blocks — such as forks, gains, trigonometric functions, and a large class of filtering operations — have only one input, condition (4) is often satisfied. Furthermore, since connected subgraphs of computations operating at the same sample-rate are common [8], our clustering technique can be applied frequently in practice. Finally, from our observations, most practical SDF graphs have single appearance schedules. Since clustering based on conditions 1-4 preserves the existence of a single appearance schedule, it preserves optimal code compactness *for the common case*.

6 Conclusion

We have developed a code scheduling framework for compiling synchronous dataflow graphs into compact target programs through the careful organization of loops, and for impleconnected component may be deadlocked, or it may contain a tightly interdependent subgraph. For example in figure 7, this rule would reject the clustering of X and Y since there is a path between these nodes that passes through Z.

This conservative but efficient rule can be applied outside of acyclic subgraphs with a few more restrictions. If there are one or more arcs directed *from* a node X *to* another node Y, then it can be shown that clustering X and Y does not introduce nor extend a tightly interdependent component if the following conditions hold[1]:

(1) Neither X nor Y is contained a tightly interdependent component.

(2) At least one arc directed from X to Y has zero delay.

(3) X and Y are invoked the same number of times in a periodic schedule.

(4) Y has no predecessors other than X or Y; that is, there is no arc directed *from* a node other than X or Y *to* Y.

In other words, if conditions 1-4 hold, and we cluster X and Y, then the tightly interdependent components of the resulting graph are the same as those of the original graph. An important special case occurs when the original graph has a single appearance schedule. In this case, we can apply any number of adjacent-node clusterings that satisfy 1-4, and the resulting graph will also have a single appearance schedule.

One important practical use of this clustering rule is to increase the number of data transfers that occur in machine registers, rather than through memory. Figure 9 shows a simple example. One possible single appearance schedule for the SDF graph in figure 9(a) is (10 X) (10 Y) Z V (10 W). This schedule, which is the optimal schedule with respect to the *minimum activation* criterion of Ritz. et. al. [16], is inefficient. 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 for the arc connecting X and Y. However, observe that conditions 1-4 hold for the pairs {X, Y} and {Z, V}. Thus we can safely cluster these pairs of nodes without cancelling the existence of a single appearance schedule. This clustering, shown in figure 9(b), leads to the single appearance schedule (10 Ω_2) Ω_1 (10 W) $\Rightarrow$ (10 X Y) Z V (10 W). In this second

---

[1] We emphasize that these conditions are sufficient, but not necessary.

Now if we cluster X and Y, we obtain the hierarchical tightly interdependent SDF graph in figure 8(b). It can easily be verified that the only minimal periodic schedule for this SDF graph is ΩZΩ, which leads to the schedule XYZXY for figure 8(a). Thus, the clustering of X and Y increases the minimum number of appearances of X in the schedule. This can be critical if X has a very large code block because it would make in-line code impractical.

A cluster that introduces a new tightly interdependent component, as in figure 7, always degrades code compactness potential: the optimally compact schedule for the clustered graph will be larger than that of the original graph. However, extending a tightly interdependent component is not always detrimental. As a simple example, it is not detrimental when the cluster is invoked only once for each invocation of the enlarged tightly interdependent component. This is the case if the arc from X is directed to Z instead of Y, as shown in figure 8(c). Here, clustering X and Z results in the schedule YXZY, which contains only one appearance of X and Z.

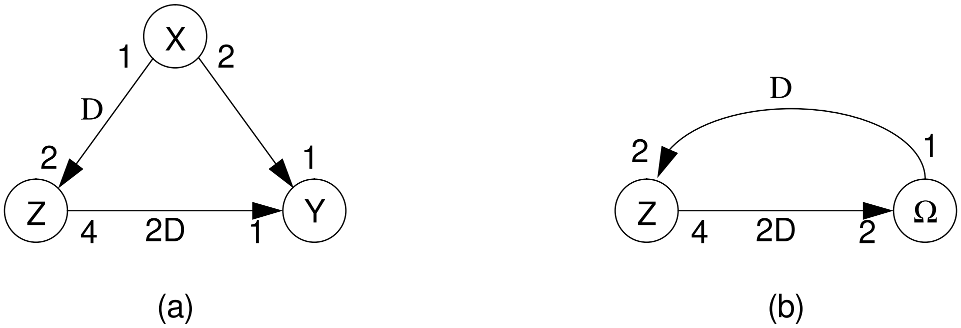
The possible introduction or enlargement of tightly interdependent components adds an additional consideration when incorporating a clustering algorithm into a loose interdependence algorithm. For example, consider the heuristic technique described in [1] for constructing looped schedules with manageable buffering requirements. As described before, this technique repeatedly clusters the pairs of adjacent nodes whose associated subgraphs have the highest invocation count. Only clustering candidates that cause deadlock are rejected.

