In *Journal of VLSI Signal Processing Systems*, Vol 21, No. 2, pages 151-166, June 1999.
© 1999 Kluwer Academic Publishers.

SYNTHESIS OF EMBEDDED SOFTWARE FROM SYNCHRONOUS DATAFLOW SPECIFICATIONS

Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee

ABSTRACT

The implementation of software for embedded digital signal processing (DSP) applications is an extremely complex process. The complexity arises from escalating functionality in the applications; intense time-to-market pressures; and stringent cost, power and speed constraints. To help cope with such complexity, DSP system designers have increasingly been employing high-level, graphical design environments in which system specification is based on hierarchical dataflow graphs. Consequently, a significant industry has emerged for the development of dataflow-based DSP design environments. Leading products in this industry include SPW from Cadence, COSSAP from Synopsys, ADS from Hewlett Packard, and DSP Station from Mentor Graphics.

This paper reviews a set of algorithms for compiling dataflow programs for embedded DSP applications into efficient implementations on programmable digital signal processors. The algorithms focus primarily on the minimization of code size, and the minimization of the memory required for the buffers that implement the communication channels in the input dataflow graph. These are critical problems because programmable digital signal processors have very limited amounts of on-chip memory, and the speed, power, and cost penalties for using off-chip memory are often prohibitively high for embedded applications. Furthermore, memory demands of applications are increasing at a significantly higher rate than the rate of increase in on-chip memory capacity offered by improved integrated circuit technology.

This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, Philips, and Rockwell.

S. S. Bhattacharyya is with the Department of Electrical and Computer Engineering and the Institute for Advanced Computer Studies, University of Maryland, College Park.

P. K. Murthy is with Angeles Design Systems, San Jose, California.

E. A. Lee is with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

1. Introduction

Numerous software design environments for digital signal processing applications, such as those described in [6, 14, 16, 23, 24, 27], support code-generation for programmable digital signal processors used in embedded systems. Traditionally, programmable digital signal processors have been programmed manually, in assembly language, and this is a tedious, error-prone process at best. Hence, generating code automatically is a desirable goal. Since the amount of on-chip memory in programmable digital signal processors is severely limited, it is imperative that the generated code be parsimonious in its memory usage. Adding off-chip memory is often highly unattractive due to increased cost, increased power requirements, and a speed penalty that will affect the feasibility of real-time implementations.

One approach to automatic code generation is to specify the program in an imperative language such as C, C++, or FORTRAN and use a good compiler. However, even the best compilers today produce inefficient code [31], although a significant research community is evolving to address the challenges of compiling imperative programming languages into implementations on embedded processors such as programmable digital signal processors [20]. In addition, specifications in imperative languages are difficult to parallelize, are difficult to change due to side effects, and offer few chances for any formal verification of program properties. An alternative is to use a block diagram language based on a model of computation with strong formal properties such as synchronous dataflow [17] to specify the system, and to perform code-generation starting from this specification. One reason that a compiler for a block diagram language is likely to deliver better performance than a compiler for an imperative language is that the underlying model of computation often exposes restrictions on the control flow of the specification, and this can be profitably exploited by the compiler.

Synchronous dataflow (SDF) [17] is a special case of dataflow. In SDF, a program is represented by a directed graph in which each vertex (**actor**) represents a computation, an edge specifies a FIFO buffer, and each actor produces (consumes) a fixed number of data values (**tokens**) onto (from) each output (input) edge per invocation. A parameter on each edge specifies the number of initial tokens (called **delays**) residing on that edge.

One code-generation strategy followed in many block diagram programming environments is called **threading**; in this method, the underlying model (in this case, SDF) is scheduled to generate a sequence of actor invocations (provided that the model can be scheduled at compile time of-course). A code generator then steps through this schedule, and for each actor encountered in the schedule, the code generator inserts a code block that implements the computation specified by the given actor. The individual code blocks, which can be specifications in assembly language or any high level language, are obtained from a predefined library of actor code blocks. Typically, in block diagram design tools for DSP, assembly language (feasible since the actors are usually small, modular components) or C is used to specify the functionality of individual code blocks. By "compiling an SDF graph", we mean exactly the strategy described above for generating a software implementation from an SDF graph specification of the system in a block diagram environment.

