SCHEDULING SYNCHRONOUS DATAFLOW GRAPHS FOR EFFICIENT ITERATION

by

Shuvra S. Bhattacharyya

Memorandum No. UCB/ERL M91/65

16 July 1991

SCHEDULING SYNCHRONOUS DATAFLOW GRAPHS FOR EFFICIENT ITERATION

by

Shuvra S. Bhattacharyya

Memorandum No. UCB/ERL M91/65

16 July 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SCHEDULING SYNCHRONOUS DATAFLOW GRAPHS FOR EFFICIENT ITERATION

by

Shuvra S. Bhattacharyya

Memorandum No. UCB/ERL M91/65

16 July 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SCHEDULING SYNCHRONOUS DATAFLOW GRAPHS FOR EFFICIENT ITERATION

Shuvra S. Bhattacharyya
Department of EECS
University of California-Berkeley
July 1991

ABSTRACT

This report develops a compile-time algorithm for scheduling synchronous dataflow (SDF) graphs in a manner that exploits opportunities for *looping* — the successive reoccurrence of identical firing patterns. Looping is highly desirable in static scheduling, since it allows sections of the target code to be executed within loop constructs, such as "do-while", and hence it provides reductions in program-memory requirements. The benefits are particularly pronounced in digital signal processing (DSP) applications, which often have many opportunities for looping inherent in them. Preliminary results of applying these loop-extraction algorithms show orders of magnitude of compaction for target program code space.

1. INTRODUCTION

Among the advantages of using synchronous data flow (SDF) programming [1] to develop digital signal processing (DSP) systems, is the facility for efficient compilation. The process, depicted in figure 1, involves maintaining a library of pre-defined code segments corresponding to each possible type of actor. A graph is compiled by first determining an 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 (APEG), before beginning the compilation process.

This report focuses on the first stage of this process, *scheduling* the graph. We assume a uniprocessor target architecture with a Harvard-style memory organization, or one of its variants [2, 3], and we attempt to generate execution orderings which make efficient use of the data and program memory spaces. We restrict our domain to single processors so that we can focus on exploring the fundamental constraints which SDF graphs impose on buffering and code space requirements. We expect however, that the techniques developed in this report can be extended to the multiprocessor case.

Programmable DSP's 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 is usually significantly slower. It is thus highly desirable, and sometimes necessary — if the external-memory option is not available, or economically feasible — to

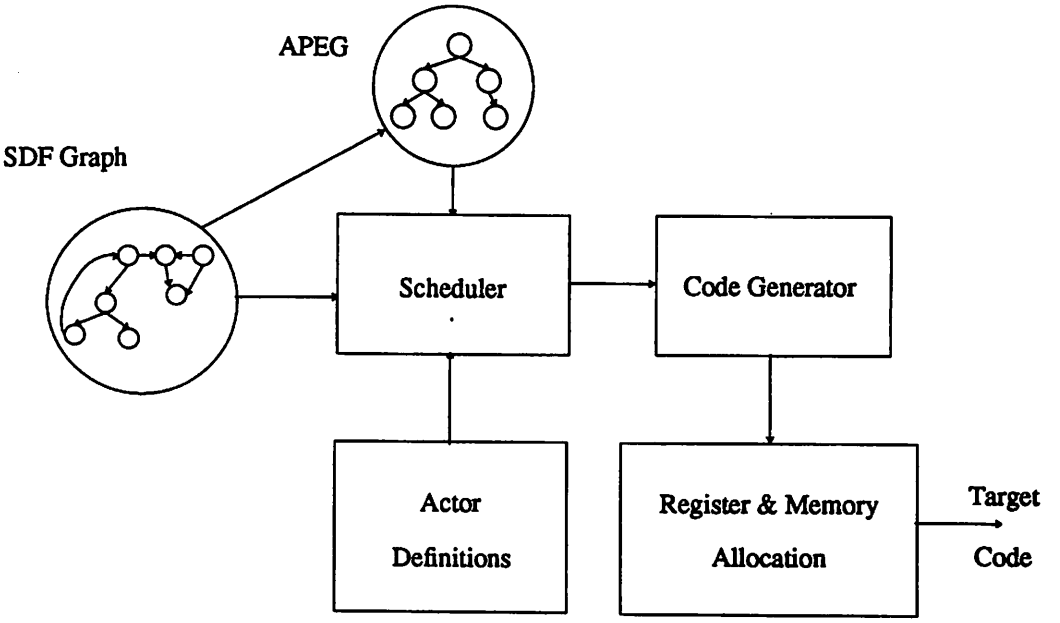
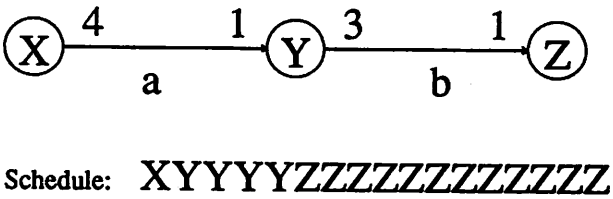


Figure 1. Compiling a synchronous dataflow graph.

have the code and data for a program fit entirely within the on-chip memory spaces.

The uniprocessor scheduling issues are illustrated in figure 2 and table 1. Figure 2 presents an SDF graph and a uniprocessor schedule for that graph, and table 1 presents 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* of a node in the SDF graph. The invocations of a node X are labeled $X_1, X_2, ..., X_n$, 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 which is required to support the schedule.



Schedule: XYYYYZZZZZZZZZZZ

Figure 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.

Arc	X_1	Y_1	Y_2	Y_3	Y_4	Z_1	Z_1	Z_2	Z_3	Z_4	Z_6
a	4	3	2	1	0	0	0	0	0	0	0
b	0	3	6	9	12	11	10	9	8	7	6
Total	4	6	8	10	12	11	10	9	8	7	6

Arc	Z_7	Z_8	Z_9	Z_{10}	Z_{11}	Z_{12}
a	0	0	0	0	0	0
b	5	4	3	2	1	0
Total	5	4	3	2	1	0

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

Figure 3 shows another schedule for the same graph, and table 2 shows the associated buffer activity profile. Examination of the *total* row reveals that the memory requirements 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.

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* $(N\ a_1, a_2, \cdots, a_n)$*, where each* a_i *represents either a node in the graph, or a sub-schedule in looped form, and* $(N\ a_1, a_2, \cdots, a_n)$ *represents the successive repetition N times, of the firing sequence* $a_1, a_2, \ldots, a_n$*. For example,* ABBABB *can be expressed in looped form, as both* (2ABB) *and* (2A(2B))*. We call a schedule expressed in looped form a*

XYZZZYZZZYZZZYZZZ

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

Arc	X_1	Y_1	Y_2	Y_3	Y_4	Z_1	Z_1	Z_2	Z_3	Z_4	Z_6
a	4	3	3	3	3	2	2	2	2	1	1
b	0	3	2	1	0	3	2	1	0	3	2
Total	4	6	5	4	3	5	4	3	2	4	3

Arc	Z_7	Z_8	Z_9	Z_{10}	Z_{11}	Z_{12}
a	1	1	0	0	0	0
b	1	0	3	2	1	0
Total	2	1	3	2	1	0

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

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, to produce code which contains a loop to implement each schedule loop. For instance, if the target language is "C", then our example would produce an output listing with the organization in example 1 below.

If given any actor N, we let $S(N)$ denote the size in machine instructions, of the in-line code segment for N, 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 without considering looping, 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.

This report presents an evolution of scheduling heuristics for detecting and exploiting opportunities for creating looping within a schedule. These opportunities are a common consequence of the application of *iteration*, which in SDF, is defined [4] as the increase in invocation rate which results from an upsampling along an arc. Section 2 formalizes our

Example 1

```
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
                ...
                ...
        }
}
```

[1] neglecting the overhead due to each loop

approach to code generation for loops which implement looped schedules. Section 3 details our scheduling objectives and assesses the related tradeoffs. Section 4 briefly presents our preliminary approach to the uniprocessor scheduling problem, and illustrates the grave limitations of this approach. The limitations arise primarily from viewing looping considerations as a post-optimization phase, rather than having them drive the scheduling process. Section 5 presents a much more effective method based on preprocessing the graph, to isolate regions for which it is likely that schedule loops can be constructed. After illustrating the benefits of this technique, the section presents two shortcomings which prevent it from being a general solution. The next section presents a third approach which remedies these shortcomings. Finally, section 7 is devoted to concluding remarks.

2. CODE GENERATION ISSUES

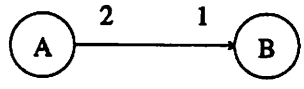
Since this report focuses on exploiting opportunities for looping, it will be useful to first examine the code-generation aspects of implementing schedule loops as loops in the target code. Unless otherwise stated, we assume that the target language is assembler source for programmable DSP's.

The primary code-generation issue for looped schedules is the accessing of buffers which implement the passage of data along arcs, from within loops. The difficulty lies in the requirement for different invocations of the same actor to have the same code. As a simple example, consider the graph and schedule in figure 4. Both invocations B_1 and B_2 must access their inputs with the same instruction. This requires that the output data for A be stored in manner which can be accessed iteratively. This in turn, suggests writing the data produced by A to successive memory locations, and having B read this data using the register autoincrement addressing mode — an addressing mode which was designed precisely for this purpose of iteratively stepping through successive items of data. Code to implement figure 4 would then have the structure outlined in figure figure 5.

This method presents two main considerations. The first consideration is that it requires a register allocator to resolve situations when the number of address registers required in a loop exceeds the number available in the target machine. The register allocator would be responsible for deciding which register to swap out at a given conflict point, and for generating the appropriate "spill code"to save and restore each swapped register's contents. In the remainder of this report, we assume that such a register allocator is available.

The second consideration is the mechanics of maintaining buffers which are accessed in a loop. It can be presented as the following question :

Question: *When is it necessary to buffer data for an arc in* successive *memory locations, and*



Schedule: A(2B)

Figure 4. A simple example of a looped schedule, which we use to introduce the difficulties in generating code for loops.

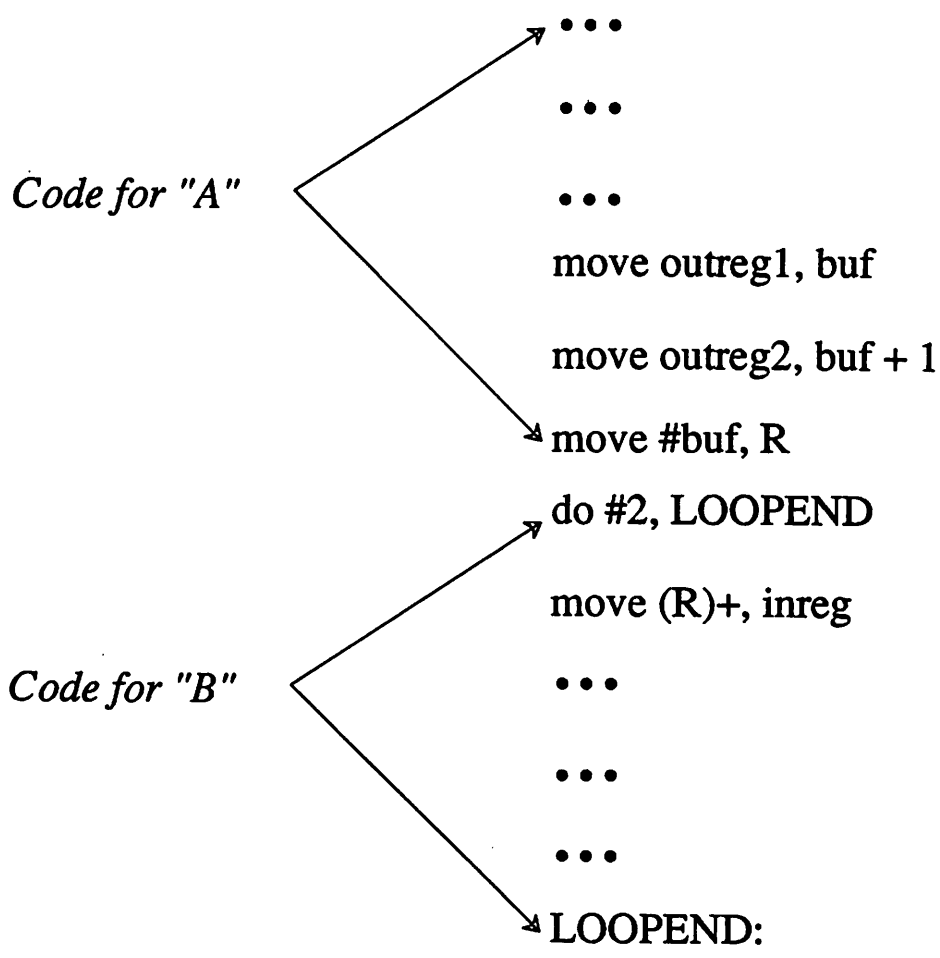


Figure 5. An outline of the code to implement the example of figure 4. Here "inreg", "outreg1" and "outreg2" represent data registers; "R" represents an address register; and "buf" represents an absolute data memory address. The statement "do #N, LABEL" specifies the successive execution N times, of the block of code between the "do" statement and the instruction at location LABEL. The syntax used in this illustration is borrowed from the assembly language for the *Motorola DSP56000*.

what is the procedure for generating accesses to such contiguous buffers?

