Journal of VLSI Signal Processing, 6, 271-288 (1993)

Scheduling Synchronous Dataflow Graphs for Efficient Looping

SHUVRA S. BHATTACHARYYA AND EDWARD A. LEE
Department of EECS, University of California-Berkeley, Berkeley, CA

Received October 21, 1991; Revised April 23, 1992.

Abstract. Synchronous dataflow (SDF) has been used to synthesize code for programmable DSPs to implement multirate and block oriented signal processing systems. However, with large block sizes, or significant sample rate changes, program memory consumption becomes a critical problem. This article develops a compile-time algorithm for scheduling SDF graphs to exploit opportunities for looping—the successive reoccurrence of identical firing patterns. Because SDF graphs allow actors to produce or consume an arbitrary number of tokens on each input or output, complicated control flow may result. Yet in static scheduling, it is desirable to execute sections of the target code within loop constructs, such as "do-while," to reduce program-memory requirements. To do this, the SDF graph is hierarchically clustered, carefully avoiding deadlock while exposing looping opportunities. Results of applying these loop-extraction algorithms show orders of magnitude of compaction for target program code space on programmable DSPs compared to in-line code.

1. Introduction

At the University of California-Berkeley, we have automated the translation of synchronous dataflow (SDF) [1] specifications of signal processing algorithms into assembly code for multiple programmable DSPs [2]. Many others have used SDF or related semantics. For example, at Carnegie Mellon, Printz has automated the mapping of SDF graphs onto linear array architectures such as the Warp [3]. Many design environments (too numerous to name here) use *signal flow graphs* or *block diagrams* which have either SDF or closely related semantics. Some others use functional languages such as Silage [4], which has semantics very close to SDF. This paper addresses a scheduling problem pertinent in all these environments, and establishes fundamental relationships between iteration and SDF.

SDF is a special case of dataflow, which was pioneered by Dennis in 1975 [5]. An SDF graph is a directed graph of functional blocks, called actors, where the arcs in the graph represent the flow of streams of data from one actor to another. When an actor fires, it consumes some pre-specified number of data samples, called *tokens*, on each input, performs a computation, possibly updating its internal state, and produces a pre-specified number of tokens on each output. The number of tokens produced and consumed must be fixed and known at compile time, an important restriction that limits the applicability of SDF to synchronous multirate systems.

Among the advantages of using SDF programming to develop signal processing systems is the possibility for efficient compilation of very high level descriptions of multirate and block-oriented algorithms. The process, depicted in figure 1, involves maintaining a library of actor definitions, determining a partitioning and execution order for the actors, and then passing this ordering to a code-generator. It is often useful to derive from the SDF graph its associated acyclic precedence graph (APG) before beginning the scheduling process.

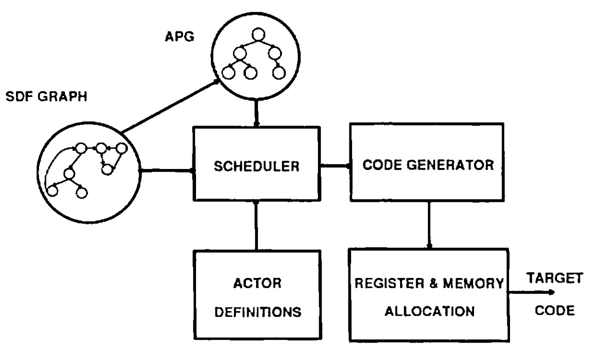


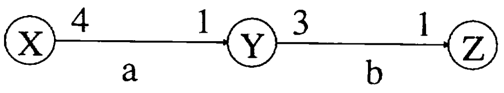
Fig. 1. Compiling a synchronous dataflow graph.

Programmable DSPs normally have a limited amount of program memory and data memory on chip. Additional memory requirements must be satisfied from off-chip memory, from which access can be significantly slower. It is thus highly desirable, and sometimes mandatory to have the code and data for a program fit entirely within the on-chip memory spaces.

This article focuses on scheduling. We assume a uniprocessor target with a Harvard architecture or one of its variants [6], [7] and generate execution orderings that make efficient use of the data and program memory spaces. We restrict our domain to single processors, but expect that the techniques can be extended to the multiprocessor case. As such, this article describes a compiler scheduling technique, and not a parallel scheduling technique.

1.1. Memory Usage

Some of the uniprocessor scheduling issues are illustrated by the example in figure 2 and table 1. In figure 2 we show an SDF graph and a uniprocessor schedule for that graph. The numbers adjacent to the inputs and outputs of each actor are the number of samples produced and consumed when the actor fires. In table 1 we show a profile—called a *buffer activity profile*—of the amount of data on each arc as the schedule is executed. Each column in the buffer activity profile represents an invocation (firing) of a node (actor) in the SDF graph. The invocations of a node X are labled X_1, X_2, . . . , X_n, where n is the number of times X is fired in the schedule. The rows in the profile correspond to arcs, and the entry for an arc α and an invocation I, denotes the number of samples residing on α immediately after I is fired. The row labeled total gives under each invocation I, the sum of the number of samples existing on all arcs, immediately following I's execution. Thus the largest value in the total row indicates the minimum number of words of data memory that is required to support the schedule.



Schedule: XYYYYZZZZZZZZZZZZ

Fig. 2. A synchronous dataflow graph and a schedule for the graph. The arcs from X to Y and Y to Z are labeled a and b respectively.

Table 1. The buffer activity profile for the graph in figure 2.

Arc	X_1	Y_1	Y_2	Y_3	Y_4	Z_1	Z_2	Z_3	Z_4	Z_6	Z_7	Z_8	Z_9	Z_{10}	Z_{11}	Z_{12}
a	4	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0
b	0	3	6	9	12	11	10	9	8	7	6	5	4	3	2	1	0
Total	4	6	8	10	12	11	10	9	8	7	6	5	4	3	2	1	0

In figure 3 and table 2 we show another schedule and buffer activity profile for the same graph. Examination of the "Total" row reveals that the memory require ments for this schedule are half of the previous one. Thus, we see that even for a very simple graph, scheduling choices can have a large impact on data memory requirements.

XYZZZYZZZYZZZYZZZ

Fig. 3. An alternative schedule for the graph of figure 2.

Table 2. The buffer activity profile for the schedule of figure 3.

Arc	X_1	Y_1	Z_1	Z_2	Z_3	Y_2	Z_4	Z_5	Z_6	Y_3	Z_7	Z_8	Z_9	Y_4	Z_{10}	Z_{11}	Z_{12}
a	4	3	3	3	3	2	2	2	2	1	1	1	1	0	0	0	0
b	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
Total	4	6	5	4	3	5	4	3	2	4	3	2	1	3	2	1	0

The impact of scheduling on program memory requirements is even greater. This impact occurs through the application of iterative programming constructs, or loops, to the target code across repetitive portions of the schedule. To illustrate this, we can express the schedule of figure 2 in a more compact form, which we call *looped form*, as X(4Y)(12Z). The following recursive definition makes this notation precise:

Definition. A schedule expressed in looped form has one or more parenthesized terms, of the form (N a_1, a_2, . . . a_n) where each a_i represents either a node in the graph, or a subschedule in looped form, and (N a_1, a_2, . . . , a_n) represents the successive repetition N times, of the firing sequence a_1, a_2, . . . , a_n. For example, *ABBABB* can be expressed in looped form, as both (2*ABB*) and (2*A*(2*B*)). We call a schedule expressed in looped form a looped schedule, and we call each parenthesized subschedule a schedule loop.

The looped schedule $X(4Y)(12Z)$ for figure 2 can be passed to the code generation phase of figure 1. For instance, if the target language is "C," then our example would produce code with the organization below:

```
main( ) {
        Code segment for X
        for (i = 0; i < 4; i++) {
                  Code segment for Y
        }
        for (i = 0; i < 12; i++) {
                  Code segment for Z
        }
}
```

If given any actor X, we let $S(X)$ denote the size in machine instructions of the in-line code segment for X, then the total program-memory requirement for the realization in example 1 is roughly[1] $S(X) + S(Y) + S(Z)$. This is a dramatic reduction over translating the graph in-line, which would require $S(X) + 4S(Y) + 12S(Z)$ words of program memory. Although the actual amount of improvement depends on the relative magnitudes of $S(X)$, $S(Y)$ and $S(Z)$, this example certainly illustrates that applying loops across repetitive sections of a schedule can produce orders of magnitude of compaction in target machine code space.

When we began investigating this problem, our first approach was to detect loops in schedules that had already been constructed under some criterion, such as minimizing data memory requirements or maximizing throughput. The problem, however, proved too subtle to yield to such simple methods. How [8] pointed out that these criteria often led to schedules in which looping opportunities were few. Hence, he proposed clustering the SDF graph by grouping actors with identical firing frequencies. This had to be done carefully, since some clusterings would lead to deadlocked schedules. Unfortunately, this method also failed to identify some attractive looping opportunities. Our third method is described in this article.

This article describes an evolution of scheduling heuristics for detecting and exploiting opportunities for looping. Code generation for such schedules has been investigated in detail, but will be postponed to a future article. Section 2 explains in detail why our preliminary approaches were not completely effective. Section 3 gives a third approach which remedies these shortcomings. Section 4 gives experimental results.