We also assume that the code-generator generates inline code; this is because the alternative of using subroutine calls can have unacceptable overhead, especially if there are many small tasks. A key problem that arises with such an in-line code generation strategy is code-size explosion. For example, if an actor appears 20 times in the schedule, then there will be 20 code blocks in the generated code. Clearly, such code duplication can consume enormous amounts of memory, especially if high actor invocation counts are involved.

Generally, the only mechanism to combat code size explosion while maintaining inline code is the use of loops in the target code. If an actor's code block is encapsulated by a loop, then multiple invocations of that actor can be carried out without any code duplication. This paper is devoted to the construction of efficient loop structures from SDF graphs to allow the advantages of inline code generation under stringent memory constraints.

As mentioned earlier, a compiler for an imperative language cannot usually exploit the restrictions in the overall control flow of a DSP application system. However, the individual actor code blocks within an actor are usually much simpler, and may even correspond to basic blocks that compilers are adept at handling. Hence, for DSP design tools in which individual actors can be programmed using high level languages, compiling an SDF graph using the methods we describe in this paper does not preclude the use of or obviate the need for a good imperative language compiler. On the contrary, we believe that the most promising approach is a strategy that combines powerful SDF optimizations at a coarse-grain level, with aggressive imperative compiler technology applied to optimize the internals of individual actor code blocks. We expect that as compiler technology improves, such a hybrid approach will eventually produce code competitive to hand-written code. However, in this paper, we only consider the code and buffer memory optimization possible at the SDF graph level. Issues relating to the interaction between compilation at the SDF graph level, and the lower-level compilation of individual actor code blocks form an important direction for further study.

2. Synchronous dataflow

Fig. 1(a) shows a simple SDF graph. Each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor, and the "D" on the edge from actor A to actor B specifies a unit delay. Each unit of delay is implemented as an initial token on the edge. Given an SDF edge e, we denote the source actor, sink actor, and delay of e by $src(e)$, $snk(e)$, and $d(e)$. Also, $p(e)$ and $c(e)$ denote the number of tokens produced onto e by $src(e)$ and consumed from e by $snk(e)$.

A **schedule** is a sequence of actor firings. We compile an SDF graph by first constructing a **valid schedule** — a finite schedule that fires each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each edge. Corresponding to each actor in the schedule, we instantiate a code block that is obtained from a library of predefined actors. The resulting sequence of code blocks is encapsulated within an infinite loop to generate a software implementation of the SDF graph.

SDF graphs for which valid schedules exist are called **consistent** SDF graphs. In [17], efficient algorithms are presented to determine whether or not a given SDF graph is consistent, and to determine the minimum number of times that each actor must be fired in a valid schedule. We represent these minimum numbers of firings by a vector $\mathbf{q}_G$, indexed by the actors in G (we

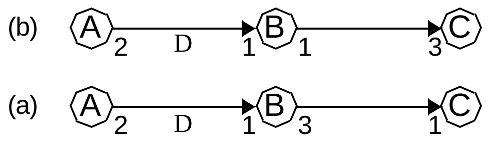


Figure 1. Examples of SDF graphs.

often suppress the subscript if G is understood). These minimum numbers of firings can be derived by finding the minimum positive integer solution to the **balance equations** for G, which specify that $\mathbf{q}$ must satisfy

$$\mathbf{q}(src(e)) \times p(e) = \mathbf{q}(snk(e)) \times c(e) \text{, for every edge } e \text{ in } G. \qquad (1)$$

The vector $\mathbf{q}$, when it exists, is called the **repetitions vector** of G.

3. Constructing memory-efficient loop structures

This section informally outlines the interaction between the construction of valid schedules for SDF graphs and the memory requirements of the compiled code.

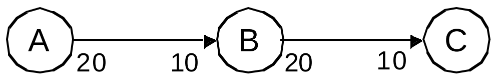
To understand the problem of scheduling SDF graphs to minimize memory requirements, it is useful to examine closely the mechanism by which iteration is specified in SDF. In an SDF graph, iteration of actors in a valid schedule arises whenever the production and consumption parameters along an edge in the graph differ. For example, consider the SDF graph in Figure 2(a), which contains three actors, labeled A, B and C. The 2-to-1 mismatch on the left edge implies that within a valid schedule, B must be invoked twice for every invocation of A. Similarly, the mismatch on the right edge implies that we must invoke C twice for every invocation of B.