This technique can be applied to the acyclic scheduling algorithm of a loose interdependence algorithm, but it must be modified to take into consideration whether or not a clustering candidate introduces tight interdependence (as in figure 7). This involves detecting whether or not the cluster introduces a strongly connected component in the originally acyclic graph, and then repeatedly applying subindependence partitioning. If decomposition terminates at a tightly interdependent subgraph, the clustering candidate must be rejected.

This check is computationally expensive. An alternative is to simply disallow a cluster of two adjacent nodes when there is a path from the source node to the sink node that passes through at least one other node. In such cases, the cluster will introduce one or more directed cycles in the originally acyclic graph, and depending on the delays on the arcs involved, the resulting strongly

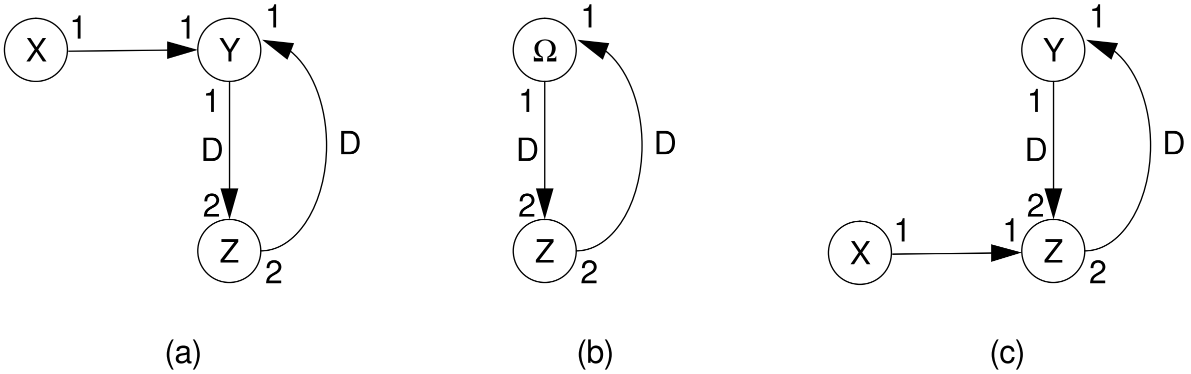
Now it can easily be verified that figure 7(b) is tightly interdependent. Thus any schedule based on this clustering decision will have more than one appearance of at least one block. In this case, the subschedule for {X, Y} will appear twice. However, the original graph, figure 7(a), has a single appearance schedule, since it is acyclic. So we see that although the clustering in figure 7 does not result in deadlock, it introduces a tightly interdependent subgraph, and thus it leads to less compact schedules.

Similarly, clustering a node that is not in any tightly interdependent subgraph with part of a tightly interdependent component can be detrimental. Such a cluster increases the extent of an existing tightly interdependent component. This is illustrated in figure 8. In figure 8(a) {Y, Z} forms a tightly interdependent component, and node X is not contained in any tightly interdependent subgraph. From property 3 of loose interdependence algorithms, any loose interdependence algorithm schedules figure 8(a) with only one appearance of X.



(a) (b)

Fig 7. A clustering decision that introduces tight interdependence.



(a) (b) (c)

Fig 8. A clustering decision that enlarges a tightly interdependent component.

maintaining code-size compactness. We have implemented such a combined program- and data-memory minimizing scheduler within the software synthesis environment for DSP that we have developed under *Ptolemy* [12].

Our observations suggest that for practical SDF graphs, tightly interdependent subgraphs are rare, and thus for most applications, any loose interdependence algorithm generates optimally compact schedules. However, we are investigating techniques to schedule tightly interdependent SDF graphs compactly to provide a more general solution. Improved methods for handling tightly interdependent components can be incorporated simply by replacing the tight scheduling algorithm of a loose interdependence algorithm.