1.2. Objectives

We ultimately wish to solve the following uniprocessor scheduling problem:

Problem. Given an SDF graph G and a target machine M—which consists of a single *CPU*, P words of program memory, and D words of data memory—let V be the set of all valid schedules for G which result in programs for M whose program and data memory requirements are within P and D respectively. Find the element of V which executes in the smallest amount of time, or with the highest throughput.

This problem is extremely difficult. We have therefore chosen a much less ambitious, but nevertheless substantial objective. The first priority is the compaction of code space. Our scheduler is thus driven by the primary goal of exploiting opportunities for looping. Three considerations have led us to adopt this primary goal:

- Using data memory requirements as a primary objective can lead to an explosion of code space requirements, as shown below.
- Data memory locations can often be reused to buffer data from multiple noninterfering arcs, so in general, with an intelligent memory allocator, a compiler can meet data memory requirements much more easily than program memory constraints. There is no such mechanism, however, for having *code* for different regions of the graph reside in the same memory space.
- DSP algorithms frequently have substantial amounts of looping inherent in them.

We approach the problem of efficient data-memory utilization as a secondary goal. If two scheduling decisions produce identical code-space consumption, we will favor the decision which requires the least amount of data space, and thus data memory considerations are viewed as a tie-breaking criterion for the main goal of code-space compaction.

We ignore the direct impact of scheduling on the execution time of a program. For example, scheduling decisions determine whether or not invocations can find their respective input data in machine registers, rather than having to read from memory. Conversely an invocation may or may not need its output copied from a register to memory, based on the schedule. We also ignore the execution time overhead associated with loops. This overhead includes loop startup overhead, index count overhead, and overhead due to introducing spill code to maintain buffer address registers. These forms of overhead are minor, and often avoidable. For example, the Motorola DSP56000 and 96000 familes have a "zero-overhead" looping mode that eliminates the loop count overhead by performing the indexing in hardware.

Some of the tradeoffs discussed above are illustrated in figure 4 and table 3. The graph in figure 4 shows the graph of figure 2 with a downsampling of 2 appended (actor "U"). Two looped schedules for this graph are shown in table 3, with a summary of the associated buffering requirements. For both schedules the program memory requirement is $S(X) + S(Y) + S(Z) + S(U)$, the lowest possible for this graph. For the first schedule, $X(4Y)(12Z)(6U)$, the data memory requirement is

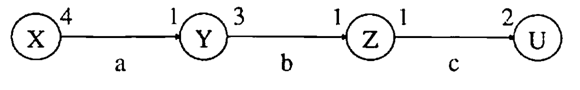


Fig. 4. An SDF graph used to illustrate tradeoffs in scheduling.

Table 3. A summary of the buffering requirements and loop overhead for two schedules for the graph in figure 4. The cost of initiating a loop is *s*.

Arc	Schedule: $X(4Y)(12Z)(6U)$	$X(2(2Y(3Z))(3U))$
a	4	4
b	12	6
c	12	6
Total buffer length	28	16
Loop startup overhead	3s	9s

28 words, while for the second schedule, $X(2(2Y)3Z))(3U))$, it is 16 words. The improvement is due to the nesting of loops in the second schedule and comes at the expense of increased loop overhead. If we let s denote the per-loop startup overhead, and we assume "zero-overhead" loop indexing, then the total overhead for each schedule is shown in the bottom row of table 3.

A third schedule without looping is shown in figure 5. This schedule minimizes the data memory requirements, reducing them to only 6 words. The buffer activity profile is shown in table 4. However, the absence of looping in this schedule results in a considerably larger code space of $S(X) + 4S(Y) + 12S(Z) + 6S(U)$.

XYZZUZYZUZZUYZZUZYZUZZU

Fig. 5. A schedule for the graph of figure 4 which does not apply looping.

Table 4. The buffer activity profile for the schedule of figure 5.

Arc	X_1	Y_1	Z_1	Z_2	U_1	Z_3	Y_2	Z_4	U_2	Z_5	Z_6	Y_4	Z_{10}	U_5	Z_{11}	Z_{12}	U_6
a	4	3	3	3	3	3	2	2	2	2	2	0	0	0	0	0	0
b	0	3	2	1	1	0	3	2	2	1	0	3	2	2	1	0	0
c	0	0	1	2	0	1	1	2	0	1	2	1	2	0	1	2	0
Total	4	6	6	6	4	4	6	6	4	4	4	4	4	2	2	2	0

The total data memory requirement is 6 words.

With respect to our current scheduling objectives, the second schedule of table 3 is the most desirable, since its code space efficiency is matched only by a schedule which consumes more data space. The tripling in loop startup overhead is neglected in our approach. The cost of the significant saving of data memory consumption in figure 5 is deemed to be too high.

2. Preliminary Approaches

The scheduling objectives defined in the previous section evolved out of observations based on two initial scheduling approaches. These approaches are described in this section.

2.1. *Post Optimization*

Our first approach to uniprocessor scheduling was to use a simple heuristic for minimizing data memory requirements [2], [9], [10]. This heuristic involves deferring nodes whose immediate descendants have sufficient data to fire until all descendants have used up their input samples, and are no longer firable. Furthermore, no node is scheduled twice until all other nodes have been tried. The technique is an intuitive way to keep excess samples from accumulating on arcs, and to thus keep overall buffering requirements low.

Our first approach for generating looped schedules was simply to post-process the minimum buffer-length scheduler with a pattern matching algorithm, which finds successively repeated sequences of firings [8]. The scheduler then groups such sequences into schedule loops. Thus in this approach, looping is not at all considered while constructing the ordering of invocations, and as a result, opportunities for creating schedule loops frequently go undetected.

Our previous example in figure 4 illustrates very well this conflict between minimum buffer length scheduling and scheduling to maximize looping. The schedule in figure 5 is obtained from the buffer minimization heuristic. Clearly this schedule fails to extract the looping which is inherent in the graph.

Minimum buffer length scheduling fails because it does not attempt to recognize identical firing patterns [8]. To improve the degree of looping in the schedule, scheduling decisions must be driven by a goal of detecting and grouping together repetitive series of computations.

2.2. *Grouping Connected Subgraphs of Uniform Frequency*

The first technique for considering looping while constructing the schedule was developed by How [8]. It involves isolating regions of the graph called *connected subgraphs of uniform frequency.* Before defining this term, we introduce some notation:

Notation. An SDF graph G can be expressed as a set $\{N, \Lambda\}$, where N is the set of nodes in G and Λ is the set of arcs in G. We say that $G = \{N, \Lambda\}$. For any SDF arc α, we denote by $p(\alpha)$ the number of samples produced onto α during an invocation of α's source node. Similarly, the number of samples consumed from α by α's sink node is denoted $c(\alpha)$. Finally, given a directed graph P, the subgraph associated with a set of nodes M in P is the graph $\{M, \Theta\}$, where Θ is the set of arcs that connect a node in M to another node in M.

We now define a class of subgraphs for an SDF graph. This section wil demonstrate how such subgraphs can be used to produce looped schedules.

Definition. Given an SDF graph $G = \{N, \Lambda\}$, suppose $M \subset N$, and let $H = \{M, \Omega\}$ denote the subgraph associated with M. We say that H is a connected subgraph of uniform frequency, abbreviated CSUF, if and only if the following two conditions hold:

(1) H is a connected graph.
(2) $\forall\ \alpha \in \Omega,\ p(\alpha) = c(\alpha)$.

If $H = \{M, \Omega\}$ is a CSUF, we also say informally, that M is a CSUF.

Thus, a CSUF is a connected set of nodes with no sample rate changes between them. Note that this does not necessarily mean that $p(\alpha)$ and $c(\alpha)$ are uniform across Ω. For example, the graph in figure 6 is a valid CSUF. The main point is that within any valid periodic schedule S for G, each member of a CSUF has the same total number of invocations.

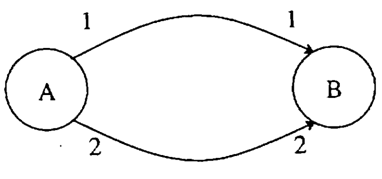


Fig. 6. A CSUF with nonuniform $p(\alpha)$.

Whenever its consolidation results in a graph for which a schedule exists, a CSUF M can be treated by a scheduler as a *supernode* that is fireable whenever all external inputs to M are available. An invocation of M corresponds to firing each node inside M once, and a schedule for this aggregate firing can easily be constructed, since M is assumed to be connected.

In figure 7 through figure 9 we illustrate how partitioning a graph into CSUFs can lead to looped schedules.

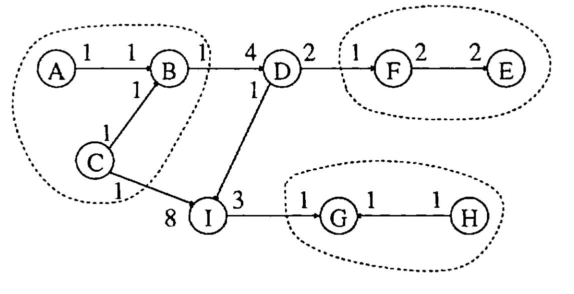


Fig. 7. A multirate graph with three CSUFs—$\{A, B, C\}$, $\{E, F\}$ and $\{G, H\}$.