Figure 2(b) shows four possible valid schedules that we could use to implement Figure 2(a). For example, the first schedule specifies that first we are to invoke A, followed by B, followed by C, followed by B again, followed by three consecutive invocations of C. The parenthesized terms in schedules 2, 3 and 4 are used to highlight repetitive invocation patterns in these schedules. For example, the term $(2BC)$ in schedule 4 represents a loop whose iteration count is 2 and whose body is the invocation sequence BC; thus, $(2BC)$ represents the firing sequence $BCBC$. Similarly, the term $(2B(2C))$ represents the invocation sequence $BCCBCC$. Clearly, in addition to providing a convenient shorthand, these parenthesized loop terms, called **schedule loops**, present the code generator with opportunities to organize loops in the target program, and we see that schedule 2 corresponds to a nested loop, while schedules 3 and 4 correspond to cascades of loops. For example, if each schedule loop is implemented as a loop in the target program, the code generated from schedule 4 would have the structure shown in Figure

2(c).

We see that if each schedule loop is converted to a loop in the target code, then each appearance of an actor in the schedule corresponds to a code block in the target program. Thus, since actor C appears twice in schedule 4 of Figure 2(b), we must duplicate the code block for C in the target program. Similarly, we see that the implementation of schedule 1, which corresponds to the same invocation sequence as schedule 4 with no looping applied, requires seven code blocks. In contrast, in schedules 2 and 3, each actor appears only once, and thus no code duplication is required across multiple invocations of the same actor. We refer to such schedules as **single appearance** schedules, and we see that neglecting the code size overhead associated with the loop control, any single appearance schedule yields an optimally compact inline implementation of an SDF graph with regard to code size. Typically the loop control overhead is small, particularly in

(a)



(b)

Valid Schedules

1. ABCBCCC
2. A(2 B(2 C))
3. A(2 B)(4 C)
4. A(2 BC)(2 C)

(c)

```
code block for A
for (i=0; i<2; i++) {
              code block for B
              code block for C
}
for (i=0; i<2; i++) {
              code block for C
}
```

Figure 2. An example used to illustrate the interaction between scheduling SDF graphs and the memory requirements of the generated code.

programmable digital signal processors, which usually have provisions to manage loop indices and perform the loop test in hardware, without explicit software control.

Scheduling can also have a significant impact on the amount of memory required to implement the buffers on the edges in an SDF graph. For example, in Figure 2(b), the buffering requirements for the four schedules, assuming that one separate buffer is implemented for each edge, are 50, 40, 60, and 50 respectively.

Note that this model of buffering — maintaining a separate memory buffer for each data flow edge — is convenient and natural for code generation, and it is the model used, for example, in the SDF-based code generation environments described in [6], [14], [24]. More technical advantages of this buffering model are elaborated on in [22].

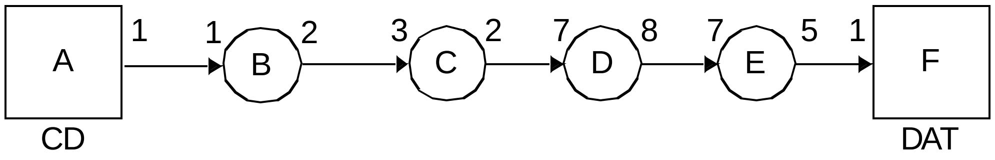
4. Relative prioritization of code and data minimization objectives

There are two natural angles for approaching the problem of joint minimization of code size and buffer memory requirements. The first approach is to study the problem of constructing a minimum buffer memory schedule, and then incorporate techniques for minimizing the code size into the approach that is developed for minimizing buffer memory. Here, the objective is to construct a minimum buffer memory implementation that has minimum code size over all minimum buffer memory implementations. Conversely, first priority could be given to minimizing code size. This would yield the goal of computing a minimum buffer memory schedule over all implementations that require minimum code size. Once such a priority-based algorithm is established, post-processing techniques can be developed to balance the solutions computed by the priority-based algorithm according to the code size and buffer memory capacities of the target implementation.