This section is devoted primarily to answering the above question.

2.1. Modulo Addressing

Most programmable DSP's offer a *modulo addressing mode* which can be used in conjunction with careful buffer sizing, to alleviate the memory cost associated with requiring buffer accesses to be sequential. This addressing mode allows for efficient implementation of circular buffers, for which indices need to be computed modulo the length of the buffer. We illustrate modulo addressing with an example.

Example 2: In the *Motorola DSP56000* programmable DSP, a modifier register M_x is

associated with each address register R_x. Loading M_x with a value $n > 0$ specifies a circular buffer of length $n + 1$. The starting address of the buffer is determined by the value V_x, stored in R_x, used to access the buffer. If we let B denote the value obtained by clearing the $\lceil \log_2 (n + 1) \rceil$ least significant bits of R_x, then assuming that $B \leq V_x \leq B + n$, an autoincrement access $(R_x)+$ causes R_x to be updated to contain $B + (V_x - B + 1) \bmod (n + 1)$.

Figure 6 illustrates the use of modulo addressing to decrease memory requirements when sequential buffer access is needed. The schedule UUVUV would clearly require a buffer size of 6 for iterative access if only linear addressing is available. The sequence of buffer diagrams in figure 6 shows how only four buffer locations are required when modulo addressing is used. W and R respectively denote the write pointer for U, and the read pointer for V, and a "." inside a buffer slot indicates a **live sample** — a sample which has been produced, but not yet consumed. Note that the accesses of the second invocation of U and the second invocation of V wrap around the end of the buffer.

Observe also, that the pointers R and W can be reset at the beginning of each schedule period, to point to the beginning of the buffer, and thus the access patterns depicted in figure 6 could be repeated every period. This would cause the locations in each buffer's access to be *static* — fixed for every iteration of the periodic schedule — and hence they would be known values at compile-time.

This illustration renders false the previous notion that for static buffering, the total number of samples exchanged on an arc, per schedule period, must always be a multiple of the buffer size. In actuality, the requirement holds only when there is a nonzero delay associated with the arc in question.

Before pursuing buffer size considerations any further, we develop in detail, the concept of a buffer and the methods for implementing buffers during code generation.

2.2. Buffers

The following definition makes precise the notion of a buffer, which we have been using informally until now.

Definition: *Let* α *be an arc in an SDF graph* G, *directed from a source node A to a sink node B. Suppose* m *samples are exchanged through* α *during each schedule period, and let* $a_{1j}, a_{2j}, \cdots, a_{mj}$ *denote respectively the first through* m*th samples produced by A to* α *during schedule period j. Similarly, let* $b_{1j}, \cdots, b_{mj}$ *successively denote the samples consumed by B from* α *during schedule period j.*[2] *Then, assuming that code has been generated to implement* G, *a* **buffer** *for* α *is a sequence* $D = \{d_1, d_2, \cdots, d_N\}$, *of successive memory locations such that the following two conditions hold* **for all** *j:*

(1) $\forall\, i \in \{1, \ldots, m\}$,
  (a) a_{ij} *is written to some* $x_{ij} \in D$; *and*
  (b) b_{ij} *is read from some* $y_{ij} \in D$;

(2) $\forall\, i \in \{1, \ldots, N\}$, *at least one of the following three conditions hold:*
  (a) $\exists\, k$ *such that* $x_{kj} = d_i$; *or*

---

[2] Observe that $\{\forall$ i and j, $a_{ij} = b_{ij}\}$ if and only if there is no delay on α.

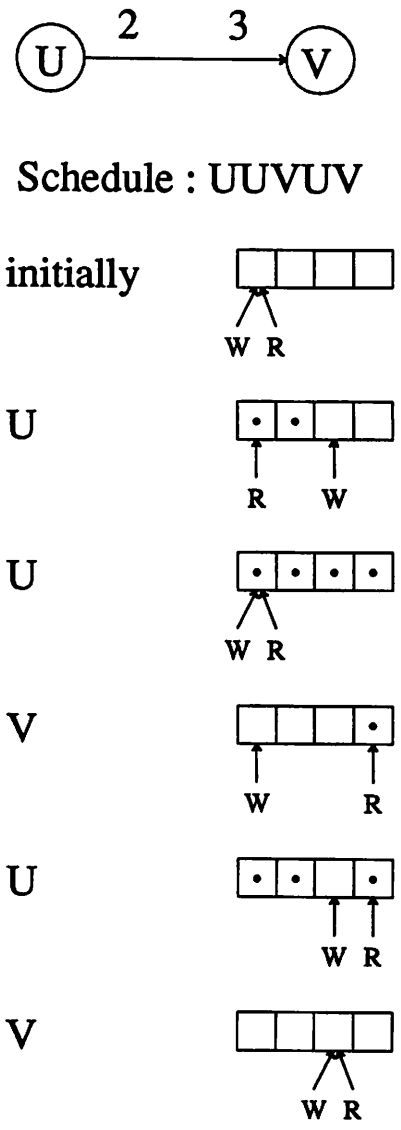


Figure 6. An illustration of modulo addressing. "W" and "R" respectively represent the write pointer for U and the read pointer for R.

(b) $\exists k$ *such that* $y_{k,j} = d_i$; *or*
(c) d_i *contains a single live sample throughout schedule period* j.

Thus a buffer B for α is a set of memory locations such that the production/consumption of a token to/from α corresponds to an access of B, and each element of B is accessed at least once per schedule period unless it contains a live sample throughout the entire period.

The following definition presents a classification of different types of buffers which are useful for code generation.

Definition: *A* **static buffer** *is a buffer for which $x_{i,j}$ and $y_{i,j}$ are independent of j. A* **contiguous buffer** *is a buffer* $\{d_1, d_2, ..., d_N\}$ such that the d_i's represent successive memory locations. An **iterative access buffer**, *abbreviated* **IAB**, *is a contiguous buffer* $\{d_1, d_2, ..., d_N\}$ *such that whenever* $1 \le i \le m-1$, $d_k = x_{i,j} \Rightarrow x_{i+1,j} = d_{(k+1) \bmod N}$, *and* $d_l = y_{i,j} \Rightarrow y_{i+1,j} = d_{(l+1) \bmod N}$. *Finally, a* **static, iterative access buffer**, *abbreviated* **SIAB**, *is a static buffer which is an IAB.*

The examples of figure 4 and figure 6 showed that an arc which is accessed from within a loop should have successive samples stored in successive memory locations, possibly in a modulo sense. This suggests that such an arc should be implemented as an SIAB. We introduce the following definition to aid us in summarizing our observations into a policy for code generation.

Definition: *Let G be a graph and let S be a looped schedule for G. We say that a node A of G is* **contained in a loop** *if and only if there are one or more schedule loops containing invocations of A. We say that an arc* α *is* **contained in a loop** *if and only if its source node is contained in loop, or its sink node is contained in a loop, or both.*

With this terminology, we can assert the fundamental policy which we use in code generation for loops:

Policy: *An arc which is contained in a loop will be implemented as an SIAB.*

2.3. Determining the Size of a Buffer

With modulo addressing, it is clear from figure 6, that the minimum number of data memory words required to implement an SIAB for a *delayless* arc is simply the maximum number of live samples which coexist on the arc.

If however, an arc α has a nonzero delay, we must impose the additional constraint that *the total number of samples exchanged along* α *during each schedule period is some positive integral multiple of the buffer length.* The need for this constraint is illustrated in figure 7. Here, the minimum buffer size according to the previous rule is four, since up to four samples can concurrently exist on the arc. Figure 7 shows the succession of buffer states if a buffer of this length is used.

Now since there is a delay on the arc, there will always be a sample in the buffer at the beginning of *each* schedule period — this is the first sample consumed by V_1. For static buffering, we need this *delay sample* — which is consumed in the schedule period *after* it is produced — to reside in the same memory location every period. Comparison of the initial and final buffer states in figure 7 reveals that this is not the case, since the write pointer *W* did not wrap around to point to its original location. Clearly, *W* could have returned to its original position if and only if the number of samples exchanged on the arc was an integer multiple of the buffer length.

We summarize with the following theorem:

Theorem: *For a given schedule, if an arc* α *is to be implemented as an SIAB, then the following conditions must be met by the buffer size N :*

1. N cannot be less than the maximum number of live samples which coexist on α.

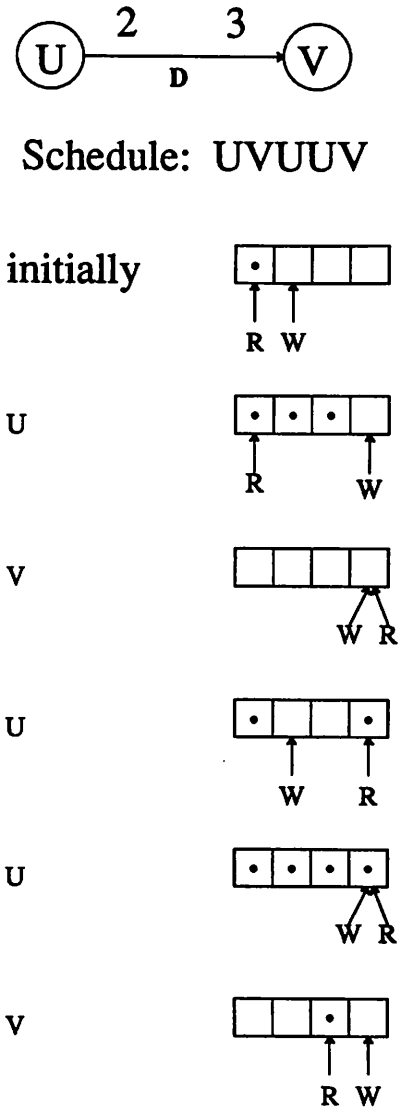


Figure 7. The effect of a delay on the minimum buffer size required for static scheduling. With a buffer size of only 4, the location of the "delay sample" shifts two positions each schedule period.

2. If α has nonzero delay, then the number of samples exchanged along α during a schedule period must be a positive integer multiple of N.

We denote by s_α, the minimum SIAB size for α given by constraints 1 and 2.

Observe that when an arc is not in a loop, it is by no means necessary to implement it as an SIAB— in fact, data memory is often wasted by doing so. The reason is simple: such a policy places unnecessary constraints on memory allocation, and such constraints can only do harm to the efficiency of the allocation. However, this report will not elaborate on the topic of

optimizing data memory allocation; it is a topic which our research has not yet explored in detail. Instead, we make the following assumptions throughout the remainder of the report, for calculating the data memory requirements of a schedule:

1. For a looped schedule, all arcs are implemented as SIAB's, occupying their own contiguous segments in memory.

2. The memory requirement for a schedule which is not a looped schedule is that obtained by applying the following fact, which is presented here without proof :

Fact: A schedule which is not a looped schedule can be implemented with a data memory cost given by

$$X + \sum_{\alpha \in \Delta} s_\alpha$$

where Δ denotes the set of arcs with nonzero delay, and X denotes the *maximum number of live samples which coexist on all arcs which have no delay*. Note that if the original graph has no delays, then the above expression reduces to the maximum number of concurrent live samples which can exist in the entire graph.