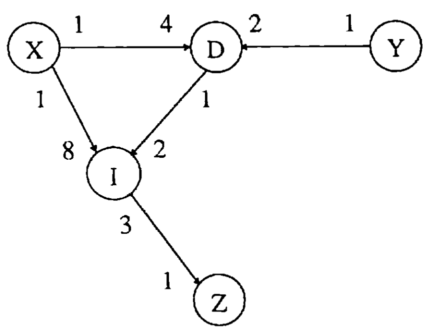


Fig. 8. The topology that results from considering each CSUF in figure 7 as a single node. X, Y and Z are used to represent $\{A, B, C\}$, $\{E, F\}$, and $\{G, H\}$ respectively.

Schedule for the clustered graph :

(2(4X)(2Y))DI(3Z)

The flattened schedule : (2(4ACB)(2FE))DI(3HG)

Fig. 9. Scheduling the clustered graph of figure 8. The first schedule considers each cluster as an atomic unit, and the second—"flattened"—schedule is obtained by replacing each appearance of a cluster in the first schedule, with a subschedule for that cluster.

The encircled regions in figure 7 outline three nontrivial CSUFs—$\{A, B, C\}$, $\{E, F\}$, and $\{G, H\}$. Each of these regions is initially considered by the scheduler as a single unit, as shown in figure 8. The minimum buffer size scheduling heuristic of the previous section can then be applied to this clustered graph. By this we mean that each level of the hierarchy is scheduled separately. The result is given in figure 9. The first schedule in figure 9 shows the schedule for the clustered graph, and this schedule is flattened—the CSUF supernodes are replaced with their respective subschedules—to obtain the second schedule.

Note that the loops in this looped schedule are based on the three CSUFs. Had these CSUFs not been consolidated as individual units, as in figure 8, the scheduler could have interrupted a repetitive sequence to invoke a fireable node from some other region of the graph. For example, figure 10 shows the first several firings of the schedule generated by our minimum buffer size heuristic applied to the unclustered graph. Observe how the CSUF $\{A, B, C\}$ is interrupted by the first firing of H, and thus the looping inherent in the connection of $\{A, B, C\}$ to the downsampled input of D goes unexploited. The invocation of H so early in the schedule also precludes exploiting the upsampled CSUF $\{G, H\}$.

ABCHEFACBEFACBACBD......

Fig. 10. The first several firings of a schedule for the graph of figure 7 using the minimum buffer size heuristic described in section 3.

The capability of CSUF-driven scheduling is well matched to DSP algorithms, since signal processing systems frequently consist of single-sample-rate subsystems, with sample-rate changes occurring only at scattered interface points. The effectiveness of the CSUF approach was demonstrated upon its incorporation within a compiler which translates SDF graphs into assembly code for the Motorola DSP56000 programmable DSP [8].

Although CSUF-based scheduling greatly improves the ability to extract looping from SDF graphs, it has two major limitations. The first shortcoming is illustrated in figure 11.[2] Here the formation of the CSUF $\{A, B, C, F\}$ results in a deadlocked clustered graph. The deadlock arises because the source node A has been subsumed by a supernode which is no longer a source. The execution of the graph must begin with A, but the supernode containing A needs external data to fire. A similar situation may occur when an arc with nonzero delay is subsumed by a CSUF.

Thus $\{A, B, C, F\}$ must be decomposed to retain as large a CSUF as possible without creating a deadlocked graph. The desired partition is shown in figure 12, along with the resulting looped schedule. Unfortunately, we have been unable to deduce a general solution to the problem of optimally decomposing a CSUF in a deadlocked clustered graph.

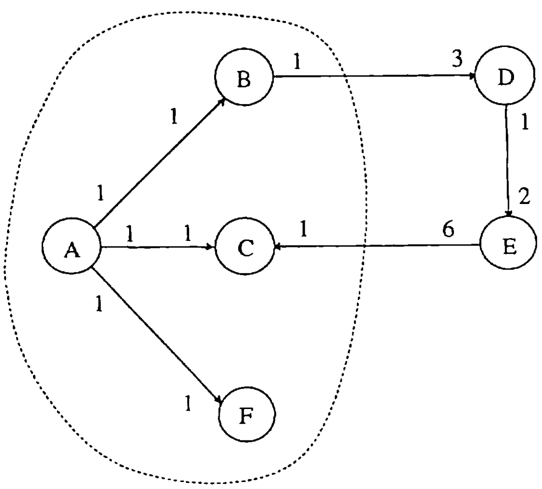
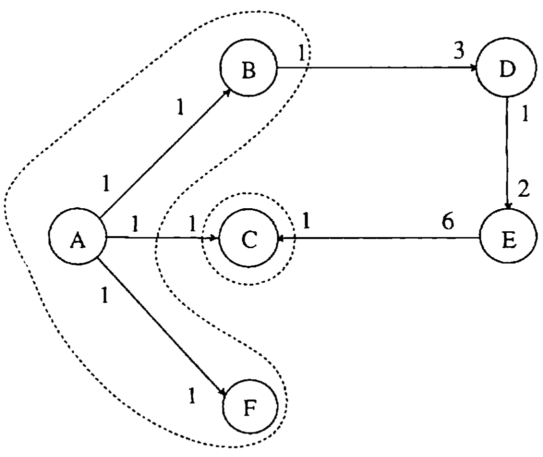


Fig. 11. The consolidation of the CSUF $\{A, B, C, F\}$ introduces a directed delay-free loop.



Schedule: (2(3ABF)D)E(6C)

Fig. 12. The desired partition of the cluster in figure 11 and the resulting looped schedule.

The second shortcoming of the CSUF approach arises from its inability to detect looping which occurs across sample rate changes. In figure 13 we show a graph with opportunities for this kind of looping,

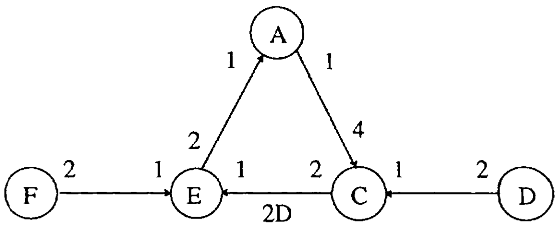


Fig. 13. An SDF graph which offers opportunities for looping that span sample rate boundaries.

D(2F(2E(2A))C)

Fig. 14. A looped schedule for the graph of figure 13.

and in figure 14 we show a looped schedule for this graph. Although figure 14 reveals that a large amount of looping is inherent in this graph—enough to allow an implementation with only one code segment per actor—clearly none of the looping results from CSUFs, since avery arc involves a sample rate change. In this case, the CSUF-driven schedule is the same as what the minimum buffer size technique yields. The result of passing this schedule through a pattern-matching postprocessor is shown in figure 15. Clearly this schedule applies significantly less looping and requires much more code space than that of figure 14. It fails to recognize repeated firing patterns across *F*, *E* and *A*. As a result, *D* is allowed to fire midway through the schedule, and this breaks up the nested loop which could have spanned almost the entire program.

F(2E(2A))FDC(2E(2A))C

Fig. 15. The schedule for the graph of figure 13 which is obtained from the CSUF approach. The schedule is much less compact than that of figure 14.

This section has demonstrated that scheduling techniques that do not attempt to recognize looping while constructing the schedule miss opportunities for looping. However, we have also shown that the CSUF method has two serious limitations—the possible introduction of deadlocks, and the inability to consider loops that span regions of different sample rate. We now give a generalized version of this technique that overcomes these limitations.

3. Pairwise Goruping of Adjacent Nodes

In this section we present an enhanced technique for hierarchically clustering the SDF graph in order to expose opportunities for looping. This method systematically handles any deadlocks which result from the cluster building process. Furthermore, the technique considers looping opportunities irrespective of whether or not they cross sample-rate boundaries. This uniform treatment leads to much better performance than the CSUF-based approach for graphs which contain many sample-rate changes. Finally, by its emphasis on hierarchical pattern-building, this improved scheduling algorithm favors nesting loops, rather than cascading them. As illustrated in figure 3, nesting loops require less buffering without increasing code-space requirements. Thus, our improved scheduler performs in accordance with the scheduling objectives outlined in Section 1.2.

Like the CSUF approach, the method described in this section repeatedly consolidates groups of nodes in the SDF graph. There are two primary differences, however, in the procedure for selecting the nodes which are to be formed into a cluster at a given step in the algorithm:

(1) Whereas CSUF clusters can involve an arbitrary number of nodes, our new method forms clusters with only two nodes at a time. The primary motivations for this incremental approach to cluster-building are to effectively organize nested iteration and to isolate the causes of deadlocks as they arise. We will elaborate on these considerations later in the section.
(2) The nodes which comprise a cluster can be of mutually differing frequencies. This allows us to exploit looping opportunities that involve sample-rate changes.

Recall that a cluster in an SDF graph is a group of nodes or subclusters that the scheduler considers as an indivisible unit to be invoked without interruption. Thus, we again require that clusters involve connected sets of nodes. It is actually possible to consider nonconnected sets of nodes. However, doing so does not improve our ability to conserve memory, since a schedule loop involving two nonconnected subsets of nodes C_1 and C_2 can be divided into separate loops for C_1 and C_2, without affecting the buffering requirements. The primary advantage of encapsulating such subgraphs within the same loop is the reduction of loop startup and index count overhead, but this benefit does not relate to our current scheduling objectives. We expect that our future work will pursue the issue of clusters containing nonconnected subgraphs.