This paper focuses on the latter angle of attack — assigning first priority to code size minimization, and second priority to minimizing the buffer memory requirement. This approach is preferable because for practical synchronous dataflow graphs, giving first priority to code size minimization typically yields a significantly more favorable code size/buffer memory trade-off than giving first priority to buffer memory minimization.

An example of this phenomenon is shown in Figure 3. The top part of this Figure depicts an SDF representation of a sample rate conversion system for interfacing a compact disk player (44.1 kHz) to a digital audio tape player (48 kHz). The sample rate conversion is performed in four stages: 2:1, 4:3, 4:7, and 5:7. Explicit up samplers and down samplers are omitted, and it is assumed that the FIR filters are general polyphase filters [26].

The bottom part of Figure 3 shows the code size and buffer memory costs for various schedules when the implementation target is a single Motorola 56000 programmable data signal processor. The first entry in this table corresponds to a minimum buffer implementation that does not incorporate any use of loops to reduce the code size. This is the *worst* minimum buffer memory implementation. The second entry corresponds to a minimum buffer memory implementation in which looping is optimally employed to reduce code size. This gives the memory costs for a minimum buffer memory schedule that has minimum code size over all minimum data schedules. The third entry shows the memory costs for a minimum code size schedule that has *maximum* buffer memory costs over all minimum code size schedules. Finally, the fourth entry shows the



	Code	Data
Minimum buffer schedule, no looping	**13735**	**32**
Minimum buffer schedule, with looping	**9400**	**32**
Worst minimum code size schedule	**170**	**1021**
Best minimum code size schedule	**170**	**264**

Figure 3. A comparison of the program and buffer memory requirements of various schedules for a sample rate conversion application.

memory cost for a minimum code size schedule that has minimum buffer memory cost over all minimum code size schedules. Comparing the second and fourth entries of the table in Figure 3, we see that in most implementation contexts, the optimal solution that results when we give first priority to code size minimization is clearly preferable to the optimal solution that results when we give first priority to buffer memory minimization: the "best minimum code size schedule" has a code size cost that is 55 times less than that of the "best minimum buffer schedule," while the buffer memory cost of the best minimum code size schedule is only 8 times larger; furthermore, the best minimum code size schedule can be accommodated within the on-chip memories of most programmable digital signal processors, while the 9400-word code size cost of the best minimum buffer schedule is too large for many processors.

5. Buffer memory metrics

Given an edge e in G, we define the **total number of samples exchanged** on e, denoted $TNSE(e, G)$, or simply $TNSE(e)$ if G is understood, by

$$TNSE(e) = \mathbf{q}_G(src(e)) \times p(e) . \tag{2}$$

Thus, $TNSE(e)$ is the number of tokens produced onto e in one period of a valid schedule.

For example, in Figure 1(a), $\mathbf{q}(A, B, C) = (3, 6, 2)$, and thus,

$$TNSE((A, B)) = TNSE((B, C)) = 6 .$$

Given an SDF graph $G = (V, E)$, a valid schedule S, and an edge e in G, $max\_tokens(e, S)$ denotes the maximum number of tokens that are queued on e during an execution of S. For Figure 1(a), if

$$S_1 = (3A)(6B)(2C) \text{ and } S_2 = (3A(2B))(2C),$$

then $max\_tokens((A, B), S_1) = 7$ and $max\_tokens((A, B), S_2) = 3$.

We define the **buffer memory requirement** of a schedule S by

$$buffer\_memory(S) \equiv \sum_{e \in E} max\_tokens(e, S) . \tag{3}$$

Thus,

$$buffer\_memory(S_1) = 7 + 6 = 13 \text{ and } buffer\_memory(S_2) = 3 + 6 = 9 .$$

A valid single appearance schedule that minimizes the buffer memory requirement over all valid single appearance schedules is called a **buffer memory optimal schedule**.