5 Clustering in a Loose Interdependence Algorithm

We refer to the process of consolidating subgraphs as atomic units for scheduling, as illustrated in figure 6(d), as *clustering*. We have already shown in this paper how repeatedly clustering based on subindependence leads to provably compact programs. In [1], we also apply clustering to decrease data memory requirements for iterative schedules. In this section, we show that certain clustering decisions can interfere with code-minimization goals, and thus that if any clustering is to be incorporated into a loose interdependence algorithm — as a preprocessing step or as part of one of the component algorithms A_1, A_2, A_3 — then the possible negative effect on code compactness should be considered. We also present examples of how useful clustering techniques can be adapted to work in accordance with code minimization objectives.

A clustering decision always degrades the code-compaction potential of an SDF graph if it introduces a new tightly interdependent subgraph that is disjoint from the existing tightly interdependent components. For example, consider the acyclic SDF graph in figure 7(a), and suppose that we cluster the adjacent nodes X and Y into the hierarchical node Ω, as shown in figure 7(b). This is precisely the clustering that would be performed by the data-memory minimizing heuristic of [1], which clusters the pairs of adjacent nodes that are invoked most frequently, provided that the clustering does not introduce deadlock.

Property 2: Any loose interdependence algorithm constructs a single appearance schedule when one exists.

Property 3: If B is a node in the input SDF graph G, and B is not contained in a tightly interdependent component of G, then any loose interdependence algorithm schedules G in such a way that B appears only once.

Property 4. If B is a node within a tightly interdependent component of the input SDF graph, then the number of times that B appears in the schedule generated by a loose interdependence algorithm is determined entirely by its tight scheduling algorithm.

Property 4 states that the effect of the tight interdependence algorithm (A_3) is independent of the subindependence partitioning algorithm (A_2), and vice-versa. Any subindependence partitioning algorithm makes sure that there is only one appearance for each node outside the tightly interdependent components, and the tight scheduling algorithm completely determines the number of appearances for nodes 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 reduces data memory requirements more), we can substitute 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*.

Thus the class of loose interdependence algorithms defines a framework for implementing memory-minimizing schedulers. We have freedom to experiment with the component algorithms — the acyclic scheduling algorithm, subindependence partitioning algorithm, and tight scheduling algorithm — while the framework guarantees that the interaction of these algorithms will not hinder the full code-size minimization potential offered by subindependence partitioning. For example, the heuristic techniques of [1] can be incorporated into the acyclic scheduling algorithm or the tight interdependence algorithm to produce large savings in buffering requirements, while

```
Step 1: For each node N in G, determine the minimum number of times, $\mathbf{q}(N)$, that N is invoked in a periodic schedule for G.
Step 2: Determine the strongly connected components $G_1, G_2, \ldots, G_s$ of G.
Step 3: Cluster $G_1, G_2, \ldots, G_s$ and call the resulting graph G'. This is an acyclic graph.
Step 4: Apply $A_1$ to G'; denote the resulting schedule S'(G).
Step 5:
    for $i = 1, 2, \ldots, s$
        Apply $A_2$ to $G_i$;
        if subgraphs $X$ and $Y$ are found such that $X$ is subindependent of $Y$ in $G_i$,
        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 $r_x = gcd\{\mathbf{q}(N) | N \text{ is a node in } X\}$[1].
            • Let $r_y = gcd\{\mathbf{q}(N) | N \text{ is a node in } Y\}$.
            • Replace the (single) appearance of $G_i$ in S'(G) with $(r_x\ S_L(X))\ (r_y\ S_L(Y))$.
        else ($G_i$ is tightly interdependent)
            • Apply $A_3$ 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
Output S'(G).
```

Given a loose interdependence algorithm $\lambda = L(A_1, A_2, A_3)$, we refer to A_1, A_2, and A_3 respectively as the *acyclic scheduling algorithm* of λ, the *subindependence partitioning algorithm* of λ, and the *tight scheduling algorithm* of λ. The following useful properties of loose interdependence algorithms are proved in [2].

Property 1: Efficient loose interdependence algorithms exist. In particular, there are loose interdependence algorithms whose overall time complexity is quadratic in $\max(n, e)$, where n is the number of nodes in the input SDF graph, and e is the number of arcs.

---

[1] This is the number of times that G invokes the subgraph X; "gcd" denotes the greatest common divisor.