Since clustering decisions in our enhanced scheduling technique involve pairs of connected nodes, we call the method *Pairwise Grouping of Adjacent Nodes*, abbreviated PGAN. The following definitions make precise the concept of adjacency in an SDF graph:

Definition. If *X* and *Y* are two distinct nodes of an SDF graph, then *X* is said to be a successor of *Y* iff there is an arc directed from *Y* to *X*. *Y* is called a predecessor of *X* iff *X* is a successor of *Y*. Finally, we say that *X* is adjacent to *Y* iff *X* is a successor of *Y* or *X* is a predecessor of *Y*.

The PGAN algorithm involves repeatedly selecting pairs of adjacent nodes to consolidate into clusters. We refer to the steps taken to choose and form a cluster as an iteration of the algorithm, and we adopt the convention of indexing the iterations with positive integers. In each iteration, the graph is modified by replacing a pair of adjacent nodes with a single node representing the cluster.

We will illustrate the procedure for selecting clusters, and then present a more detailed description of the overall algorithm. First, however, we digress to consider looping from the perspective of the APG (acyclic precedence graph), and to relate these considerations to the SDF graph.

3.1. *Examining the APG for Looping*

It is instructive to view the formation of clusters in an SDF graph with respect to the impact on the corresponding APG. The APG is particularly illustrative in discussing looping, since it explictly shows the repetition inherent in an algorithm. We depict in figure 16 and figure 17 respectively a multirate SDF graph and its associated APG. The optimal looped schedule with respect to the objectives defined in Section 2 can easily be obtained from inspection of the precedence graph—$(2D)(3A(2(3EFG)H))BC$. Note that this result is much more difficult to deduce from examination of the original SDF graph.

Looping information is visually easier to extract from a well-drawn APG because looping opportunities are manifested as repeated subgraphs. These subgraphs

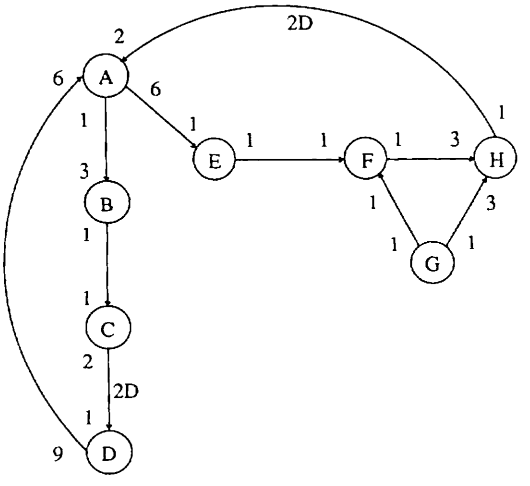


Fig. 16. A multirate SDF graph which offers several opportunities for looping.

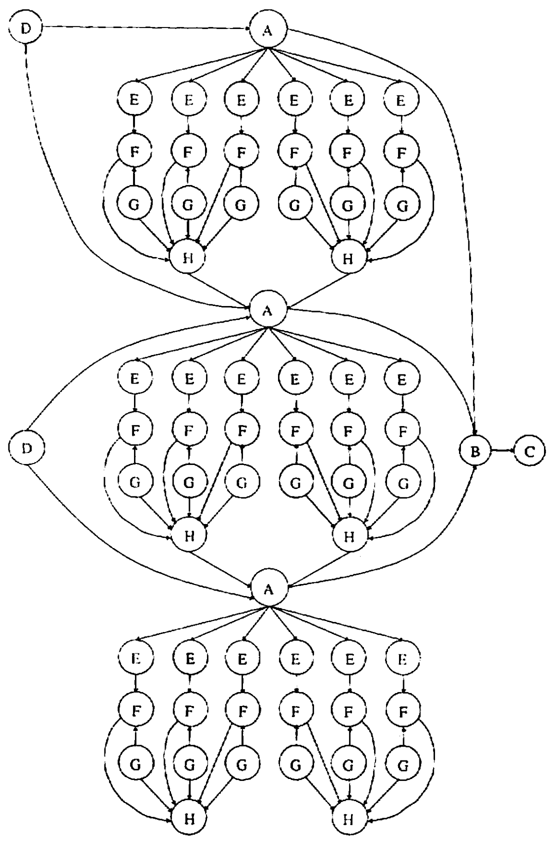


Fig. 17. The APG for the graph of figure 16. Observe that the APG representation exposes very clearly the looping inherent in the original SDF graph.

can be considered as hierarchical clusters in a manner analogous to our interpretation of clusters in the SDF graph. In figure 18, table 5 and figure 19 we illustrate this process for the example of figure 16. The series of graphs in figure 18 shows a succession of cluster formations, each involving two or more repeated subgraphs. In table 5 we list looped schedules for the root graph and each cluster, and in figure 19 we give the result of recursively replacing the appearance of each cluster with its subschedule in the root schedule. This result agrees with the "optimal" schedule.

Since each of the clusters in figure 18 spans all invocations of the nodes which it subsumes, they all correspond to hierarchical clusters in the original SDF graph, as well as clusters of the APG. The equivalent sequence of cluster formations in the original SDF graph is shown in figure 20.

An example of an APG subgraph which does not translate to the SDF graph is the consolidation of the first and second invocations of D with, respectively,

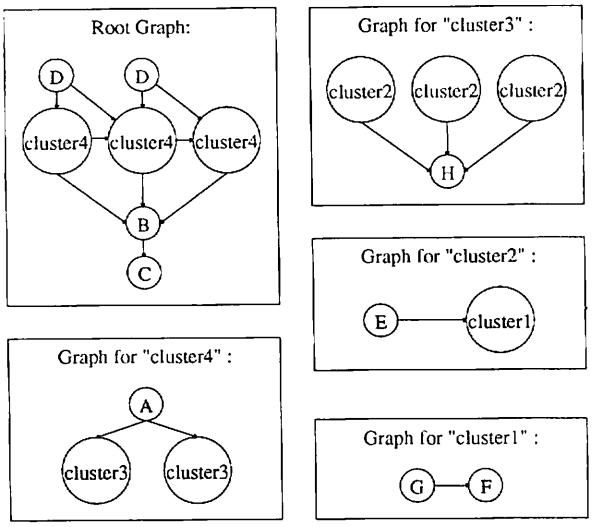


Fig. 18. A hierarchy of clusters of repeated subgraphs for the APG of figure 17.

Table 5. The schedule for each of the clusters depicted in figure 18.

Graph	Schedule
Root	(2 *D*) (3 *cluster4*) *B C*
cluster4	*A* (2 *cluster3*)
cluster3	(3 *cluster2*) *H*
cluster2	*E cluster1*
cluster1	*G F*

(2D)(3A(2(3EGF)H))BC

Fig. 19. The looped schedule suggested by the hierarchical decomposition of figure 18.

the first and second invocations of *cluster4* in figure 18. The formation of this cluster and the resulting schedule are depicted in figure 21. Observe that the code space size is no longer one code segment per node; instead the code corresponding to a very large subgraph—that for *cluster4*—must appear twice in the target program. The large increase in code size results from the inability to encompass all invocations of *cluster4* within the newly formed schedule loop involving *D* and *cluster4*. Clearly a schedule loop *L*, involving *D* and *cluster4* could span all invocations of *cluster4* only if the ratio of appearances of *D* to the number of appearances of *cluster4* in *L* was equal to the ratio of the total number of invocations of *D* to the total number of *cluster4* invocations. This is precisely the condition which relates clusters in an SDF graph to clusters in the associated APG. We summarize and elaborate on these points with following definition and fact.

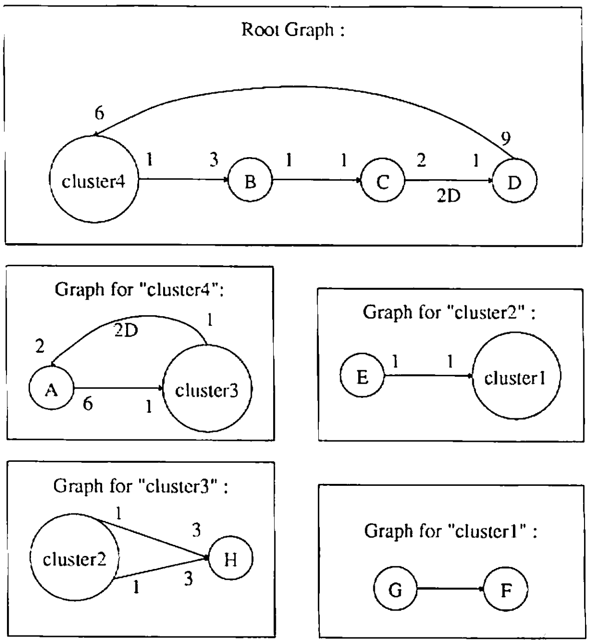
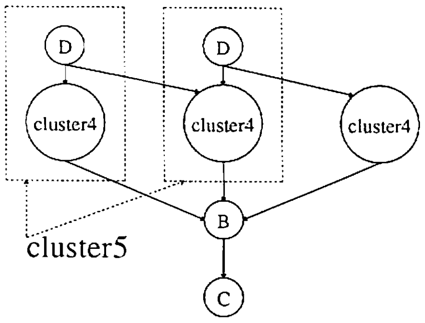


Fig. 20. The hierarchy of subgraphs in the SDF graph which corresponds to the organization of APG clusters shown in figure 18.



Schedule:(2 cluster5) cluster4 B C <->
(2DA(2(3EGF)H))A(2(3EGF)H)BC

Fig. 21. The consolidation of a repeated APG subgraph which does not correspond to a cluster in the SDF graph, and the resulting schedule.