If Z is a subset of actors in a connected, consistent SDF graph G,

$$\rho_G(Z) \equiv gcd(\{\mathbf{q}_G(A) | A \in Z\}),^1 \tag{4}$$

and we refer to this quantity as the **repetition count** of Z.

6. Subindependence

Since valid single appearance schedules implement the full repetition inherent in an SDF graph without requiring subroutines or code duplication, it is useful to examine the topological conditions required for such schedules to exist. First, suppose that G is a connected, consistent acyclic SDF graph containing n actors. Then we can take some root actor R_1 of G and fire all $\mathbf{q}_G(R_1)$ invocations of R_1 in succession. After all invocations of R_1 have fired, we can remove R_1 from G, pick a root actor R_2 of the new acyclic SDF graph, and schedule its $\mathbf{q}_G(R_2)$ repetitions in succession. Clearly, we can repeat this process until no actors are left, to obtain the single appearance schedule $(\mathbf{q}_G(R_1)R_1)(\mathbf{q}_G(R_2)R_2)\ldots(\mathbf{q}_G(R_n)R_n)$ for G. Thus, we see that any consistent acyclic SDF graph has at least one valid single appearance schedule.

The following result has been established concerning the existence of single appearance schedules for general SDF graph topologies (SDF graphs that are not necessarily acyclic) [4].

Theorem 1: • An SDF graph has a single appearance schedule if and only if each strongly connected component has a single appearance schedule.

• A strongly connected SDF graph has a single appearance schedule only if we can partition the actors into two subsets P_1 and P_2 such that P_1 is precedence-independent of P_2 throughout a single schedule period. That is, for each edge α directed from a member of P_2 to a member of P_1, $d(\alpha) \geq c(\alpha)\mathbf{q}(snk(\alpha))$.

This form of precedence-independence is referred to as ***subindependence***. Thus a strongly connected SDF graph has a single appearance schedule only if its actors can be partitioned into

1. The greatest common divisor is denoted by *gcd*.

subsets P_1 and P_2 such that P_1 is subindependent of P_2. If such a partition exists, the strongly connected SDF graph is ***loosely interdependent***, otherwise it is ***tightly interdependent***. The following theorem relates the concept of loose interdependence the existence of to single appearance schedules [3]:

Theorem 2: A strongly connected, consistent SDF graph G has a single appearance schedule if and only if every strongly connected subgraph of G is loosely interdependent.

Partitioning loosely interdependent SDF graphs based on subindependence relationships defines a decomposition process for hierarchically scheduling SDF graphs. This decomposition process leads to single appearances schedules whenever they exist [3].

However, this method of decomposition is useful even when single appearance schedules do not exist. This is due to two key properties of tightly interdependent SDF graphs:

• Tight interdependence is “additive”: If Z_1 and Z_2 are two subsets of actors in an SDF graph such that $(Z_1 \cap Z_2)$ is non-empty, and the subgraphs associated with Z_1 and Z_2 are both tightly interdependent, then the subgraph associated with $(Z_1 \cup Z_2)$ is tightly interdependent. Thus each SDF graph G has a unique set of non-overlapping “maximal” tightly interdependent subgraphs, which are called the ***tightly interdependent components*** of G.

• Partitioning a loosely interdependent SDF graph G based on subindependence cannot decompose a tightly interdependent subgraph of G. Thus, if P_1, P_2 partition the actors of G such that P_1 is subindependent of P_2, and if T is a subset of actors whose corresponding subgraph is tightly interdependent, then $T \subseteq P_1$ or $T \subseteq P_2$.

Thus, if a loosely interdependent SDF graph is recursively decomposed based on subindependence, the decomposition process will always terminate on the same subgraphs — the tightly interdependent components.

7. Loose Interdependence Algorithms

This property of tightly interdependent subgraphs has been applied to develop a flexible scheduling framework for optimized compilation of SDF graphs. The scheduling framework is based on a class of uniprocessor scheduling algorithms that we call ***loose interdependence algorithms***. A loose interdependence algorithm consists of three component algorithms, which we call the ***acyclic scheduling algorithm***, the ***subindependence partitioning algorithm***, and the ***tight scheduling algorithm***. The *acyclic scheduling algorithm* is any algorithm for constructing single appearance schedules for acyclic SDF graphs; the *subindependence partitioning algorithm* is any algorithm that determines whether a strongly connected SDF graph is loosely interdependent and if so, finds a subindependent partition; and the *tight scheduling algorithm* is any algorithm that generates a valid schedule for a tightly interdependent SDF graph. The precise manner in which the three component sub-algorithms interact to define a loose interdependence algorithm is specified in [3].