This is the policy we used in discussing figure 2 and figure 3, and we will continue to use this policy throughout the remainder of the report.

2.4. The Mechanics of Code Generation

This subsection describes the mechanics for generating code for a loop obtained from a looped schedule. The primary issue is determining the addresses for instructions which access arcs that are contained in loops. State variable accesses pose no special problems, since the state variable storage for an actor is fixed across all invocations. The addressing information required for accessing inputs and outputs can efficiently be generated by maintaining a data structure which we call the *buffer profile* of a looped schedule.

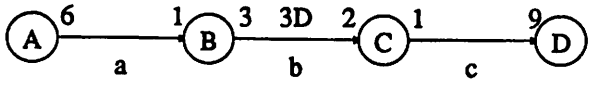
Figure 8 and table 3 illustrate the methods presented in this subsection. Figure 8 presents a multirate SDF graph and a looped schedule for this graph, and table 3 represents the buffer profile for this schedule. Assuming that each arc is implemented as a static buffer, the buffer profile gives the following data for each input and output access of every invocation:

1. The offset into the buffer at which the invocation accesses the arc.

2. Whether or not the buffer access wraps around the end of the buffer, and then performs one or more accesses from the beginning of the buffer. Such an access would require that modulo registers be set up beforehand.

The data structure is created by simulating the buffer accesses as the schedule is traversed. After the simulation, we can create a *buffer length* data structure containing the length of each buffer. We can then perform memory allocation, to obtain a *buffer address* list, containing the starting address of each buffer.

With these data structures — *buffer profile*, *buffer length*, and *buffer address* — we can easily generate the absolute address for any arc I/O access which does not occur within a



Schedule : CA(2B(2C)BC)BCCBD

Figure 8. A multirate graph with a looped schedule.

	invocation	offset	modulo
A : output	1	0	no
B : input	1	0	no
	2	1	no
	3	2	no
	4	3	no
	5	4	no
	6	5	no
B : output	1	3	no
	2	0	no
	3	3	no
	4	0	no
	5	3	no
	6	0	no
C : input	1	0	no
	2	2	no
	3	4	no
	4	0	no
	5	2	no
	6	4	no
	7	0	no
	8	2	no
	9	4	no
C : output	1	0	no
	2	1	no
	3	2	no
	4	3	no
	5	4	no
	6	5	no
	7	6	no
	8	7	no
	9	8	no
D : input	1	0	no

Table 3. The buffer profile for the schedule of figure 8.

loop. As figure 5 showed however, absolute addressing cannot be used in general, within loops.

2.4.1. Common Code Space Sets

Loops in the target code provide savings in code space by allowing different invocations of the same actor to map to the same code segment. For instance, returning to example 1, we see that a single code segment for Y executes the four invocations Y_1, Y_2, Y_3 and Y_4, and similarly 12 invocations of Z are implemented with a single code block for Z. We summarize with the following definition, which will be useful in illustrating our approach to address generation within loops.

Definition: *Given an SDF graph* G *and a looped schedule for that graph, let A be a node in* G *which is contained in a loop. Then a* **common code space set,** *abbreviated* **CCSS,** *of A, is a maximal set of invocations of A, which map to the same code segment. We say that a CCSS is* **nontrivial** *if and only if it contains more than one element. Note that a node in an SDF graph may have more than one CCSS.*

From the schedule of figure 8, we see that A has no nontrivial CCSS's; B has two: $\{B_1, B_3\}$ and $\{B_2, B_4\}$; and C also has two : $\{C_2, C_3, C_5, C_6\}$ and $\{C_4, C_7\}$. These findings are summarized in table 4. From the buffer profile of table 3 and these CCSS profiles, we can generate addresses for all arc I/O accesses.

For example, we see that the members of the CCSS $\{B_1, B_3\}$ access arc b with different offsets. This requires that all read accesses to b within the loop containing $\{B_1, B_3\}$, be carried out with an autoincremented address register R. R can be loaded before entering the loop, and if it must be swapped out at any point, it can be saved to a designated memory location $B_{readpointer}$. Code to restore $B_{readpointer}$ into R must then be inserted prior to any subsequent CCSS region for B within the loop. This readpointer thus functions as a state variable for B through the duration of the loop.

It is not always the case however, that autoincrement addressing must be used to access arcs within a loop. Examination of table 3 reveals that the members of CCSS $\{B_1, B_3\}$ write to arc c from the same offset (3), so absolute addresses *buf* 2+3 *buf* 2+4, and *buf* 2+5 can be used to write the three samples. Absolute addressing can also be used for the CCSS $\{B_2, B_4\}$'s write accesses of c, and the CCSS $\{C_4, C_7\}$'s read accesses of c as well.

$\{B_1, B_3\}$

$\{B_2, B_4\}$

$\{C_2, C_4, C_5, C_6\}$

$\{C_4, C_7\}$

Table 4. The nontrivial CCSS's for the schedule of figure 8.

2.4.2. A Policy for Code Generation

The preceding discussion has developed our policy for generating addresses to access an arc which is contained in a loop:

Policy: *Suppose we are given an SDF graph* G *and a looped schedule for* G. *Let X be a node in* G, *and suppose there is an arc* α *leaving(entering) X. Suppose also that* Γ *is a CCSS of X. Then the code segment for* Γ *can write to (read from)* α *through absolute addresses if and only if all elements of* Γ *write to (read from)* α *at the same offset. If absolute addressing cannot be used, we use register autoincrementing, possibly in modulo-mode.*

This policy uses absolute addressing whenever it can. In practice however, it is often more efficient to use a data register or indirection through an address register (an absolute address typically resides in a separate instruction word, resulting in a slower instruction). Theoretically, a data register can always be used in place of absolute addressing, but this is efficient only if the sample stored in the register is read before the register is required by another computation. Also, if an invocation consumes several samples from a single arc, it would be much more efficient to read those samples with an autoincrement address register even if the locations of the samples are known at compile-time. Our future research will carefully examine such code-optimization issues, but for the present, we adhere to the policy asserted above, since it provides a clear and fully-developed framework for making our scheduling examples concrete.

3. OBJECTIVES

An extremely desirable capability for a uniprocessor scheduler is the ability to solve the following 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.*

This problem is extremely difficult. We have therefore chosen a much less ambitious, but nevertheless substantial, guideline for our uniprocessor scheduling efforts, as a starting point for addressing this simultaneous consideration of data memory/program memory/execution time optimization, and as a method for improving the quality of our SDF compiler. This guideline involves, as the first priority, the compaction of code space. Our scheduler is thus driven by the primary goal of exploiting opportunities for looping. Two considerations have led us to adopt this primary goal:

- 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 do not elaborate on considerations such as these, in which the schedule interacts with the efficiency of the generated code. 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 DSP56000/96000* has a "zero-overhead" looping mode which eliminates the loop count overhead, by performing the indexing in hardware.

Figures 9-12 illustrate some of the tradeoffs discussed above. The graph in figure 9 shows the graph of figure 2 with a downsampling of 2 appended.

Figure 10 shows a schedule for figure 9, with a summary of the associated buffering requirements given in table 5. The program memory requirement is $S(X)+S(Y)+S(Z)+S(U)$, the lowest possible code space size for this graph, and the data memory requirement is 28 words.

Figure 11 presents an alternative schedule which exhibits the same minimum code size, but decreases data memory utilization by 43%. The difference is due to the *nesting* of loops in figure 11, and this is traded off by an increase in loop overhead over figure 10. Table 6 outlines the data memory cost for this alternative schedule.

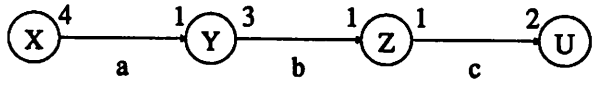


Figure 9. An SDF graph used to illustrate tradeoffs involved in scheduling.

X(4Y)(12Z)(6U)

Figure 10. A looped schedule for the graph of figure 9.

Arc	Buffer Length
a	4
b	12
c	12
Total	28

Table 5. A summary of the buffering requirements for the schedule of figure 10.

X(2(2Y(3Z))3U)

Figure 11. An alternative looped schedule. Observe that this schedule involves several nested loops.

Arc	Buffer Length
a	4
b	6
c	6
Total	16

Table 6. The buffering requirements for the schedule of figure 11.

If we let s denote the per-loop startup overhead, and we assume "zero-overhead" indexing, then the total overhead for each schedule is shown in table 7. While verifying the calculation for table 7, observe that a loop startup overhead is incurred for every initiation of every loop.

Finally, figure 12 presents a third schedule. Looping is not applied within this schedule, and thus we can compute the data memory requirement from the buffer activity profile, which is given in table 8. This requirement is found to be much lower than those of the previous schedules. However, the absence of looping in this schedule results in a significantly larger code space requirement of $S(X) + 4S(Y) + 12S(Z) + 6S(U)$.

With respect to our current scheduling objectives, the schedule of figure 11 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, as is the significant saving of data memory consumption in figure 12.

Our future research directions include investigating more thoroughly, the tradeoffs between the three scheduling objectives of minimizing execution time, and data and program memory requirements.

Schedule	Overhead
X(4Y)(12Z)(6U)	$3s$
X(2(2Y(3Z))3U)	$9s$

Table 7. A comparison of the loop-startup overhead for the schedules of figure 10 and figure 11, assuming that the cost of initiating any loop is s.

XYZZUZYZUZZUYZZUZYZUZZU

Figure 12. A schedule for the graph of figure 9 which does not apply looping.

Arc	X_1	Y_1	Z_1	Z_2	U_1	Z_3	Y_2	Z_4	U_2	Z_5	Z_6
a	4	3	3	3	3	3	2	2	2	2	2
b	0	3	2	1	1	0	3	2	2	1	0
c	0	0	1	2	0	1	1	2	0	1	2
Total	4	6	6	6	4	4	6	6	4	4	4

Arc	Y_4	Z_{10}	U_5	Z_{11}	Z_{12}	U_6
a	0	0	0	0	0	0
b	3	2	2	1	0	0
c	1	2	0	1	2	0
Total	4	4	2	2	2	0

The total data memory requirement is 6 words.

Table 8. The buffer activity profile for the schedule of figure 12.

4. PRIOR WORK

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 and the next.

Our first approach to uniprocessor scheduling was to use a simple heuristic for minimizing data memory requirements [567]. 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 attempted to find successively repeated sequences of firings [8]. The scheduler then grouped such sequences into schedule loops.

Thus in this approach, looping was not at all considered while constructing the ordering of invocations, and as a result, opportunities for creating schedule loops frequently went undetected.

Our previous example in figures 9-12 illustrates very well this conflict between minimum buffer length scheduling and scheduling to maximize looping. Figures 10-11 show two different looped schedules which can be obtained for the graph of figure 9. Figure 12 is the schedule 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. The rest of this report describes our approaches to carrying out this scheduling goal.

5. 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 will demonstrate how such subgraphs can be used to produce looped schedules.

Definition: *Given an SDF graph* $G = \{N, \Lambda\}$, *suppose* $M \in N$, *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 13 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.

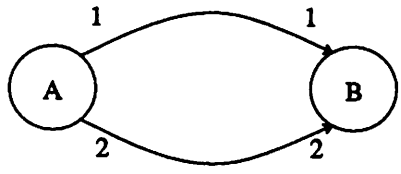


Figure 13. A CSUF with nonuniform $p(\alpha)$.