Definition. Let $G = \{N, \Lambda\}$ be an SDF graph and let P denote its associated APG.[3] For any $A \in N$, we define the frequency of A, denoted $\upsilon(A)$, to be equal to the number of invocations of A appearing in P. For $A_1, A_2 \in N$, we define

$$F(A_1, A_2) = \upsilon(A_1) \,/\, gcd(\upsilon(A_1), \upsilon(A_2)),$$

where *gcd* denotes the greatest common divisor operator.

If we form a cluster with two nodes A_1 and A_2, we can interpret this cluster as $gcd(\upsilon(A_1), \upsilon(A_2))$ repetitions of a group of firings involving $F(A_1, A_2)$

invocations of A_1 and $F(A_2, A_1)$ invocations of A_2, since $F(A_1, A_2) / F(A_2, A_1)$ is the proper fraction form of the frequency ratio between A_1 and A_2. The following fact, which is proved in [11], expresses this observation in terms of the impact on the APG of forming a cluster in an SDF graph.

Fact. *Let* G be an SDE graph and let P be its associated APG. Let A be a pair of nodes of G, and let $\hat{G}$ denote the SDF graph which results from consolidating A and B into a cluster Ψ in G. If the formation of Ψ does not produce a deadlocked graph, and we let $r = gcd(v(A), v(B))$, then the APG $\hat{P}$ for $\hat{G}$ can be obtained by combining

$$\{A_1, A_2, \ldots, A_{F(A,B)}, B_1, B_2, \ldots, B_{F(B,A)}\};$$

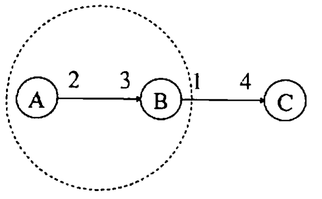
$$\{A_{F(A,B)+1}, A_{F(A,B)+2}, \ldots, A_{2F(A,B)}, B_{F(B,A)+1}, B_{F(B,A)+2}, \ldots, B_{2F(B,A)}\};$$

$$\ldots$$

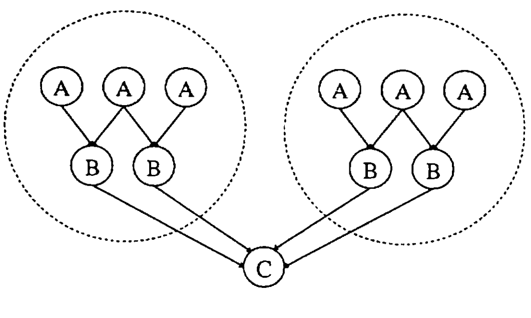
$$\{A_{(r-1)F(A,B)+1}, A_{(r-1)F(A,B)+2}, \ldots, A_{rF(A,B)}, B_{(r-1)F(B,A)+1}, B_{(r-1)F(B,A)+2}, \ldots, B_{rF(B,A)}\}$$

into $\Psi_1, \Psi_2, \ldots, \Psi_r$ respectively.

In figure 22 and table 6 we illustrate the significance of $F(*,*)$, and $v(*)$ in relating a cluster in the SDF



(a) A cluster in an SDF graph.



(b) The associated APG cluster.

Fig. 22. An example that illustrates the significance of $F(*,*)$ and $v(*)$.

Table 6. Quantities elating the SDF cluster and the APG cluster of figure 22.

- $v(A) = 6$
- $v(B) = 4$
- $gcd(v(A), v(B)) = 2$
  = The number of invocations of the APG cluster.
- $F(A, B) = v(A) / gcd(v(A), v(B)) = 3$
  = The number of invocations of A per cluster invocation.
- $F(B, A) = v(B) / gcd(v(B), v(A)) = 2$
  = The number of invocations of B per cluster invocation.

graph to its APG counterpart. The above fact and the example in figure 21 suggest that a large increase in code size can result from forming a cluster in the APG which does not have a counterpart in the SDF graph. For this reason we do not consider the consolidation of repeated APG subgraphs that do not correspond to clusters in the associated SDF graph, and thus our clustering decisions can operate directly on the SDF graph. However, later we will show that we can check for deadlocks more efficiently by working with the APG. This consideration renders the APG more suitable than the SDF graph for the implementation of PGAN.

The PGAN algorihm involves repeatedly selecting pairs of nodes as clusters to expose opportunities for scheduling nested loops. Before coalescing a candidate cluster we must first verify that its formation will not result in a deadlocked graph. In the next section we will develop our technique for selecting candidate clusters and then address the problem of avoiding deadlock.

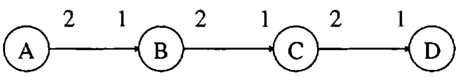
3.2. Cluster Selection

Our development of the PGAN algorithm began with an approach that selects the two nodes of a candidate cluster one at a time. The first node, called the *base node* for the cluster, is simply the node which we consider most likely, at the current algorithm iteration, to be contained in the deepest level of a nested loop. Choosing the second node then involves selecting which of the nodes adjacent to the base node will be combined with it to form the candidate cluster. We will present preliminary criteria for choosing the base node and the adjacent node for this initial approach, illustrate a shortcoming, and develop improved selection criteria.

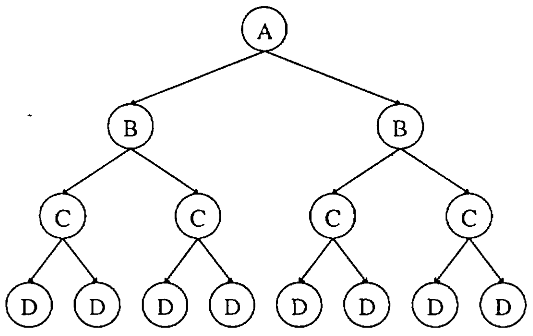
3.2.1. Selecting the Base Node.

Policy 1. The base node at a given algorithm iteration is chosen as the highest frequency node which has not already been considered as a base node.

We prioritize nodes based on frequency because to recognize a nested loop construct, the inner—or higher frequency—loops must be coalesced before committing clusters to the outer regions. This requirement is revealed in figure 23 through figure 25. In figure 23, we have an SDF graph and the associated APG for a simple example of nested iteration. The result of first coalescing nodes *B* and *C*, which both have lower frequency than *D*, is shown in figure 24. The resulting schedule does not fully exploit the nested loop inherent in the original graph. On the other hand, if we start the clustering process with the highest frequency node *D*, then we can obtain the desired nested loop structure in the final schedule, as shown in figure 25.

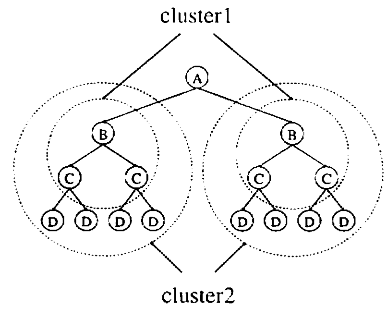


(a) An SDF graph ...



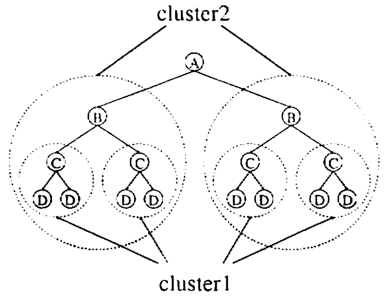
(b) and its APG.

Fig. 23. An SDF graph which suggests a nested loop. Scheduling this graph into this nested loop requires proper selection of the base node.



Schedule:
A(2 cluster2) <-> A(2 cluster1 (4D)) <-> A(2B(2C)(4D))

Fig. 24. The result of first coalescing nodes of lower frequency in the APG of figure 23. The schedule does not exhibit the optimal nested looping.



Schedule:
A(2 cluster2) <-> A(2B(2 cluster1)) <-> A(2B(2C(2D)))

Fig. 25. Clustering the highest frequency nodes first achieves the desired nested looping.

Since the cluster hierarchy translates directly to the hierarchy in the loops of the resulting schedule, the innermost clusters—the clusters which we create first—must correspond to the desired inner loops. Since the inner loops are the most frequently executed, it follows that nodes with the highest frequency must be involved in the earliest clustering decisions.

3.2.2. Selecting the Adjacent Node. The selection of the base node reflects the section of the graph which is most likely to be involved in the innermost loop of a nested loop. Now we address the problem of choosing the node adjacent to the base node to include in a candidate cluster. We refer to this second node as the *adjacent node*. Again, our aim is to detect opportunities for nested loops, whenever they are present.