For example, consider the technique described in section 2 where we remove arcs having "sufficient" delay and then cluster the strongly connected components of the resulting graph G'. Since any root strongly connected component R_1 of G' is subindependent, any tightly interdependent component T is either completely contained in R_1 or completely contained in the complement $(G' - R_1)$ of R_1[1]. If $R_1 = T$, then we can decompose R_1 no further — this branch of the overall decomposition process terminates at R_1, and conversely if R_1 properly contains T, then R_1 must be loosely interdependent (otherwise T would not be a maximal tightly interdependent subgraph), so we can further subdivide R_1 via subindependence.

Similarly, if $(G' - R_1)$ contains T, and R_2 is a strongly connected component at the root of $(G' - R_1)$, then R_2 contains T or $(G' - R_1 - R_2)$ contains T. Clearly, by repeatedly applying this process, we will eventually arrive at a strongly connected component R_n that contains T. Furthermore, we will be able subdivide R_n further if and only if R_n properly contains T. Thus we see that this recursive subindependence decomposition method terminates at each of the unique tightly interdependent components of the original SDF graph.

4 Loose Interdependence Algorithms

Our memory-minimizing scheduling framework is based on a class of scheduling algorithms that we call **loose interdependence algorithms**. Given any algorithm A_1 for constructing a single appearance schedule for an *acyclic* SDF graph; any algorithm A_2 that determines whether a strongly connected SDF graph is loosely interdependent, and if so, finds a subindependent partition; and any algorithm A_3 that constructs a valid schedule for a tightly interdependent SDF graph, we define the *loose interdependence algorithm associated with* (A_1, A_2, A_3), denoted $L(A_1, A_2, A_3)$, as the following algorithm:

Algorithm $L(A_1, A_2, A_3)$

```
Input: an SDF graph G.
Output: a valid looped schedule S_L(G) for G.
```

---

[1] Strictly speaking, either the set of nodes in T is a subset of the set of nodes in R_1, or it is a subset of the set of nodes in $(G' - R_1)$.

ity for a subindependence-based scheduling algorithm. This is the form in which we have implemented subindependence partitioning.