The following useful properties of loose interdependence algorithms are established in [3].

- Any loose interdependence algorithm constructs a single appearance schedule when one exists.
- If N is an actor in the input SDF graph and N is not contained in a tightly interdependent component of G, then any loose interdependence algorithm schedules G in such a way that N appears only once.
- If N is an actor within a tightly interdependent component of the input SDF graph, then the number of times that N appears in the schedule generated by a loose interdependence algorithm is determined entirely by the tight scheduling algorithm.

The last property states that the effect of the tight scheduling algorithm is independent of the subindependence partitioning algorithm, and vice-versa. Any subindependence partitioning algorithm guarantees that there is only one appearance for each actor outside the tightly interdependent components, and the tight scheduling algorithm completely determines the number of appearances for actors inside the tightly interdependent components. For example, if we develop a new subindependence partitioning algorithm that is more efficient in some way (e.g. it is faster, or reduces buffering cost more), we can substitute it for any existing subindependence partitioning algorithm without changing the compactness of the resulting looped schedules. Similarly, if we develop a new tight scheduling algorithm that schedules any tightly interdependent graph more compactly than the existing tight scheduling algorithm, we are guaranteed that using the new algorithm instead of the old one will lead to more compact schedules overall.

8. Modem example

Figure 4(a) shows an SDF implementation of a modem taken from [17]. The repetitions vector is given by

$$\mathbf{q}(A, B, \ldots, P) = [16, 16, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1]^T.$$

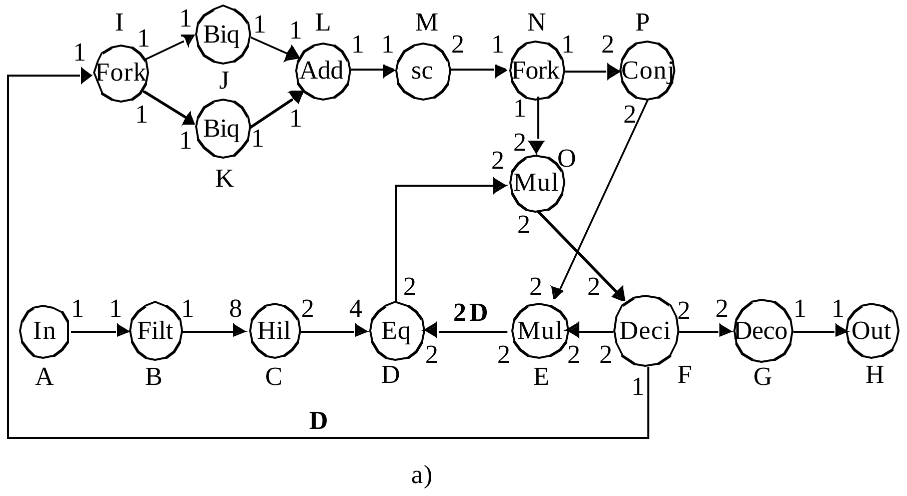
There is one strongly connected component, corresponding to actors

$$D, O, E, F, I, J, K, L, M, N, P\ .$$

This strongly connected component is clustered to give an acyclic graph as depicted in Figure 4(b). A possible single appearance schedule for this clustered graph is

$$(16A)(16B)(2C)\Omega_1 GH. \tag{5}$$

Now the strongly connected component has a subindependent partition given by $\{D, I\}$ and $\{O, E, F, J, K, L, M, N, P\}$. Since the subgraphs corresponding to these two subsets of actors



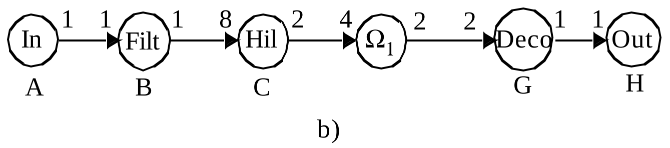


Figure 4. a) A block diagram of a modem application. b) Acyclic graph after clustering the strongly connected components in a).