In figure 26 through figure 28 we give an example in which the detection of nested looping depends upon proper selection of the adjacent node. In figure 26, an SDF graph and its associated APG are shown. From

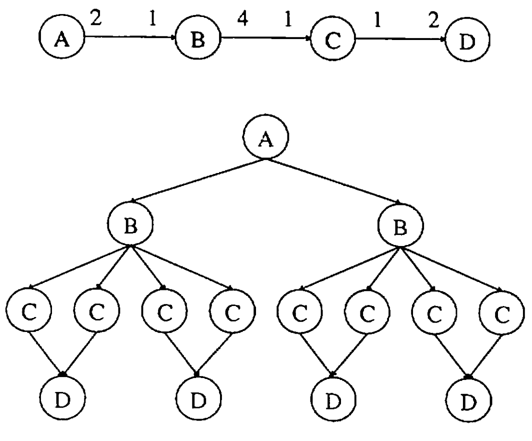
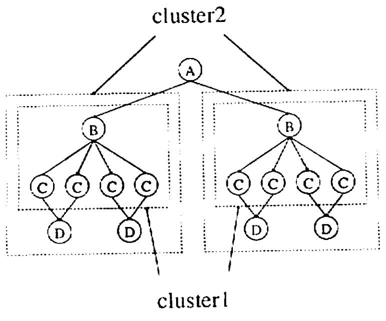
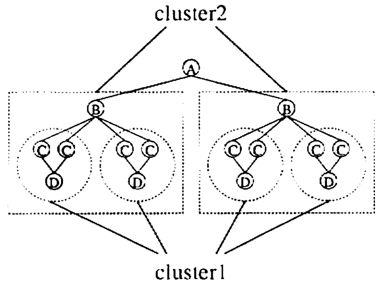


Fig. 26. An example used to illustrate the impact which the selection of the adjacent node can have on the schedule.



Schedule:
A(2 cluster2) <-> A(2 cluster1 (2D)) <-> A(2B(4C)(2D))

Fig. 27. Selecting the candidate of lower frequency as the adjacent node. The resulting schedule does not fully exploit the nested looping suggested by this graph.



Schedule:
A(2 cluster2) <-> A(2 B (2 cluster1)) <-> A(2B(2(2C)D))

Fig. 28. Selecting the higher frequency candidate *D* as the adjacent node leads to the desired nested looping.

these graphs, we see that *C* must be the base node for the initial clustering decision. The choice of *C* as the base node presents two possibilities for the adjacent node—*B* and *D*. In figure 27 and figure 28 we detail the consequences of choosing *B* and *D*, respectively, as the adjacent node. Comparison of these figures reveals—in a manner analogous to that of figure 24 and figure 25—that the appropriate adjacent node for achieving optimal nested looping is the node with the higher frequency, *D*.

This illustration at first glance persuades us to always choose the highest frequency candidate for the adjacent node, just as with the selection of the base node. However, more careful consideration reveals that we must also consider the relationship in frequency between each candidate adjacent node and the base node. Consider figure 29 through figure 31, which present a multirate graph, and in the same manner depict the respective consequences of selecting each of the two possible adjacent nodes for the initial base node *C*. Observe that selecting the lower frequency node, *B*, results in a schedule with the desired nested loop,

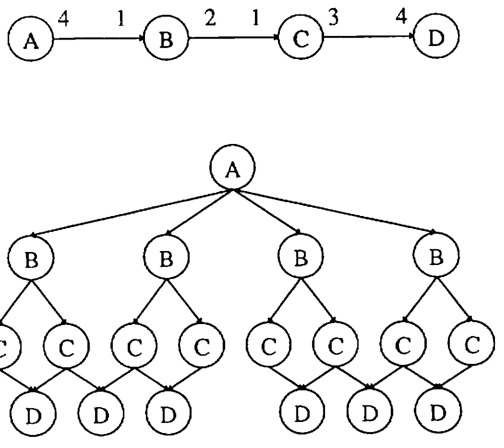
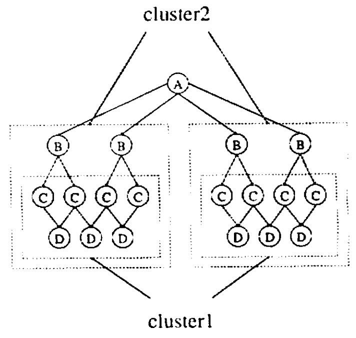
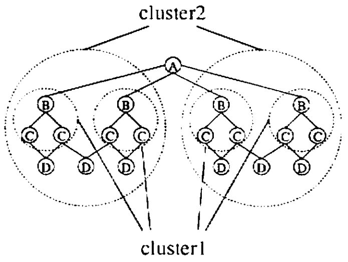


Fig. 29. This example illustrates that frequency should not be the only criterion for selecting the adjacent node.



Schedule:
A(2 cluster2) <-> A(2 (2B) cluster1) <-> A(2(2B)(4C)(3D))

Fig. 30. Selection of the higher frequency candidate *D* does not result in the nested loop suggested by the graph of figure 29.



Schedule:
A(2 cluster2) <-> A(2(2 cluster1)(3D)) <-> A(2(2B(2C))(3D))

Fig. 31. Selecting the lower frequency candidate, *B*, results in the desired nested looping.

whereas the result of selecting the node of higher frequency is suboptimal. This result arises from the fact that combining *C* and *D* commits four invocations of *C* to each invocation of the resulting cluster, while there is a different repeated subgraph involving fewer invocations of C (per loop iteration). This in turn suggests that we revise our policy to select the adjacent node candidate which matches up with the fewest number of base node invocations within a single invocation of the resulting cluster. We use the notation developed in the previous subsection to state this policy more precisely.

Policy 2. Suppose that we are given a base node B, and suppose S is the set of all nodes which are adjacent to B. Then we choose as the adjacent node that member A of S for which $F(B, A)$ is minimum.

3.2.3. Selecting the Cluster of Highest Frequency. Policies 1 and 2 together comprise our initial approach for selecting clusters to organize nested looping. Our development of policy 2 illustrates that for a given base node B, we should choose as the adjacent node the candidate A which minimizes $F(B, A)$. Since $F(x, y) = v(x) / gcd(v(x), v(y))$, we see that this is equivalent to choosing the node which maximizes $gcd(v(B), v(A))$. Thus among the nodes adjacent to B, we choose the one which combines with B to form the cluster of highest frequency.

This interpretation of policy 2 suggests an alternative criterion for cluster-selection: we simply choose the pair of mutually adjacent nodes which results in the cluster of highest frequency. This policy agrees with the method above whenever the highest frequency cluster contains the highest frequency node—but clearly this need not be the case.

In figure 32 through figure 34, we illustrate an example in which cluster selection based on the base node and adjacent node approach fails to identify the highest frequency cluster. Since A is the node of highest frequency, it will be chosen as the base node for the initial, "inner" cluster, and the result is a cluster of frequency 2. As figure 34 shows, however, first coalescing B and C into a cluster of frequency 4 leads to the desired nested looping.

To aid in summarizing these observations, we introduce the following definition, which introduces notation for the frequency of a pair of nodes.

Definition. Let $P = (A, B)$ be a pair of nodes in an SDF graph. Then we define $v(P)$, called the frequency of P, by $v(P) = gcd(v(A), v(B))$.

We now summarize with the follwoing policy, which specifies the cluster-selection criteria for the PGAN algorithm at any given algorithm iteration.

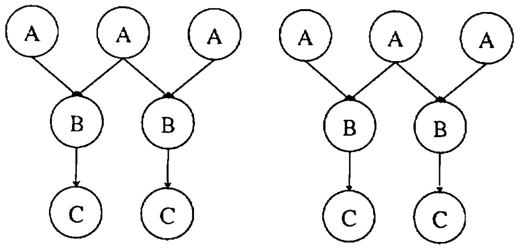


Fig. 32. An example used to illustrate that cluster selection based on the base node and adjacent node can fail to extract the highest frequency candidate.

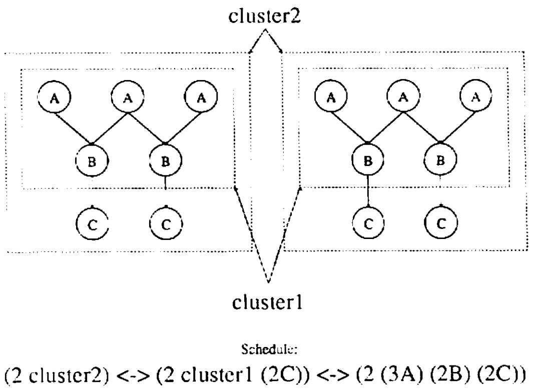


Fig. 33. This cluster hierarchy results from applying the base node and adjacent node scheme to the APG of figure 32. Observe that the looped schedule does not exhibit the full amount of nesting which can be obtained from this example.

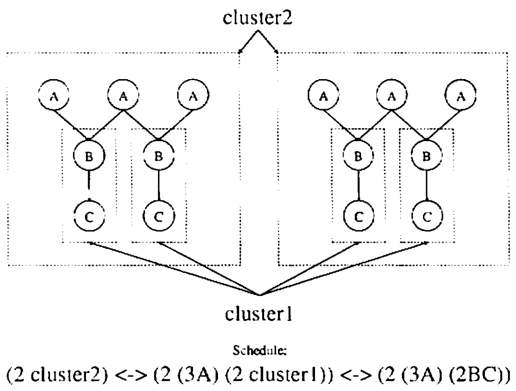


Fig. 34. This figure shows the cluster hierarchy obtained from selecting the highest frequency clusters for the example of figure 32. The resulting schedule fully exploits the nested looping suggested by this graph.

Policy 3. Let S be the set of mutually adjacent pairs of nodes which have not yet been selected as candidate clusters. Then we choose as the candidate cluster that member P of S which maximizes $v(P)$.

3.3. Checking for Deadlock

Once we have selected a candidate cluster C, we must verify that the formation of C does not result in a deadlocked clustered graph. One approach is to form the cluster C and attempt to schedule the resulting graph. Lee and Messerschmitt [12] show that for a certain class of scheduling algorithms successful completion guarantees that a period schedule exists, and hence that the graph is not deadlocked. We could thus choose one such scheduling algorithm, and check that it indeed runs to completion immediately after the formation of C. If instead, it reaches a point when no nodes are fireable, then we must abort the consideration of C as a cluster.