Whenever it doesn't introduce a delay-free loop in the graph, a CSUF M can be treated by a scheduler as a *supernode*, which 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.*

Figures 14-17 illustrate how partitioning a graph into CSUF's can lead to looped schedules. The encircled regions in figure 14 outline three nontrivial CSUF's — $\{A, B, C\}$, $\{E, F\}$, and $\{H, G\}$. Each of these regions is initially considered by the scheduler as a single unit, as shown in figure 15. The minimum buffer size scheduling heuristic of the previous section, can then be applied to this *clustered* graph. The result is given in figure 16.

The first schedule in figure 16 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 CSUF's. Had these CSUF's not been consolidated as individual units, as in figure 15, the scheduler could have interrupted a repetitive sequence, to invoke a fireable node from some other region of the graph.

For example, figure 17 shows the first several firings of a valid schedule, according to our minimum buffer size heuristic. 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

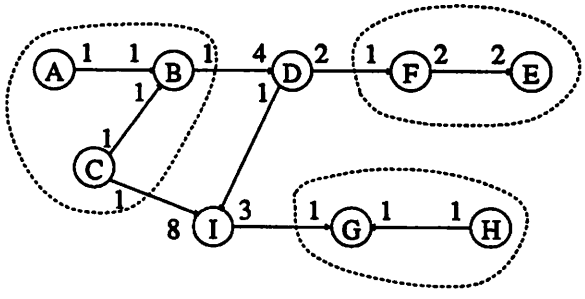


Figure 14. A multirate graph with three CSUF's — {A,B,C}, {E,F}, and {G,H}.

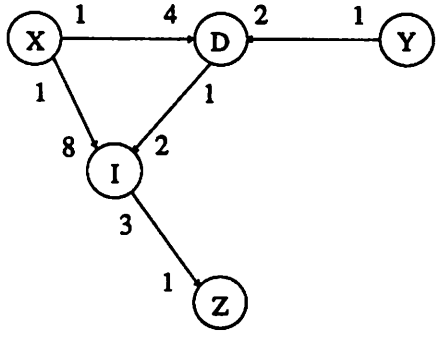


Figure 15. The topology that results from considering each CSUF in figure 14 as a single node. X, Y and Z are used to represent {A,B,C}, {E,F}, and {G,H} respectively.

Schedule for the clustered graph :

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

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

Figure 16. Scheduling the clustered graph of figure 15. 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.

downsampled input of D, goes unexploited. The invocation of H so early in the schedule, also precludes exploiting the upsampled CSUF $\{G, H\}$.

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 limitations, which prevent it from being a general solution.

The first shortcoming is illustrated in figure 18.[3] 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 maintaining the deadlocked state. The desired partition is shown in figure 19, 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.

The second shortcoming of the CSUF approach arises from its inability to detect looping which occurs across sample rate changes. Figure 20 depicts a graph with opportunities for this kind of looping, and figure 21 shows a looped schedule for this graph.

Although figure 21 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

ABCHEFACBEFACBACBD.....

Figure 17. The first several firings of a schedule for the graph of figure 14 using the minimum buffer size heuristic described in section 4.

[3]This example is taken from [8].

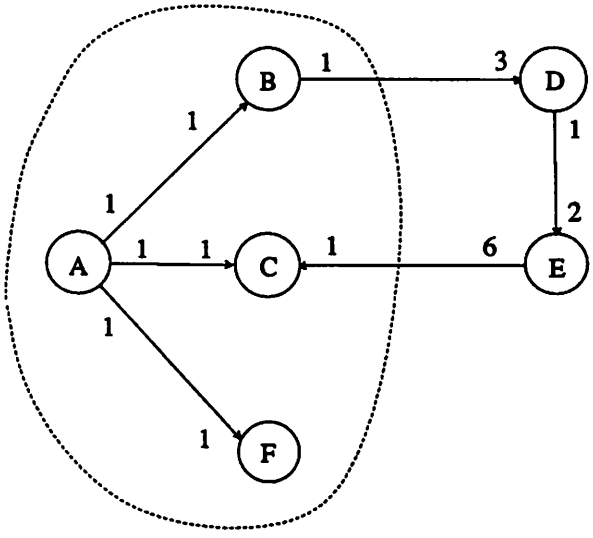
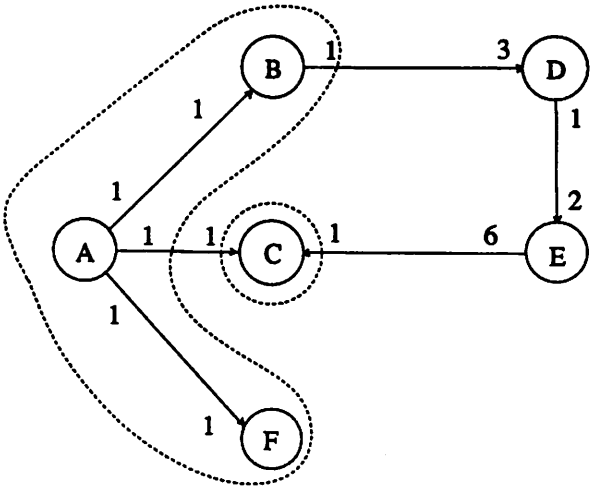


Figure 18. The consolidation of the CSUF {A, B, C, F} introduces a directed delay-free loop.



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

Figure 19. The desired partition of the cluster in figure 18 and the resulting looped schedule.

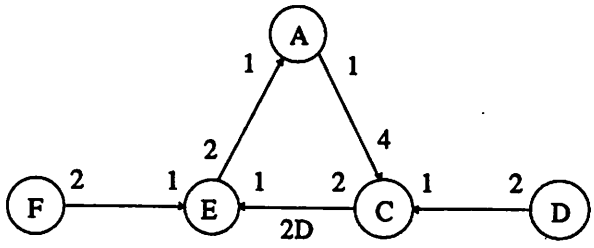


Figure 20. An SDF graph which offers opportunities for looping that span sample rate boundaries.

D(2F(E(2A))BC)

Figure 21. A looped schedule for the graph of figure 20.

the looping results from CSUF's, since every 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 22. Clearly this schedule applies significantly less looping, and requires much more code space, than that of figure 21.

The reason is because 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.

This section has demonstrated that SCCF-based scheduling can produce large improvements in the degree of looping, over scheduling techniques which do not attempt to recognize looping while constructing the schedule. However, we have also shown that two limitations — the possible introduction of deadlocks, and the inability to consider loops which span regions of different sample rate — prevent this method from being a general approach. The remainder of this report presents a generalized version of this technique, which overcomes these limitations, and thereby, extends our ability to extract looping from SDF graphs.

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

Figure 22. The schedule for the graph of figure 20 which is obtained from the CSUF approach. The schedule is much less compact than that of figure 21.

6. PAIRWISE GROUPING 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, whenever possible. As illustrated in figure 3, nesting loops requires less buffering without increasing code-space requirements. Thus, our improved scheduler performs in accordance with the scheduling objectives outlined in section 3.

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 the interpretation of a *cluster* in the SDF graph, is a group of nodes, or subclusters, which 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 for continuous scheduling. However, doing so does not improve our ability to conserve memory consumption, 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, defined in [ref]. 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. We summarize the procedure of applying PGAN to an SDF graph G, with a pseudocode

description:

$G_1 = G$
$i = 1$
loop forever
    Attempt to form a cluster in graph G_i.

    If all clustering opportunities are exhausted
    Then exit from the loop
    Else
        • *Let* C_i *denote the cluster formed in this iteration;*
        • *Set* G_{i+1} *equal to the graph that results from replacing in* G_i *the two nodes of* C_i *with a single node that represents* C_i.
end loop

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 APEG, and to relate these considerations to the SDF graph.

6.1. Examining the APEG for Looping

It is instructive to view the formation of clusters in an SDF graph with respect to the impact on the corresponding APEG. The APEG is particularly illustrative in discussing looping, since it explicitly shows the repetition inherent in an algorithm. Figures 23 and 24 depict respectively a multirate SDF graph and its associated APEG. The optimal looped schedule with respect to the objectives defined in section 3 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 an APEG because looping opportunities are manifested as repeated subgraphs. These subgraphs can be considered as hierarchical clusters in a manner analogous to our interpretation of clusters in the SDF graph. Figure 25, table 9 and figure 26 illustrate this process for the example of figure 23. The series of graphs in figure 25 shows a succession of cluster formations, each involving two or more repeated subgraphs. Table 9 lists looped schedules for the root graph, and each cluster, and figure 26 presents the result of recursively replacing the appearance of each cluster with its subschedule, in the root schedule. This result agrees with the "optimal" schedule.

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

Table 9. The schedule for each of the clusters depicted in figure 25.

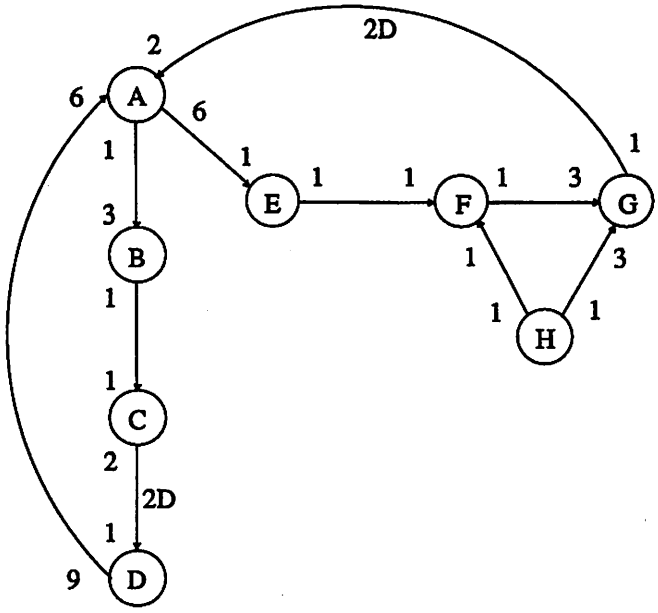


Figure 23. A multirate SDF graph which offers several opportunities for looping.