are both acyclic, the recursive application of a loose interdependence algorithm terminates by applying the acyclic scheduling algorithm to each of the partitions, yielding the single appearance schedule $DIJKLM(2N)OPFE$ for this strongly connected component. This schedule is then substituted into the top-level schedule (5) to give a single appearance schedule for the entire graph:

$$(16A)(16B)(2C)DIJKLM(2N)OPFEGH.$$

9. Minimizing buffer memory

In the scheduling framework above, the acyclic scheduling algorithm can be designed such that the total buffer-memory requirement is minimized to a certain extent (which we will elaborate on later). In this section, we assume that the SDF graph is acyclic; the non-acyclic case will be dealt with later.

It is shown in [22] that the buffer-memory minimization problem is NP-complete, even for arbitrary, acyclic homogenous[1] SDF graphs. Two heuristics, along with a post-processing algorithm have been developed; these two algorithms are complementary in the sense that one performs well on graphs having a more regular topology and regular rate changes, while the other performs well on graphs having irregular topologies and irregular rate changes.

Essentially, for an acyclic graph, the problem of constructing a buffer-memory optimal single appearance schedule boils down to generating an appropriate topological ordering of the vertices in the graph, and then generating an optimal loop hierarchy. The number of topological sorts in an acyclic graph can be exponential in the size of the graph; for example, a complete bipartite graph with $2n$ actors has $(n!)^2$ possible topological sorts. Each topological sort gives a valid flat single appearance schedule (i.e, a single appearance schedule with no nested loops). The post-processing step then computes a buffer-memory optimal loop hierarchy. For example, the graph in Figure 5 shows a bipartite graph with 4 actors. The repetitions vector for the graph is given by $(12, 36, 9, 16)^T$, and there are 4 possible topological sorts for the graph. The flat schedule corresponding to the topological sort $ABCD$ is given by $(12A)(36B)(9C)(16D)$. This can be nested as $(3(4A)(3(4B)C))(16D)$, and this schedule has a buffer memory requirement of 208. The flat schedule corresponding to the topological sort $ABDC$, when nested optimally, gives

1. A homogenous SDF graph has $p(e) = c(e) = 1$ for all edges e.

the schedule $(4(3A)(9B)(4D))(9C)$, with a buffer memory requirement of 120.

The post-processing step of computing a loop hierarchy for a given actor ordering can be accomplished optimally for delayless graphs by using a dynamic programming algorithm [22], called the dynamic programming post-optimization (**DPPO**) algorithm. An extension of this algorithm, called generalized DPPO (**GDPPO**), has been developed to optimally handle actor orderings for SDF graphs that have delays and that may contain cycles [5]. Given any consistent SDF graph G, and an ordering L of the actors in G, GDPPO computes a single appearance schedule that minimizes the buffer memory requirement over all single appearance schedules that have the given actor ordering (assuming that at least one valid single appearance schedule exists that has the given actor ordering). Here, by the actor ordering of a single appearance schedule, we mean the lexical order in which the actors appear — for example, the actor ordering associated with the schedule $(4(3A)(9B)(4D))(9C)$ is (A, B, D, C). The running time of GDPPO algorithm on sparse SDF graphs is $O(|V|^3)$, where V is the set of vertices.

10. The Buffer Memory Lower Bound

In [2] the following lower bound on $max\_tokens(e, S)$ is derived, given a consistent SDF graph G, an edge e in G, and a valid single appearance schedule S.

Definition 1: The **buffer memory lower bound (BMLB)** of an SDF edge e, denoted $BMLB(e)$, is given by

$$BMLB(e) = \begin{cases} (\eta(e) + d(e)) \text{ if } (d(e) < \eta(e)) \\ d(e) \text{ if } (d(e) \ge \eta(e)) \end{cases}, \tag{6}$$

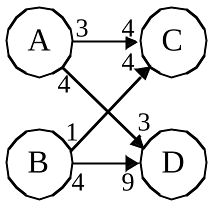


Figure 5. A bipartite SDF graph to illustrate the different buffer memory requirements possible with different topological sorts.

where $\eta(e) = \frac{p(e)c(e)}{gcd(\{p(e), c(e)\})}$.