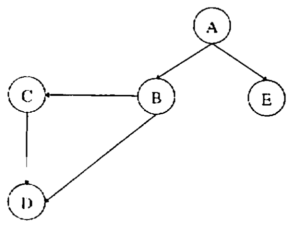
Since we must check for deadlock after the selection of every candidate cluster which subsumes a source node or a delay, this approach will be extremely time consuming. In this subsection, we propose an alternative method which verifies the feasibility of a candidate cluster by checking whether or not its formation introduces a cycle in the APG. Note that we define the PGAN algorithm as a process of repeatedly selecting base nodes and adjacent nodes, and consolidating them whenever deadlocks don't result. The specific method for deadlock detection is an implementation issue, and our method of checking for cycles in the APG requires an implementation in which PGAN clustering decisions are carried out on a hierarchically maintained APG rather than an SDF graph.

The following definitions are fundamental to developing our scheme for efficient deadlock detection using the APG:

Definition. A *path* in an APG G is a finite sequence p of arcs $a_1, a_2, \ldots, a_n$, such that the source of a_{i+1} is the sink of a_i, for $i \in \{1, \ldots, n-1\}$. We say that p is a path from the source node of a_1 to the sink node of a_n. If x and y are two nodes in the same APG G then we define the expression "$x \rightarrow y$" to have value 1 if there is a path in G from x to y and 0 otherwise.

Definition. Given an APG K, we define a *reachability matrix* for K, as any matrix R which satisfies the following conditions:

(a) The rows and columns of R are both indexed by the nodes of K.
(b) If A and B are two nodes in K, then the entry $R[A, B]$ is 1 if there is a path from A to B, and 0 otherwise. Thus, $R[A, B] = A \rightarrow B$, and every diagonal element of R is 0. Note that since a reachability matrix contains Boolean entries, it can be implemented with a storage cost of only one bit per entry, in principle.



Reachability Matrix

	A	B	C	D	E
A	0	1	1	1	1
B	0	0	1	1	0
C	0	0	0	1	0
D	0	0	0	0	0
E	0	0	0	0	0

Fig. 35. An APG and a reachability matrix for that APG.

In figure 35 we show an APG and its reachability matrix. The diagonal elements of a reachability matrix must all be zero since a nonzero diagonal element exists if and only if there is a cycle in the graph. Thus, we can determine whether or not a cluster C of nodes z_1, $z_2, \ldots, z_N$ in an APG A introduces deadlock by calculating the reachability matrix $\hat{R}$ for the clustered precedence graph $\hat{A}$. If $\hat{R}$ contains any nonzero diagonal elements then the formation of cluster C in A introduces a deadlock. If on the other hand C is found to be a valid cluster, then we retain $\hat{A}$ and $\hat{R}$ respectively as the APG and reachability matrix for the next algorithm iteration.

We show now that the reachability matrix $\hat{R}$ can be computed efficiently from R. Our development requires the following notation:

Notation. Given an APG G, we denote by $N(G)$ the set of nodes in G. Thus $N(\hat{A}) = N(A) - C + \{z\}$, the result of removing from $N(A)$ the nodes in C, and adding the supernode z. Also, given two entries a and b in a reachability matrix, we denote by $a + b$, the logical *or* of the binary quantities a and b, and we denote by ab, the logical *and* of a and b.

Assume without loss of generality that $z_1, \cdots z_N$ correspond to the first N rows and columns of R. Now to compute $\hat{R}$ from R, first observe that if neither node x nor y are in the cluster, then $\hat{R}[x, y] = R[x, y] + \hat{R}[x, z]\hat{R}[z, y]$, which can easily be computed once the row and column corresponding to z have been computed. This operation is depicted by arrow p of figure 36.

Now, if $y \in N(\hat{A})$ and $y \neq z$ then $\hat{R}[y, z] = (y \rightarrow z_1) + (y \rightarrow z_2) + \cdots + (y \rightarrow z_N)$. Thus *or*-ing the N column vectors $R[*, z_i]$ and then discarding the top N entries yields the entire column $\hat{R}[*, z]$ except the diagonal entry $\hat{R}[z, z]$. Similarly *or*-ing the rows $R[z_i, *]$ and discarding the z_i entries yields the nondiagonal elements of the row $\hat{R}[z, *]$. These oper-

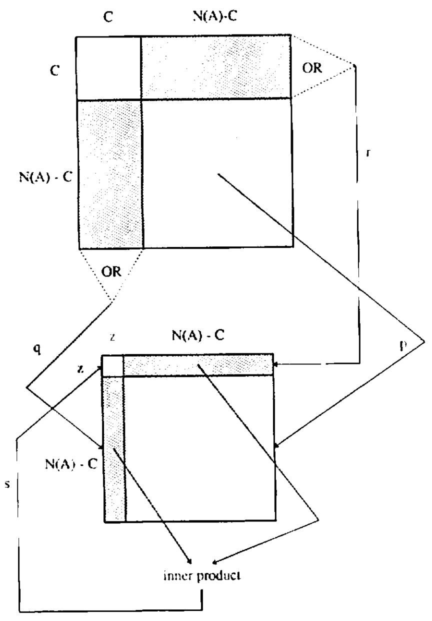


Fig. 36. This figure illustrates the process of updating a reachability matrix to reflect the formation of a cluster. The upper box represents the original matrix; the lower box represents the updated matrix; C represents the set of indices corresponding to the set of nodes in the candidate cluster; $N(A) - C$ represents the indices for nodes excluded from the cluster; and z represents the index for the cluster's supernode. Arrows p, q, r and s each identify the process by which a region of the new matrix is derived.

ations are depicted in arrows q and r respectively in figure 36.

Finally, the diagonal entry $\hat{R}[z, z]$ can be computed from the observation that this path exists if and only if there is a node $p \in N(\hat{A})$, $p \neq z$, such that there is a path from z to p *and* there is a path from p to z. Thus

$$\hat{R}[z, z] = \sum_{p \neq z} (z \rightarrow p)(p \rightarrow z),$$

which is simply an inner product of the vector of nondiagonal elements of $\hat{R}[z, *]$, computed above, with the corresponding column segment $\hat{R}[*, z]$. The computation of $\hat{R}[z, z]$ is shown in arrow s of figure 36. It follows that C introduces a deadlock if and only if the calculation for $\hat{R}[z, z]$ yields "1".

As shown in figure 36, the only computations involved in the calculation for $\hat{R}$ are the elementwise *or*-operations of arrows q and r, the simple update of arrow p and the inner product of arrow s. This method is thus computationally efficient enough to be practical for checking that candidate clusters do not introduce deadlock. The quadratic storage cost, however, may be prohibitive if the APG is large. We are currently investigating techniques to intelligently preprocess large precedence graphs before applying our PGAN algorithm.

3.4. Summary

The preceding development of PGAN is summarized below for an SDF graph G:

1. Create a list L consisting of all pairs P of mutually adjacent nodes in G, sorted in decreasing order of $v(P)$.
2. Loop until L is empty
   (a) Remove the element $P = (A, B)$ at the head of L.
   (b) If the consolidation of A and B into a single node does not introduce a deadlock in G, then:
      (1) Replace A and B with a single node C in G.
      (2) Remove from L all members which contain either A or B.
      (3) For each node Q adjacent to C, compute $v(Q, C)$ and insert the pair (Q, C) into L in a position which preserves the sorted order of L.

3.5. Scheduling the Clustered Graphs

Until now, we have tacitly assumed the availability of a scheduler which can exploit the looping opportunities exposed by PGAN. In this subsection, we briefly discuss our approach to this scheduling problem, and then we discuss the results of combining this approach with PGAN.

After the PGAN cluster-building phase, the root graph and the graph for each cluster must be scheduled. We consider in this article scheduling for a single target processor only; scheduling for multiple processors will be deferred. Our scheduling algorithm attempts at each step to find a complete set of invocations—all of the invocations of a node at a given level of the hierarchy—and schedules such a set as a schedule loop. All of the invocations of a node A are fired in succession if all of the invocations' inputs are available and A does not have a successor which can have all of its invocations fired one after the other. If a complete set cannot be found, a node which has no fireable successor is chosen to be fired, and this selection is performed in such a way that no node is scheduled twice before all other nodes have been tried.

The check for fireable successors in the SDF graph must detect the possible presence of a directed loop in which all of the nodes are fireable. Since such a loop will never yield a fireable node without fireable successors, we arbitrarily select one of the nodes in it to schedule. The scheduling policy outlined above, and the tendency of the cluster-building process to favor nested loops, are our mechanisms for carrying out the scheduling objectives defined in Section 2.