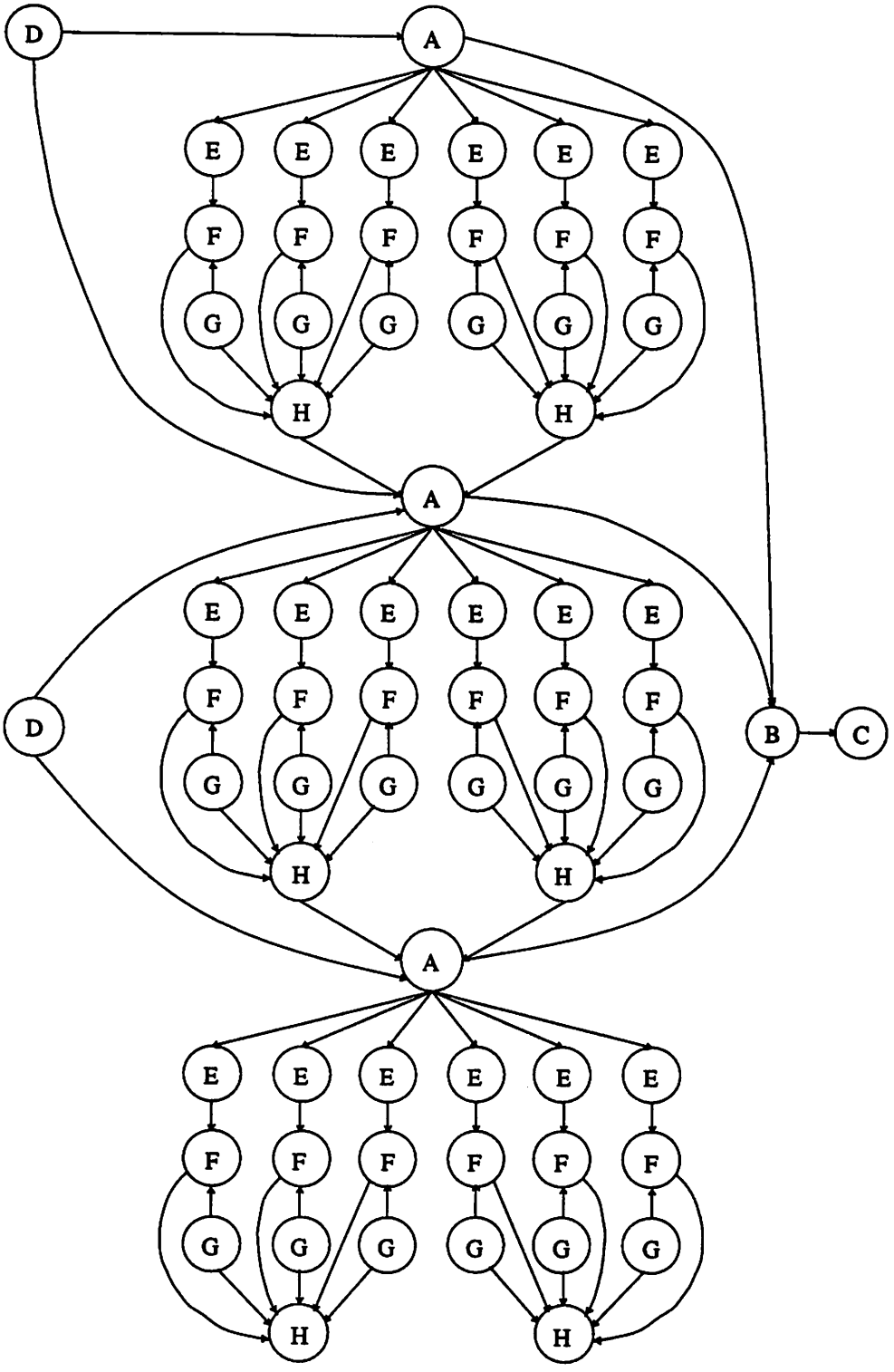


Figure 24. The APEG for the graph of figure 23. Observe that the APEG representation exposes very clearly, the looping inherent in the original SDF graph.

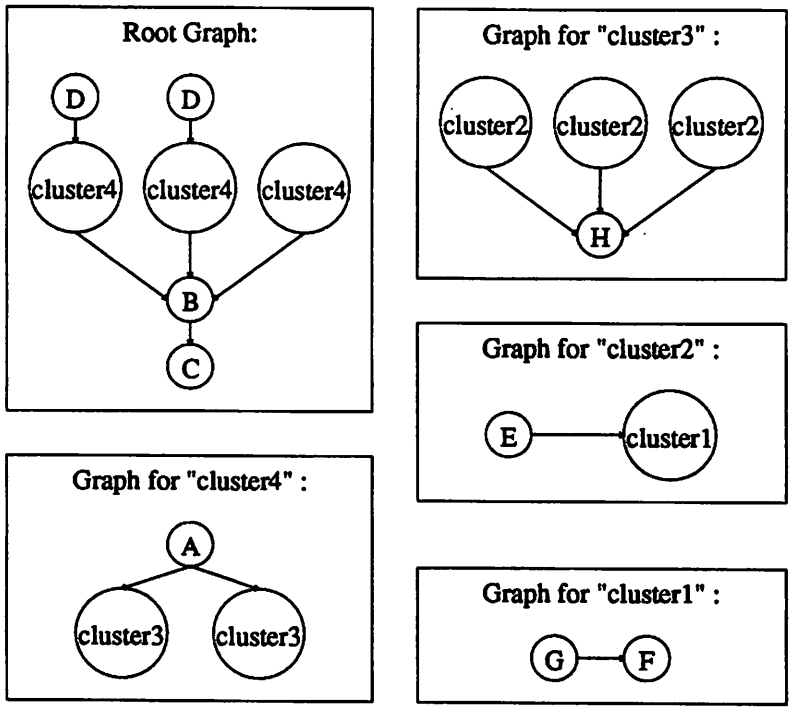


Figure 25. A hierarchy of clusters of repeated subgraphs for the apeg of figure 24.

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

Figure 26. The looped schedule suggested by the hierarchical decomposition of figure 25.

Since each of the clusters in figure 25 spans all invocations of the nodes which it subsumes, they all correspond to hierarchical clusters in the original SDF graph. Figure 27 shows the equivalent sequence of cluster formations in the original SDF graph.

An example of an APEG subgraph which does not translate to the SDF graph is the consolidation of the first and second invocations of *D* with, respectively, the first and second invocations of *cluster4* in figure 25. The formation of this cluster and the resulting schedule are depicted in figure 28. Observe that the code space size is no longer one code segment per actor; 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 APEG. We summarize and elaborate on these points with following definition and fact.

Definition: *Let* G = $\{N, \Lambda\}$ *be an SDF graph and let* P *denote its associated APEG (for*

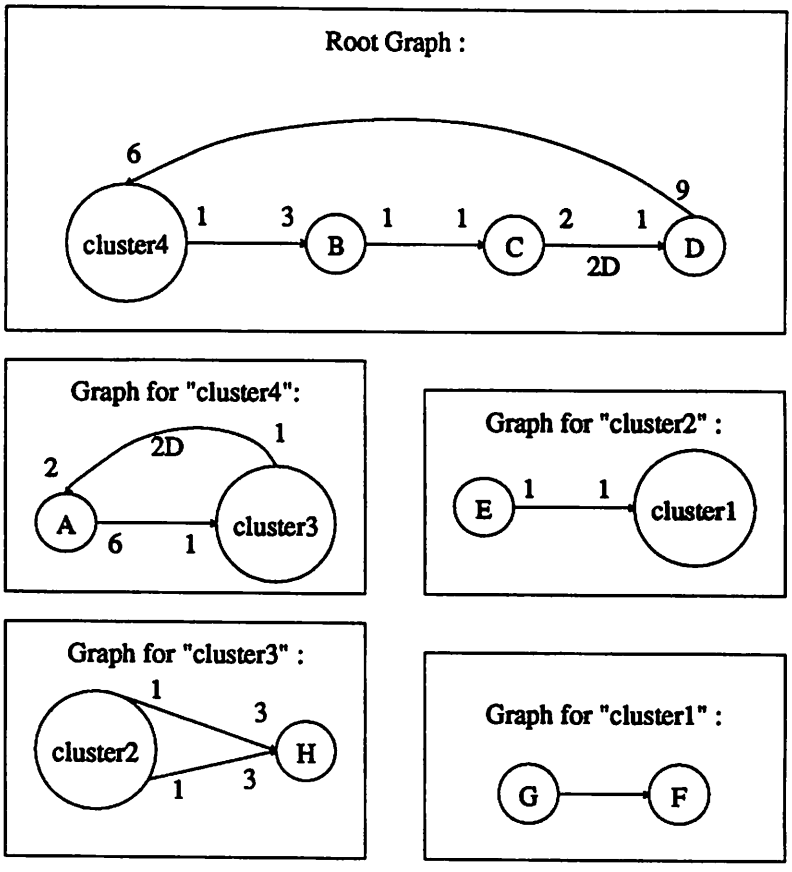
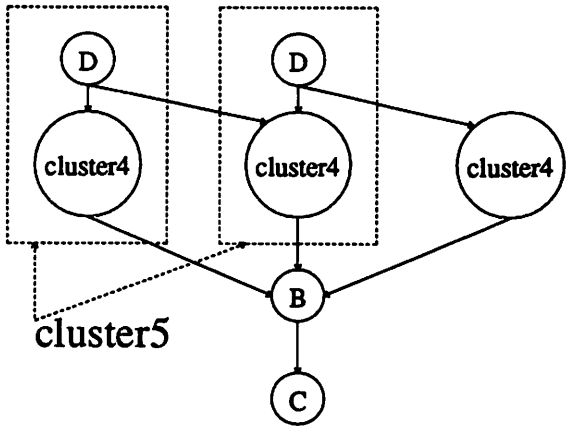


Figure 27. The hierarchy of subgraphs in the SDF graph which corresponds to the organization of APEG clusters in figure 25.



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

Figure 28. The consolidation of a repeated APEG subgraph which does not correspond to a cluster in the SDF graph, and the resulting schedule.

blocking factor 1).[4] *For any* A ∈ N, *we define the* **frequency** *of* A, *denoted* v(A), *to be equal to the number of invocations of* A *appearing in* P. *For* A_1, A_2 ∈ N, *we define*

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

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(v(A_1),v(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 reduced form of the frequency ratio between A_1 and A_2. The following fact expresses this observation in terms of the impact on the APEG of forming a cluster in an SDF graph.

Fact: *Let* G *be an SDF graph and let* P *be its associated APEG. Let A and B 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 APEG* $\hat{P}$ *for* $\hat{G}$ *can be obtained by combining*

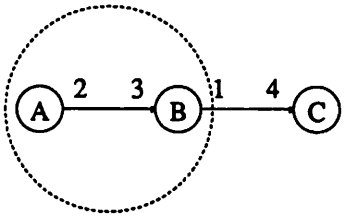
$\{A_1, A_2, ..., A_{F(A,B)}, B_1, B_2, ..., B_{F(B,A)}\}$;
$\{A_{F(A,B)+1}, A_{F(A,B)+2}, ..., A_{2F(A,B)}, B_{F(B,A)+1}, B_{F(B,A)+2}, ..., B_{2F(B,A)}\}$;
...
...
...

[4]This discussion can easily be generalized to consider arbitrary blocking factors, but we refrain from doing so for clarity.

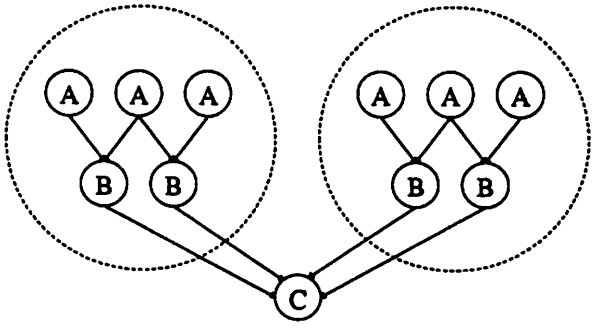
$\{A_{(r-1)F(A,B)+1}, A_{(r-1)F(A,B)+2}, \ldots, A_{rF(A,B)}, B_{(r-1)F(B,A)+1}, B_{(r-1)F(B,A)+2}, \ldots, B_{rF(B,A)}\}$

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

Figure 29 and table 10 illustrate the significance of F(*,*), and ν(*) in relating a cluster in the SDF graph to its APEG counterpart.



(a) A cluster in an SDF graph.



(b) The associated APEG cluster.

Figure 29.

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

Table 10. Quantities relating the SDF cluster and the APEG cluster of figure 29.

Figure 28 and the above fact suggest that a large increase in code size can result from forming a cluster in the APEG which does not have a counterpart in the SDF graph. For this reason we do not consider the consolidation of repeated APEG subgraphs which 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 which candidate clusters may introduce more efficiently by working with the APEG. This consideration renders the APEG more suitable than the SDF graph for the implementation of PGAN.

The PGAN algorithm 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 subsection 4 will then address the problem of checking for deadlock.

6.2. 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 first present the criteria for choosing the base node and the adjacent node for this initial approach. We will then illustrate a shortcoming of this approach which results from separating the selection of the two nodes in a cluster. Finally, we will conclude the subsection with an improved selection criteria which remedies this shortcoming and forms the core of the PGAN algorithm.

6.2.1. Selecting the Base Node

The following policy summarizes the criterion for selecting the base node for a cluster:

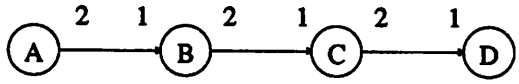
Policy: *The base node at a given algorithm iteration is chosen as the highest frequency node which has not already been considered as a base node.*

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 figures 30-32. Figure 30 shows an SDF graph and the associated APEG for a simple example of nested iteration.