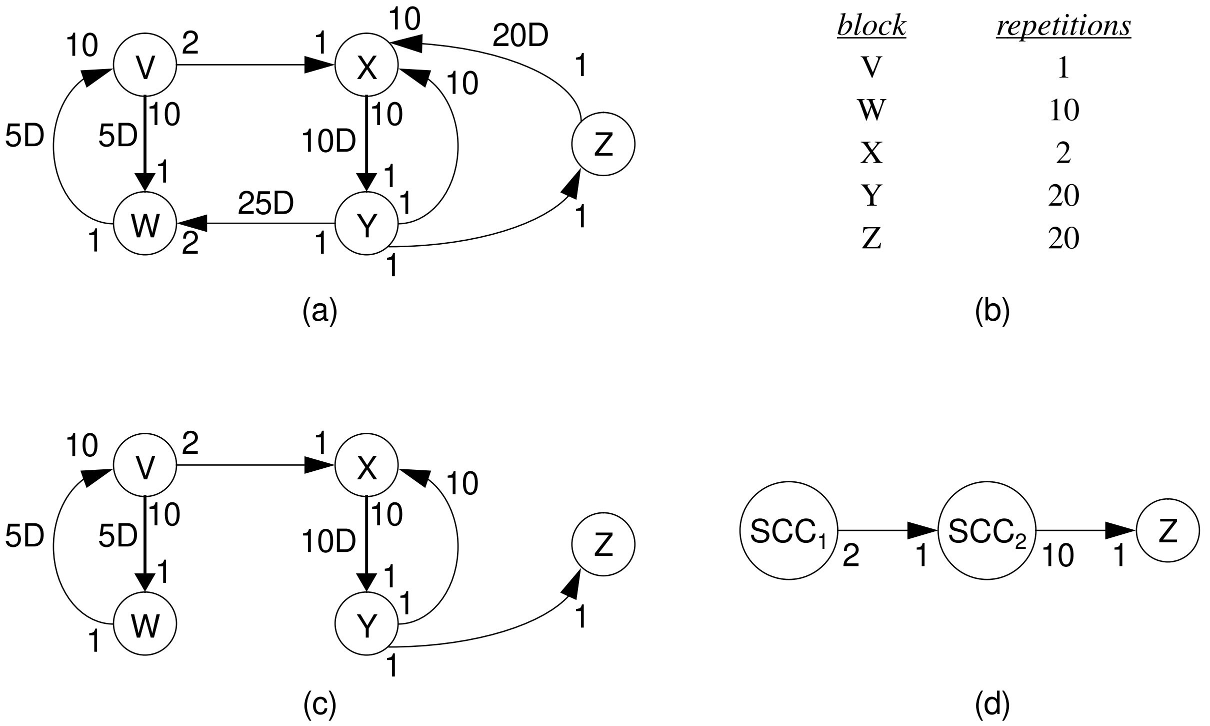
3 Loose and Tight Interdependence

If we can partition a strongly connected SDF graph G into two subgraphs such that one subgraph is subindependent of the other, then we say that G is **loosely interdependent**, and if a strongly connected SDF graph is not loosely interdependent, then we say that it is **tightly interdependent**. For example, the SDF graph in figure 5(a) is loosely interdependent. However, if we move one of the two delays to the lower arc, then the resulting graph, depicted in figure 5(b), is tightly interdependent — there is no way to partition this graph so that one part of the partition is subindependent of the other.

It can be shown that a tightly interdependent SDF graph never has a single appearance schedule, and an arbitrary SDF graph has a single appearance schedule if and only if it contains no tightly interdependent subgraphs. Thus, the graph in figure 5(b), and any SDF graph that contains this as a subgraph, does not have a single appearance schedule.

Another important property of tight interdependence is that it is *additive*: the union of two intersecting tightly interdependent SDF graphs is also tightly interdependent. Thus each SDF graph has a *unique* set of maximal connected tightly interdependent subgraphs, which we call its *tightly interdependent components.*

Finally, subindependent partitioning cannot "break up" a tightly interdependent component: if G is a strongly connected SDF graph, T is a tightly interdependent subgraph of G, and P_1 is subindependent of P_2 in G, then T is a subgraph of P_1, or T is a subgraph of P_2. An important consequence of this property is that for a given SDF graph, all subindependence-based decomposition techniques will terminate at the same subgraphs. With any subindependence partitioning algorithm, we will be able to repeatedly decompose an SDF graph until we are left only with the tightly interdependent components.



block	*repetitions*
V	1
W	10
X	2
Y	20
Z	20

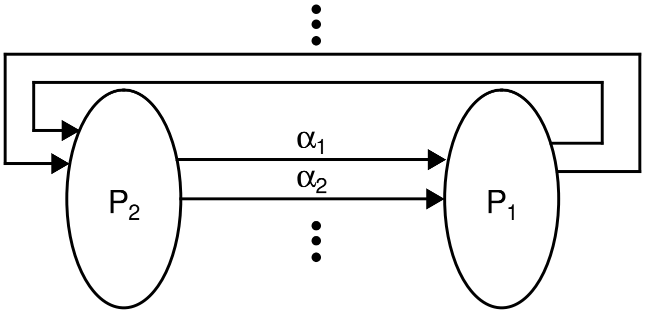
(a) (b)

(c) (d)

Fig 6. Partitioning a strongly connected SDF graph based on subindependence.

We can generalize this decomposition technique to get more scheduling flexibility. If we cluster each strongly connected component of G' then the resulting SDF graph is acyclic. This clustering process is illustrated in figure 6(d). Here the strongly connected components {V, W} and {X, Y} have been replaced by single nodes SCC_1 and SCC_2 respectively, and the SDF parameters on the input and output arcs of each SCC_i have been adjusted to reflect the total number of samples produced or consumed through one invocation *of the subgraph* SCC_i. For example, a minimal periodic schedule for {X, Y} invokes Y 10 times, so the number of samples produced on the arc directed from Y to Z is adjusted by a factor of 10.

We can construct a valid schedule for the graph in figure 6(a) by first constructing a schedule for the acyclic clustered graph of figure 6(d), and then replacing each appearance of an SCC_i with a minimal periodic schedule for that subgraph. The clustered graph in figure 6(d) reveals all possible subindependence partitions for the original graph: $\{SCC_1\}$ is subindependent of $\{SCC_2, Z\}$, and $\{SCC_1, SCC_2\}$ is subindependent of {Z}. Since the subindependence partition affects the final schedule, we see that clustering the strongly connected components allows the most flexibilnecessary condition for a strongly connected SDF graph to have a single appearance schedule is illustrated in figure 4. The condition is that we must be able to partition the nodes into two subsets such that one subset is precedence-independent of the other subset *throughout a single schedule period*. We refer to this form of data-independence as **subindependence**.



P_1 is subindependent of P_2 if:
For each α_i directed from P_2 to P_1,
(number of delays on α_i) ≥ (number of samples consumed from α_i each period).