4. Experimental Results

We have implemented PGAN in *Ptolemy*, a heterogeneous platform for software-prototyping [13]. We have found over a large range of examples that the resulting schedules apply at least as much looping as the schedules that are obtained from the CSUF method. The degree of improvement depends on the proportion of sample-rate changes in the graph. As discussed in Section 4, CSUF scheduling does not consider looping opportunities which span sample-rate boundaries. Since PGAN uniformly considers nodes of differing frequency, it does not suffer from the same problem. In figure 37 and figure 38 we depict the APG, and the clustering sequences and schedules which result from

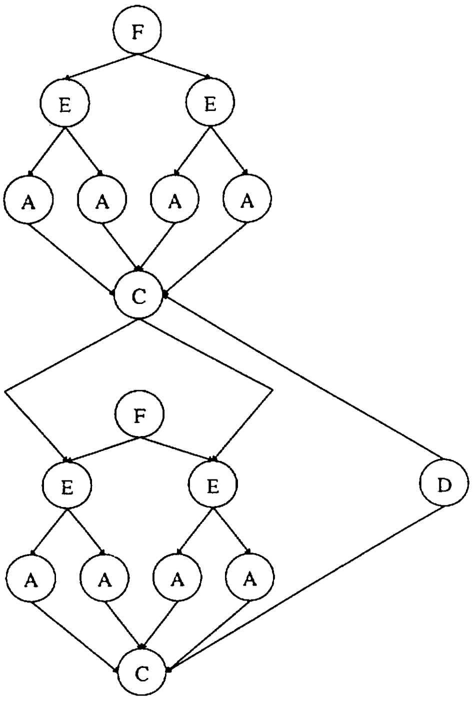
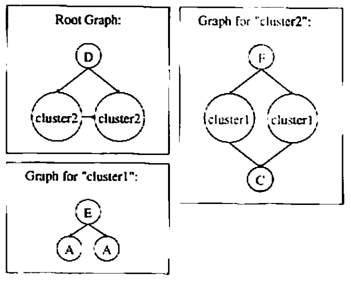


Fig. 37. The APG for the graph of figure 13, which was used to show the inability of CSUF to detect looping oportunities which occur across sample rate boundaries. We return to this example to illustrate how PGAN succeeds in detecting these opportunities.



PGAN Schedule:

D(2 cluster2) <-> D(2 F (2 cluster1) C) <-> D(2F(2E(2A))C)

vs.

The CSUF Schedule:

F(2E(2A))FDC(2E(2A))C

Fig. 38. The PGAN clustering sequence and the resulting schedule for the APG of figure 37. The CSUF schedule is given also, for comparison.

applying PGAN to the graph of figure 13—the example that was used to illustrate the inability of CSUF to handle sample-rate changes. We juxtapose the less efficient CSUF schedule for comparison.

PGAN's incremental approach to avoiding deadlocks is illustrated in figure 39 through figure 41, with the same example that was used to discuss the deadlock problem of CSUF. Observe that at each clustering step, invocations of *C* can never be considered for consolidation, since doing so would introduce a cycle in the APG. As a result, *C* is not represented in any cluster, and the hierarchical structure reflects the desired partition of figure 12. As expected, our recursive scheduling procedure yields the optimal schedule.

The graph of figure 16 also contains looping opportunities that span sample-rate boundaries. The PGAN schedule given in figure 19 contains only one code-segment per node. The CSUF schedule, shown in figure 42 is much less efficient.

5. Conclusions

An evolution of algorithms for extracting looping information from SDF graphs has been presented. The

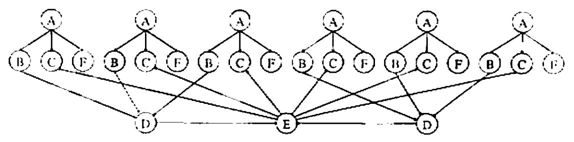


Fig. 39. The APG for the graph of figure 11, which was used to illustrate the problem of partitioning CSUFs which introduce deadlocks. We return to this example to demonstrate that PGAN's incremental approach to cluster-building avoids the partitioning problem.

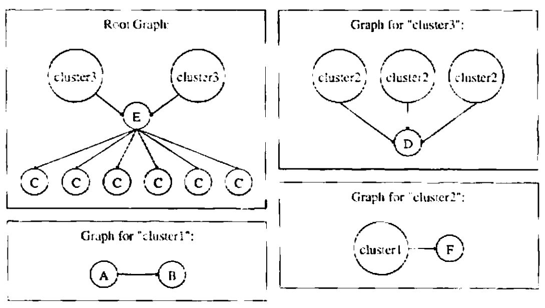


Fig. 40. The PGAN clustering sequence for the APG of figure 39.

Schedule:

(2 cluster3) E (6C) <->

(2 (3 cluster2) D) E (6C) <->

(2 (3 cluster1 F) D) E (6C) <->

(2(3ABF)D)E(6C)

Fig. 41. The schedule which results from the organization in figure 40.

(2DA(2(3EGF)H))A(2(3EGF)H)BC

Fig. 42. The CSUF schedule for the example of figure 16.

first method—postprocessing a minimum buffer-length scheduler with a pattern-matcher—shows that scheduling decisions must be driven by looping considerations in order to effectively exploit opportunities for looping. The method of isolating connected subgraphs of uniform frequency (CSUF), and scheduling them as indivisible units, was our first attempt at recognizing repetitive firing patterns, during the scheduling phase. This technique exhibited a dramatic improvement over our first method. Two limitations surfaced, however—the problem of partitioning deadlocked clustered graphs, and the more significant problem of not being able to recognize looping which spans sample-rate boundaries.

These limitations were overcome by our third approach, Pairwise Grouping of Adjacent Nodes (PGAN). The technique has been implemented within *Ptolemy*, a heterogeneous platform for software-prototyping [13], and preliminary results confirm that this approach exploits opportunities for looping more effectively than its predecessors.

This article has also highlighted many directions for future research. These problems include more complete consideration of scheduling trade-offs, further examining the interaction between scheduling and code-generation, and extending our work to the multiprocessor case.

Notes

1. Neglecting the overhead due to each loop.
2. This example is taken from [8].
3. For unity blocking factor [1]. This discussion can easily be generalized to consider arbitrary blocking factors, but we refrain from doing so for clarity.

References

1. E.A. Lee and David G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE,* September 1987.
2. E.A. Lee, W.-H. Ho, E. Goei, J. Bier, and S.S. Bhattacharyya, "Gabriel: a design environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* vol. 37(11), 1989, pp. 1751-1762.
3. 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 15, 1991. PhD Thesis.
4. P.N. Hilfinger, *Silage References Manual, Draft Release 2.0,* Computer Science Division, EECS Dept., University of California at Berkeley, July 8, 1989.
5. J.B. Dennis, "First version of a dataflow procedure language," *MIT/LCS/TM-61.* Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge, MA 02139.
6. E.A. Lee, "Programmable DSP architectures: part I," *IEEE ASSP Magazine,* vol. 5(4), October, 1988, pp. 4-19.
7. E.A. Lee, "Programmable DSP architectures: part II," *IEEE ASSP Magazine,* vol. 6(1), January, 1989, pp. 4-14.
8. S. How, "Code Generation for Multirate DSP Systems in GABRIEL," Master's Degree Report, U.C. Berkeley, May, 1990.
9. W.-H. Ho, Edward A. Lee, and D.G. Messerschmitt, "High level dataflow programming for digital signal processing," *VLSI Signal Processing III,* IEEE Press, 1988.
10. W.-H. Ho, "Code Generation for Digital Signal Processors Using Synchronous Dataflow," Master's Degree Report, U.C. Berkeley, May, 1988.
11. S.S. Bhattacharyya, "Clustering Formalism For Synchronous Dataflow," Technical Report UCB/ERL M92/30, U.C. Berkeley, Berkeley, CA 94720, April, 1992.
12. E.A. Lee and D.G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Transactions on Computers* vol. C-36(2), 1987, pp. 24-35.
13. S. Ha, J. Buck, E.A. Lee, and D.G. Messerschmitt, "PTOLEMY: A Platform for Heterogeneous Simulation and Prototyping," European Simulation Conference, June, 1991.

Shuvra S. Bhattacharyya received the B.S. degree in Electrical and Computer Engineering from the University of Wisconsin-Madison in 1987, and the M.S. degree in Electrical Engineering from the University of California-Berkeley in 1991. From 1991 to 1992, he was employed by Kuck and Associates in Champaign, IL, where he designed and implemented optimizing program transformations for C and Fortran compilers. Currently, he is pursuing the Ph.D. in Electrical Engineering at U.C. Berkeley. Mr. Bhattacharyya is a member of IEEE and ACM.

Edward A. Lee is an associate professor in the Electrical Engineering and Computer Science Department at U.C. Berkeley. His research activities include parallel computations, architecture and software techniques for programmable DSPs, design environments for development of real-time software, and digital communication. He was a recipient of a 1987 NSF Presidential Young Investigator award, an IBM faculty development award, the 1986 Sakrison prize at U.C. Berkeley for the best thesis in Electrical Engineering, and a paper award from the IEEE Signal Processing Society. He is co-author of "Digital Communication," with D.G. Messerschmitt, Kluwer Academic Press, 1988, and "Digital Signal Processing Experiments" with Alan Kamas, Prentice Hall, 1989, as well as numerous technical papers. His B.S. degree is from Yale University (1979), his masters (S.M.) from MIT (1981), and his PhD from U.C. Berkeley (1986). From 1979 to 1982 he was a member of technical staff at Bell Telephone Laboratories in Holmdel, New Jersey, in the Advanced Data Communications Laboratory, where he did extensive work with early programmable DSPs, and exploratory work in voiceband data modem techniques and simultaneous voice and data transmission. He is chairman of the VLSI Technical Committee of the Signal Processing Society, co-program chair of the 1992 Application Specific Array Processor Conference, and on the editorial board of the *Journal of VLSI Signal Processing.*