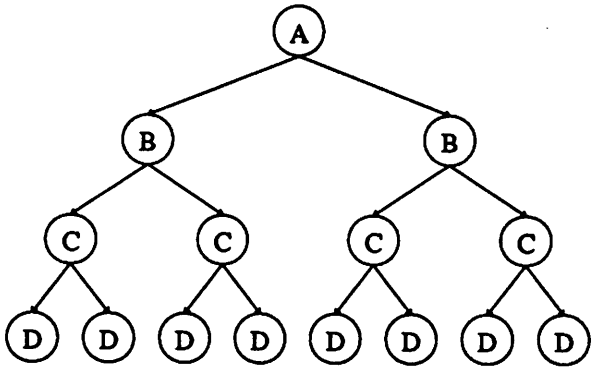
Figure 31 depicts the result of first coalescing nodes *B* and *C*, which both have lower frequency than *D*. The resulting schedule does not fully exploit the nested loop inherent in the original graph.

Figure 32 on the other hand, shows that 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.

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.

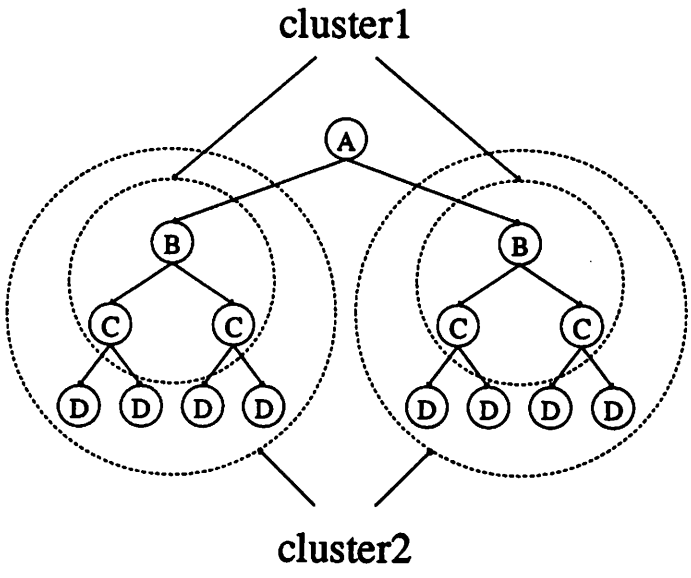


(a) An SDF graph ...



(b) and its APEG.

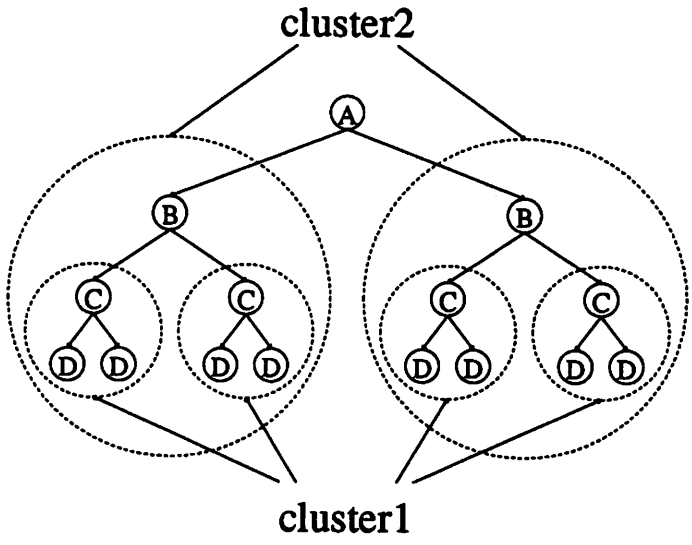
Figure 30. An SDF graph which suggests a nested loop. Scheduling this graph into this nested loop requires proper selection of the base node.



Schedule:

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

Figure 31. The result of first coalescing nodes of lower frequency in the APEG of figure 30. The schedule does not exhibit the optimal nested looping.



Schedule:

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

Figure 32. Clustering the highest frequency nodes first achieves the desired nested looping.

6.2.2. 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 inner most loop of a nested loop. Now we address the problem of choosing the node adjacent to the base node, which completes the specification of 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.

Figures 33-35 present an example in which the detection of nested looping depends upon proper selection of the adjacent node. Figure 33 shows an SDF graph and its associated APEG. From 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*.

Figures 34 and 35 detail the consequences of choosing *B* and *D*, respectively, as the adjacent node, and comparison of these figures reveals — in a manner analogous to that of figure 31 and figure 32 — 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 figures 36-38, which present a multirate graph, and in same manner as figures 30-35, 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, whereas the result of selecting the node of higher frequency is suboptimal. This

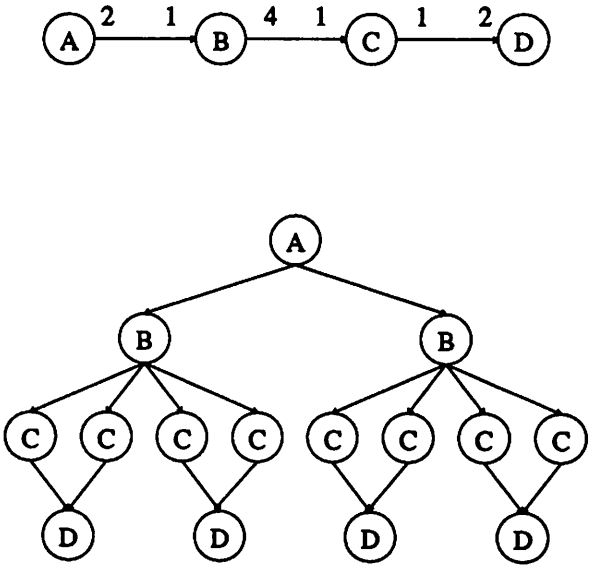
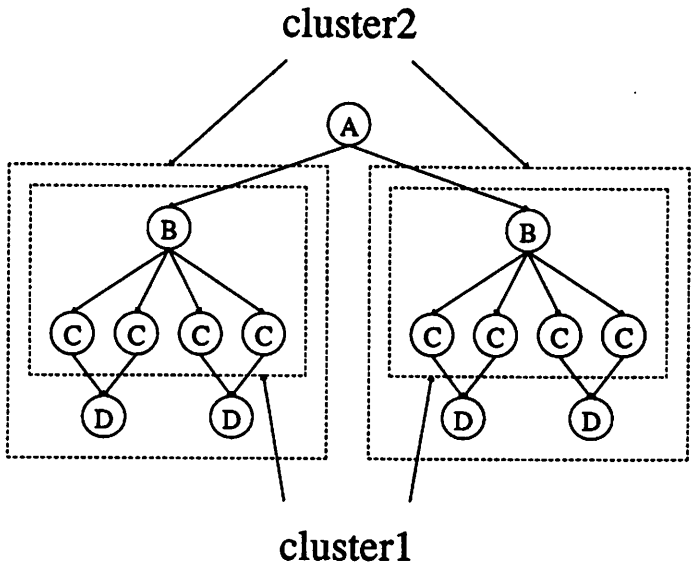


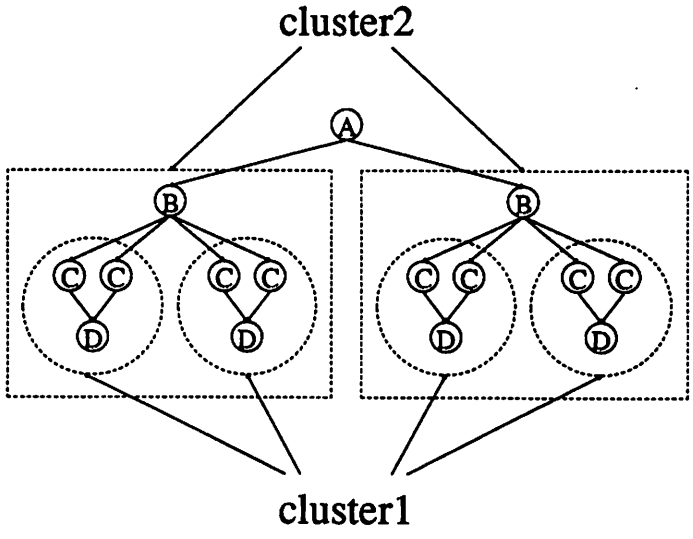
Figure 33. An example used to illustrate the impact which the selection of the adjacent node can have on the schedule.



Schedule:

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

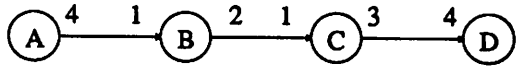
Figure 34. Selecting the candidate of lower frequency as the adjacent node. The resulting schedule does not fully exploit the nested looping suggested by this graph.



Schedule:

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

Figure 35. Selecting the higher frequency candidate D, as the adjacent node, leads to the desired nested looping.



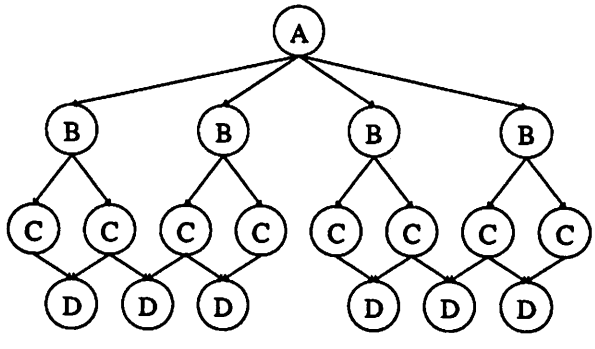
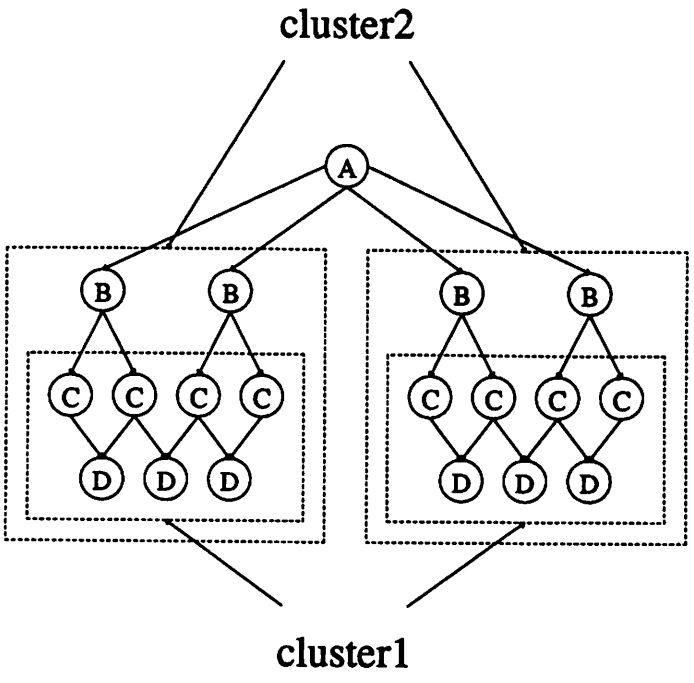


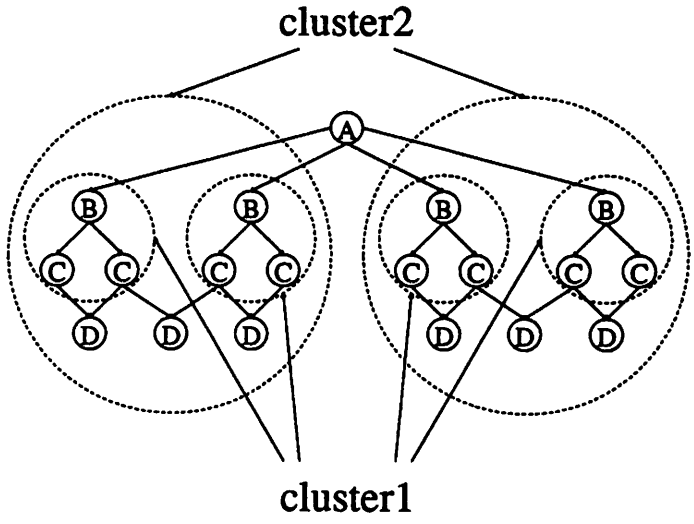
Figure 36. This example illustrates that frequency should not be the only criterion for selecting the adjacent node.