Fig 4. An illustrated definition of subindependence.

Thus in a strongly connected SDF graph G, P_1 is subindependent of P_2 if all of the data required by P_1 from P_2 for a single schedule period of G is always available at the beginning of each schedule period. If this condition holds, then we can schedule G by scheduling all invocations associated with P_1 together, and then concatenating a schedule for all invocations in P_2. This simplifies the problem of constructing a single appearance schedule for G into the problem of constructing single appearance schedules for P_1 and P_2 separately. By repeatedly applying this type of decomposition, we can obtain a single appearance schedule whenever one exists.

Figure 5(a) illustrates how subindependence partitioning leads to single appearance schedules. Here, due to the two delays on the arc directed from block W to block X, the subgraph {X, Y} is subindependent of the subgraph {W, Z}. Thus we can obtain a single appearance schedule for the overall graph by constructing a single appearance schedule for {X, Y} and concatenating a single appearance schedule for {W, Z}. But it is easy to generate single appearance schedules for

---

1. A strongly connected component of a directed graph is a maximal subgraph C that has the property that between every distinct pair of nodes X nd Y in C, there is directed path from X to Y and a directed path from Y to X.

and Z are interleaved, and thus a separate activation is required for each invocation — 21 total activations are required. On the other hand, the schedule X(10 Y)(10 Z) requires only three activations — one for each block. In the objectives of [16], the latter schedule is preferable, because in that particular code-generation framework, there is a large overhead involved with each activation. With effective register allocation and instruction scheduling, such overhead can often be avoided, however, as [13] demonstrates. So we prefer the former schedule, which has less looping overhead and requires less data memory.

To understand the problem of constructing single appearance schedules, it is useful to review the manner in which *iteration* is specified in an SDF graph [9]. Figure 3 shows a simple example of nested iteration. Here, Y must be invoked twice for every execution of X, and Z must be invoked twice for every execution of Y. Two possible schedules for figure 3 are X (2 Y (2 Z)), which is a single appearance schedule that corresponds to a nested loop, and X (2 YZ) (2 Z), which is not a single appearance schedule.

A single appearance schedule can easily be constructed for an acyclic SDF graph. We pick a node at the root of the acyclic graph and schedule all invocations of this node in succession; remove this node from the graph and pick a root node of the remaining graph; schedule all invocations of this node in succession; and so on until we have traversed all nodes in the graph. This process yields the schedule X (2 Y) (4 Z) for the example in figure 3.

Clearly this simple scheduling technique will always yield a single appearance schedule for any acyclic SDF graph, although this will be the schedule that has the greatest buffering cost. To construct single appearance schedules with lower buffering costs via nested loops, we can apply the heuristic techniques of [1], but this must be done in a controlled manner to preserve compactness. We will elaborate on this in section 5.

It can be shown that an arbitrary SDF graph has a single appearance schedule if and only if each strongly connected component[1] considered separately has a single appearance schedule. A

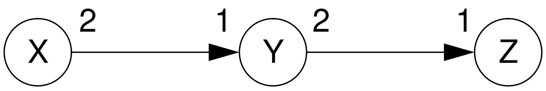


Fig 3. Nested iteration in an SDF graph.

niques form the foundation of the scheduling framework defined in this paper. The theoretical justification behind the developments in this and subsequent sections can be found in [2].

We define an optimally compact target program as one that corresponds to a schedule in which each block appears only once, and we refer to such schedules as *single appearance schedules*. For example, recall the two schedules XYZXY and (2 XY)Z for figure 1 that we discussed earlier. Blocks X and Y each appear twice in the first of these schedules, so this is not a single appearance schedule. If we use this schedule, then we must implement the repetition of X and Y in a schedule period by duplicating the associated code blocks or by inserting subroutine calls. However, we see that the second schedule, (2 XY)Z is a single appearance schedule, and thus with this schedule, we can implement all of the repetition in the algorithm through loops. This enables in-line code generation without a code-space penalty.

Neglecting the possible overhead associated with the loops, we see that any single appearance schedule leads to a minimum code size implementation. The overhead associated with loops is normally small with today's programmable DSPs, which commonly offer a "zero-overhead" looping facility. Also, loop overhead — in terms of code size — is independent of the amount of repetition involved in the loop. This is in contrast to subroutine calls, which require at least one extra instruction for each block invocation that occurs through a subroutine.