Schedule:

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

Figure 37. Selection of the higher frequency candidate D does not result in the nested loop suggested by the graph of figure 36.



Schedule:

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

Figure 38. Selecting the lower frequency candidate, B, results in the desired nested looping.

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: *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 of* S *for which* F(b,*) *is minimum.*

6.2.3. Selecting the Cluster of Highest Frequency

Policies 5 and 6 together comprise our initial approach for selecting clusters to organize nested looping. Our development of policy 6 illustrated that for a given base node b, we should choose as the adjacent node the candidate which minimizes F(b,*). Since F(x,y) = v(x) / gcd(v(x),v(y)), we see that this is equivalent to choosing the node which minimizes gcd(v(b),v(*)). 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 6 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.

Figures 39-41 illustrate an example in which cluster selection based on the base node - adjacent node approach fails to extract 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 41 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* $p = (p_1, p_2)$ *be a pair of nodes in an SDF graph G. Then we define* $v(p)$, *called*

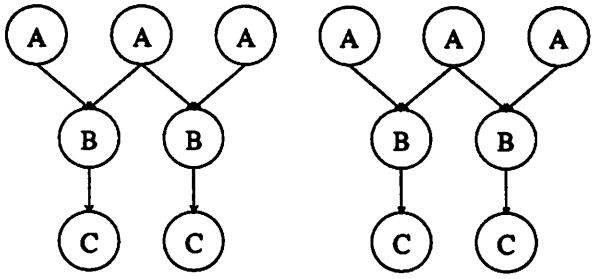
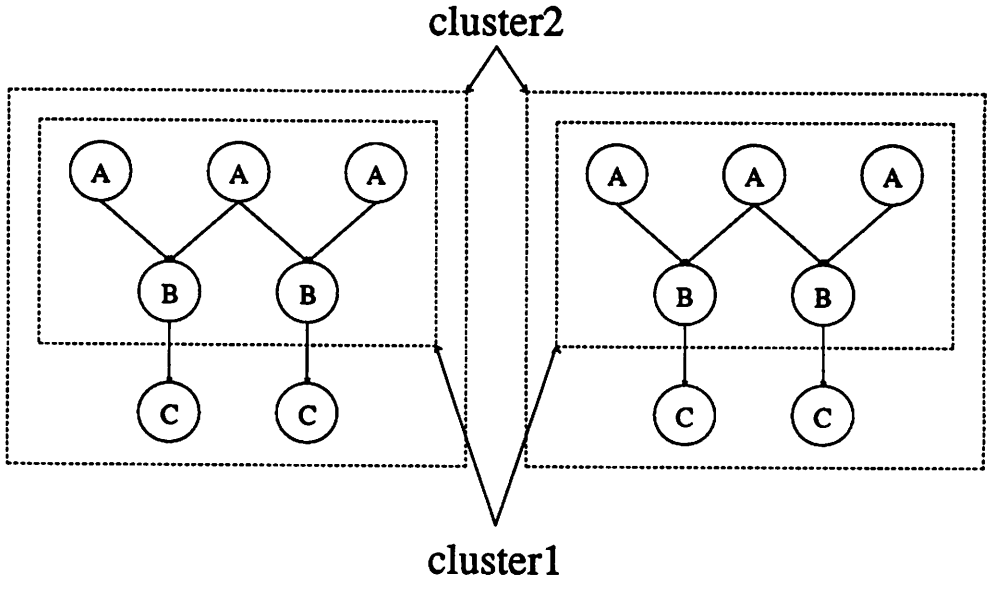


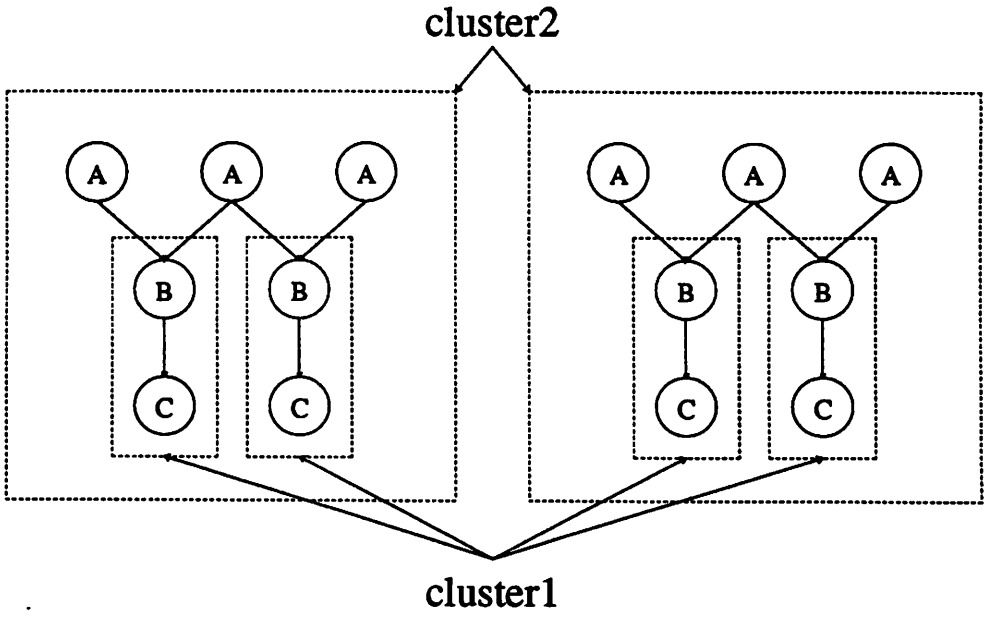
Figure 39. An example used to illustrate that cluster selection based on the base node-adjacent node can fail to extract the highest frequency candidate.



Schedule:

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

Figure 40. This cluster hierarchy results from applying the base node- adjacent node scheme to the APEG of figure 39. Observe that the looped schedule does not exhibit the full amount of nesting which can be obtained from this example.



Schedule:

(2 cluster2) <-> (2 (3A) (2 cluster1)) <-> (2 (3A) (2BC))

Figure 41. This figure shows the cluster hierarchy obtained from selecting the highest frequency clusters for the example of figure 39. The resulting schedule fully exploits the nested looping suggested by this graph.

the **frequency** *of* p, *by* $v(p) = \gcd(v(p_1), v(p_2))$.

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

Policy: *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 of* S *which maximizes* $v(*)$.

6.3. 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. [9] shows that for a certain class of scheduling algorithms, successful completion guarantees that a periodic 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 APEG. 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 APEG requires an implementation in which PGAN clustering decisions are carried out on a hierarchically maintained APEG rather than an SDF graph.

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

Definition: *A* **path** *in an APEG G is a finite sequence* p *of arcs* $a_1, a_2, ..., a_n$, *such that the source of* a_{i+1} *is the sink of* a_i, *for* $i \in \{1, ..., n-1\}$. *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 APEG G then we define the expression "x→y" to be 1 if there is a path in G from x to y and 0 otherwise.*

Definition: *Given an APEG 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→B, *and every diagonal element of R is 0.*

Figure 39 shows an APEG, and a reachability matrix for it. Note that since a reachability matrix contains boolean entries, it can be implemented with a storage cost of only one bit per entry.

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 $z_1, z_2, ..., z_N$ in an APEG A introduces deadlock by

(1) assuming that consolidating C results in a valid APEG $\hat{A}$ — equivalently, we assume that C does not introduce a deadlock — and

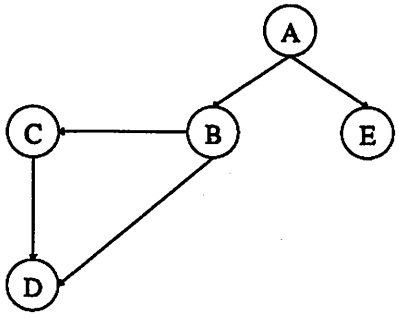
(2) calculating the reachability matrix $\hat{R}$ for $\hat{A}$.

If $\hat{R}$ contains any nonzero diagonal elements then our assumption (1) is false, and it follows that 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 APEG and reachability matrix for the next algorithm iteration.

We show now that the reachability matrix $\hat{R}$ for the APEG which results from consolidating $C = \{z_1, z_2, ..., z_N\}$ into a single node z under assumption (1) above, can be computed efficiently from R, the reachability matrix for A. Our development here requires the following notation:

Notation: Given an APEG 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.

Now to compute $\hat{R}$ from R, first observe that if x and y are in N($\hat{A}$) and $x,y \neq z$, then $\hat{R}[x, y] = R[x, y]$, since the path in A from x to y does not intersect the subgraph associated with C. This submatrix transfer is depicted in arrow p of figure 43.



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

Figure 42. An APEG and a reachability matrix for that APEG.

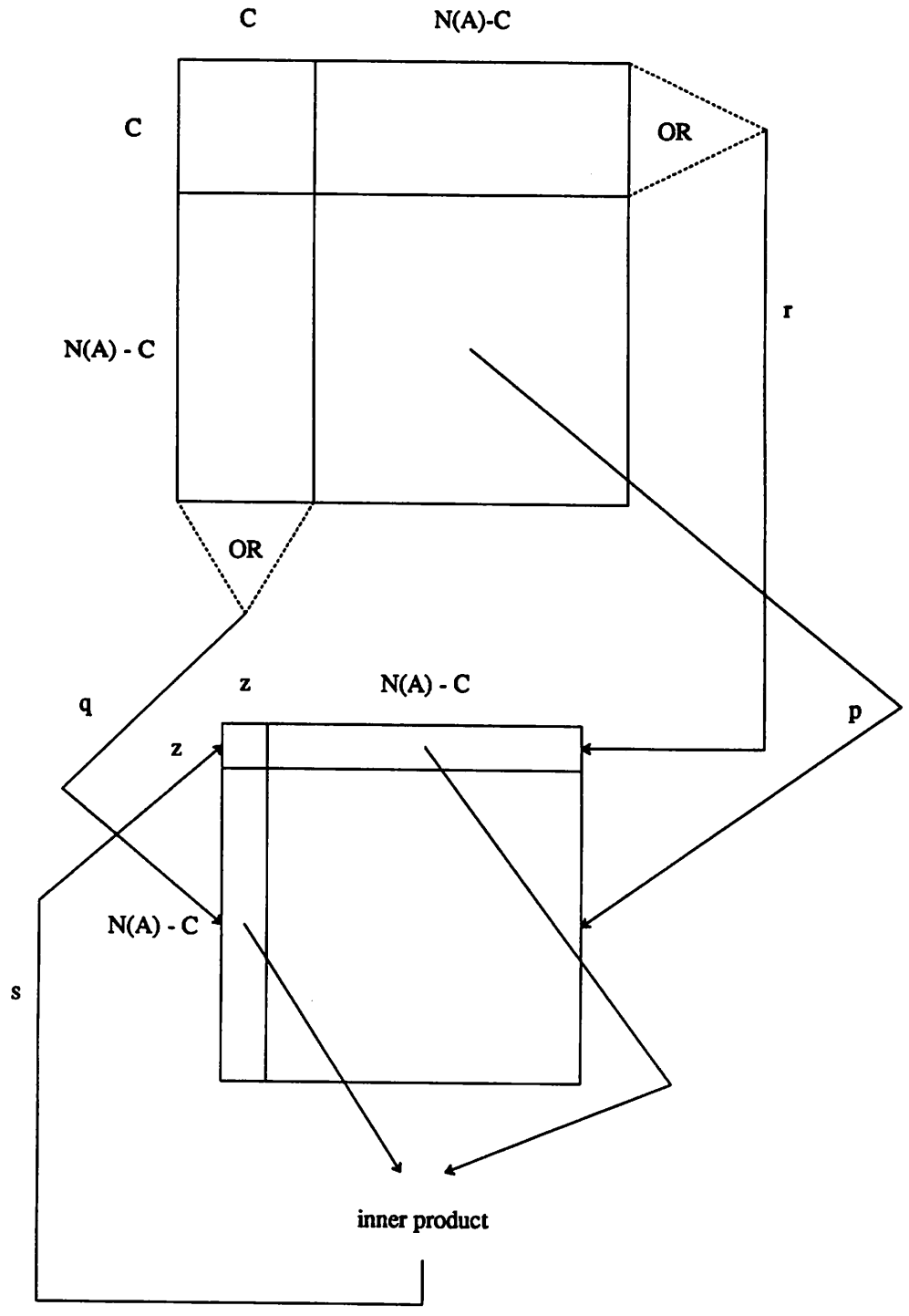


Figure 43. This figure illustrates the process of updating a reachability matrix to reflect the formation of a cluster. The upper grid represents the original matrix; the lower grid represents the updated matrix; "C" represents the set of indices corresponding to the set of nodes in the candidate cluster; "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.

Second, if $y \in N(\hat{A})$ and $y \neq z$ then $\hat{R}[y, z] = (y \rightarrow z_1) + (y \rightarrow z_2) + ... + (y \rightarrow z_N)$. Thus *or*-ing the N column vectors $R[*,z_i]$ and then truncating the entry for each z_i, yields the entire column $\hat{R}[*,z]$ excluding the diagonal entry $\hat{R}[z, z]$. Similarly *or*-ing the rows $R[z_i,*]$ and eliminating the z_i entries yields the nondiagonal elements of the row $\hat{R}[z,*]$. These are depicted in arrows q and r respectively of figure 43.

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 43. Since this element is the only diagonal element in $\hat{R}$ which is not simply transferred from R, it follows that C introduces a deadlock if and only if the calculation for $\hat{R}[z,z]$ yields "1".

As the outline in figure 43 highlights, the only computations involved in the calculation for $\hat{R}$ are the elementwise *or*-operations of arrows q and r, and the simple inner product of arrow s. This method is thus efficient enough to be practical for checking that candidate clusters do not introduce deadlocks.

6.4. Summary

The preceding development of PGAN is summarized below with a pseudocode outline of the algorithm. 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.

6.4.1. Scheduling the Clustered Graphs

After the PGAN cluster-building phase, the root graph and the graph for each cluster must be scheduled. Our scheduling algorithm attempts at each step, to find a complete set of invocations — all of the invocations for an actor — and schedules such a set as a schedule loop. All of the invocations of an actor 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 in performed in such a way that no actor is scheduled twice before all other actors 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 3.

An outline of the hierarchical cluster-building process of PGAN.

Suppose we are given an SDF graph *G*. The following steps describe the operation of PGAN on *G*:

1. *Create a list L consisting of all pairs of mutually adjacent nodes in G, sorted in decreasing order of* $v(*)$.

2. *Loop until L is empty*

   (a) *Remove the element* $p = (p_1,p_2)$ *at the head of L.*

   (b) *If the consolidation of* p_1 *and* p_2 *into a single node does not introduce a deadlock in G, then:*
   (1) *Replace* p_1 *and* p_2 *with a single node C in G.*
   (2) *Remove from L all members which contain either* p_1 *or* p_2.
   (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 L's sorted order.*

6.4.2. Results

We have implemented PGAN within *Ptolemy*, a heterogeneous platform for software-prototyping [10]. 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 5, 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. Figures 44-45 depict the APEG, and the clustering sequences and schedule, which result from applying PGAN to the graph of figure 20 — the example which 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 46 through figure 48, 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 APEG. As a result, *C* is not represented in any cluster, and the hierarchical structure reflects the desired partition of figure 19. As expected, our recursive scheduling procedure yields the optimal schedule.

The graph of figure 23 also contains looping opportunities which span sample-rate boundaries. The PGAN schedule given in figure 26 contains only one code-segment per actor. The

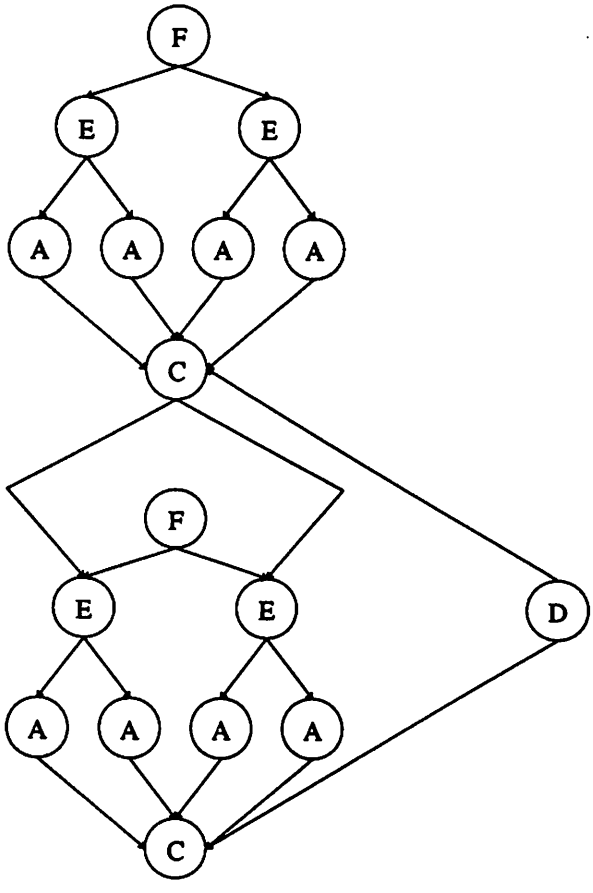
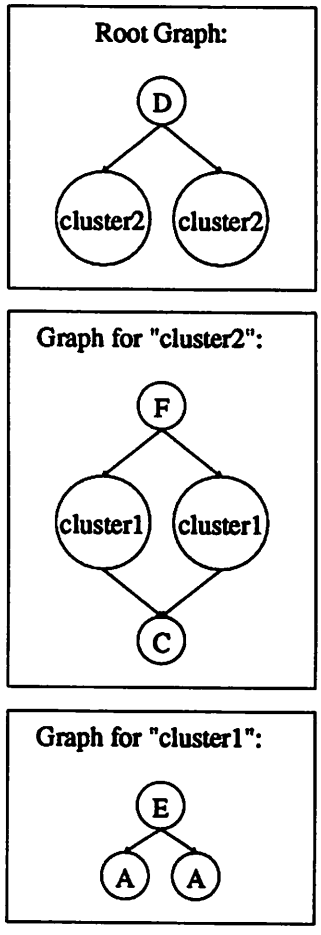


Figure 44. The APEG for the graph of figure 20, which was used to show the inability of CSUF to detect looping opportunities which occur across sample rate boundaries. We return to this example to illustrate how PGAN succeeds in detecting these opportunities.



Schedule:

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

vs.

The CSSF Schedule:

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

Figure 45. The PGAN clustering sequence and the resulting schedule for the APEG of figure 44. The CSUF schedule is given also, for comparison.

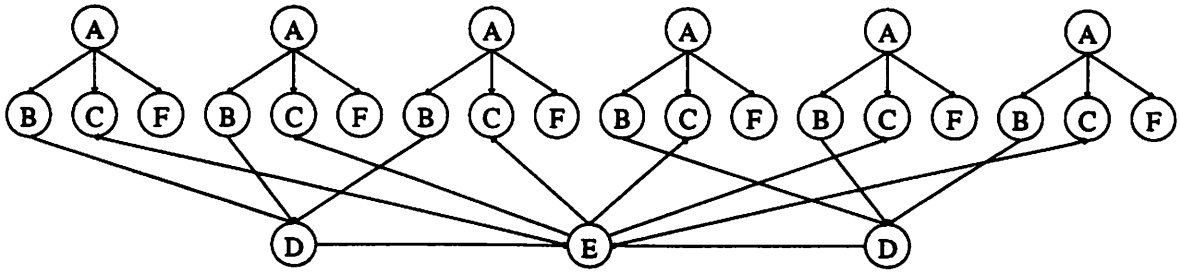


Figure 46. The APEG for the graph of figure 18, which was used to illustrate the problem of partitioning CSUF's which introduce deadlocks. We return to this example to demonstrate that PGAN's incremental approach to cluster-building avoids the partitioning problem.

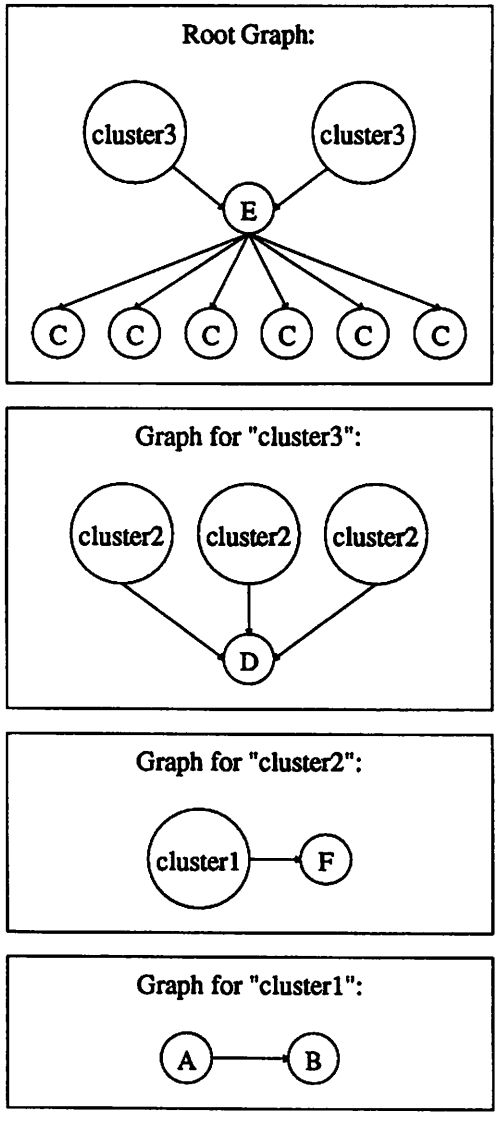


Figure 47. The PGAN clustering sequence for the APEG of figure 46.

Schedule:

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

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

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

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

Figure 48. The schedule which results from the organization in figure 47.

CSUF schedule, shown in figure 49, is much less efficient.

7. CONCLUSIONS

An evolution of algorithms for extracting looping information from SDF graphs, has been presented. The first method — postprocessing a minimum buffer-length scheduler with a pattern-matcher — illustrated 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 [10], and preliminary results confirm that this approach exploits opportunities for looping more effectively than its predecessors.

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

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

Figure 49. The CSUF schedule for the example of figure 23.

1. Edward A. Lee and David G. Messerschmitt, "Synchronous Dataflow," *Proceedings of the IEEE*, (September 1987).

2. Edward A. Lee, "Programmable DSP Architectures: Part I," *IEEE ASSP Magazine*, (October, 1988).

3. Edward A. Lee, "Programmable DSP Architectures: Part II," *IEEE ASSP Magazine*, (January, 1989).

4. Edward A. Lee, "Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages," *VLSI Signal Processing III*, IEEE Press, (1988).

5. Edward A. Lee, Wai Hung Ho, Edwin Goei, Jeffrey Bier, and Shuvra Bhattacharyya, "Gabriel: A Design Environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37(11) pp. 1751-1762 (November, 1989).

6. Wai Ho, Edward A. Lee, and David G. Messerschmitt, "High Level Dataflow Programming for Digital Signal Processing," *VLSI Signal Processing III*, IEEE Press , (1988).

7. Wai Ho, "Code Generation for Digital Signal Processors Using Synchronous Dataflow," *Master's Degree Report, U.C. Berkeley*, (May, 1988).

8. Stephen How, "Code Generation for Multirate DSP Systems in GABRIEL," *Master's Degree Report, U.C. Berkeley*, (May, 1990).

9. Edward A. Lee and David G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Transactions on Computers* C-36(2) pp. 24-35 (January, 1987).

10. Soonhoi Ha, Joseph Buck, Edward A. Lee, and David G. Messerschmitt, "PTOLEMY: A Platform for Heterogeneous Simulation and Prototyping," *European Simulation Conference*, (June, 1991).