In this section we describe a necessary condition for an SDF graph to have a single appearance schedule, and we describe how it leads to a strategy for constructing single appearance schedules whenever they exist, and in the next section we show how it forms the basis for our scheduling framework. This necessary condition was developed independently, in a slightly different form, by Ritz et. al. [16], although their application of the condition is quite different from ours. Ritz. et. al. use these conditions in the context of an algorithm to synthesize *single appearance minimum activation* schedules, which minimize the number of "context-switches" between blocks. For example, in the schedule X(10 YZ) for figure 2, successive invocations of Y

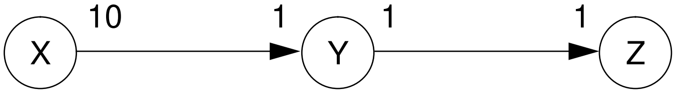


Fig 2. An example used to illustrate *minimum activation* schedules.

techniques. We demonstrate that this method for verifying the compatibility of a clustering decision with code-size minimization objectives applies to arbitrary clustering candidates, not just those that apply to buffer compaction. Another important application of "compatible clustering" that we have implemented is clustering to increase the amount of data transfers that do not go through memory — i.e. that take place only in registers.

Thus, we define a code scheduling framework that is centered on the goal of minimizing code size, but that can also be formally extended to incorporate other considerations, such as buffer compaction, as secondary objectives. Such a framework may not be suitable in some situations. For example, if an application involves a small set of operations (instructions) and an extremely large amount of data transfers, then the scheduler will probably have to consider data memory consumption as the primary objective to arrive at an acceptable solution. However, program memory constraints are usually more difficult to satisfy than data memory constraints. One reason for this is that data memory locations can be reused between noninterfering arcs [5], whereas there is usually no efficient way to have code for different parts of a graph reside in the same memory space. Also, it has been observed that schedules that are constructed solely for the purpose of minimizing buffering costs typically offer few opportunities for looping [8], whereas careful clustering can often be used to significantly reduce buffering requirements while maintaining a high degree of program compactness [1].

This type of *biased* scheduling framework is particularly suited for a goal-directed software synthesis environment, such as that described in [15]. However, we also believe that the insights involved in these techniques will be helpful in developing or refining schedulers that treat other objectives equally or with more priority than code size, but this area needs more investigation.

2 Subindependence

In this section we define a form of precedence independence that can be exploited to obtain an efficient algorithm for synthesizing optimally compact target programs. These techschedule for the graph and then replacing each instance of a block with the machine code segment that implements that block in the target processor. This process is described in detail in [7, 13].

The schedule determines to a large extent how much program and data memory will be required. For example, for figure 1, the schedule XYZXY leads to the least total buffering cost (over all schedules) since it consumes as many samples as possible from an arc before producing new data on the arc. However, due to the irregularity in the schedule, we cannot organize any loops in the target code. So, we must either duplicate the code blocks associated with blocks X, and Y in the target code, or we must arrange for X and Y to execute through subroutine calls — which results in less code duplication, but more subroutine call/return overhead. Thus, although the schedule offers low data memory requirements, it induces higher program size or subroutine penalty.

Another schedule for figure 1 is XYXYZ, which allows us to construct a loop in the target code. The looping opportunity offered by this schedule is highlighted if we express the schedule in the shorthand notation (2 XY) Z, where a parenthesized term (n N_1 N_2 … N_m) represents a loop whose iteration count is n, and whose loop body is N_1 N_2 … N_m. With this new schedule, we can implement all of the repetition in the algorithm without code duplication. However the buffering overhead is greater since none of the samples produced by block Y are consumed by Z until all invocations of Y have fired.

Programmable DSPs typically have a rather limited amount of program and data memory on chip, and there is usually a large speed penalty for accessing off-chip memory. For example, the Motorola DSP56001 has an on-chip capacity of 512 instruction and 512 data words, and Star Semiconductor's SPROC can store 1k instructions and 1k data. In the Motorola DSP56001, one on-chip instruction and two on-chip data words can be accessed in parallel, while there is only one external memory interface. Thus, there is a speed penalty for accessing off-chip memory regardless of how fast the external memory is. In this paper, we show how to minimize the program memory requirement of an SDF graph by constructing loops in the target code, and we show how to incorporate existing clustering heuristics for reducing data-memory requirements into our minimum code size scheduler *in a way that preserves the optimality of the code-size minimization*