If $G = (V, E)$ is an SDF graph, then

$$\left(\sum_{e \in E} BMLB(e)\right). \tag{7}$$

is called the BMLB of G, and a valid single appearance schedule S for G that satisfies $max\_tokens(e, S) = BMLB(e)$ for all $e \in E$ is called a **BMLB schedule** for G.

Not all consistent SDF graphs have valid BMLB schedules. For example the SDF graph of Figure 1(a) does not have a BMLB schedule. In contrast, for the SDF graph in Figure 1(b), it can easily be verified that the schedule $A(2B(3C))$, which has a buffer memory requirement of $3 + 3 = 6$, is a BMLB schedule.

Although BMLB schedules do not exist for all SDF graphs, empirical observations suggest that many practical graphs have BMLB schedules [5].

11. Pairwise Grouping of Adjacent Nodes

The first of the two heuristics that we discuss for generating topological orderings of acyclic SDF graphs with the objective of buffer memory minimization is a bottom-up procedure called **Acyclic Pairwise Grouping of Adjacent Nodes (APGAN)**. In this technique, a cluster hierarchy is constructed by clustering exactly two adjacent vertices at each step. At each clusterization step, a pair of adjacent actors is chosen that maximizes ρ_G over all adjacent pairs that are **clusterable**, which means that they do not introduce cycles in the graph when clustered.

Figure 6 illustrates the operation of APGAN. Figure 6(a) shows the input SDF graph. Here $\mathbf{q}(A, B, C, D, E) = (6, 2, 4, 5, 1)$, and for $i = 1, 2, 3, 4$, Ω_i represents the ith hierarchical actor instantiated by APGAN. The repetition counts of the adjacent pairs are given by

$$\rho(\{A, B\}) = \rho(\{A, C\}) = \rho(\{B, C\}) = 2 \text{, and} \tag{8}$$

$$\rho(\{C, D\}) = \rho(\{E, D\}) = \rho(\{B, E\}) = 1. \tag{9}$$

Thus, APGAN will select one of the three adjacent pairs $\{A, B\}$, $\{A, C\}$, or $\{B, C\}$ for its first clusterization step. The adjacent pair $\{A, C\}$ introduces a cycle when clustered, while the other two adjacent pairs do not introduce cycles. Thus, APGAN chooses arbitrarily between $\{A, B\}$ and $\{B, C\}$ as the first adjacent pair to cluster.

Fig. 6(b) shows the graph that results from clustering $\{A, B\}$ into the hierarchical actor Ω_1. In this graph, $\mathbf{q}(\Omega_1, C, D, E) = (2, 4, 5, 1)$, and it is easily verified that $\{\Omega_1, C\}$ uniquely maximizes ρ over all adjacent pairs. Since $\{\Omega_1, C\}$ does not introduce a cycle, APGAN selects this adjacent pair for its second clusterization step. Fig. 6(c) shows the resulting graph.

Figs. 6(d&e) show the results of the remaining two clusterizations in our illustration of APGAN. We define the **subgraph corresponding to Ω_i** to be the subgraph that is clustered in the ith clusterization step. A valid single appearance schedule for Fig. 6(a) can easily be constructed by recursively traversing the hierarchy induced by the subgraphs corresponding to the Ω_is. We start by constructing a schedule for the top-level subgraph, the subgraph corresponding to Ω_4. This yields the "top-level" schedule $(2\Omega_2)\Omega_3$ (we suppress loops that have an iteration count of one) for the subgraph corresponding to Ω_4. We continue in this manner to yield the valid single appearance schedule $S_p \equiv (2(3A)B(2C))(5D)E$ for Fig. 6(a).

From S_p and Fig. 6(a) it easily verified that $buffer\_memory(S_p)$ and $\left(\sum_{e \in E} BMLB(e)\right)$, where E is the set of edges in Fig. 6(a), are identically equal to 43, and thus in the execution of APGAN illustrated in Fig. 6, a BMLB schedule is returned.

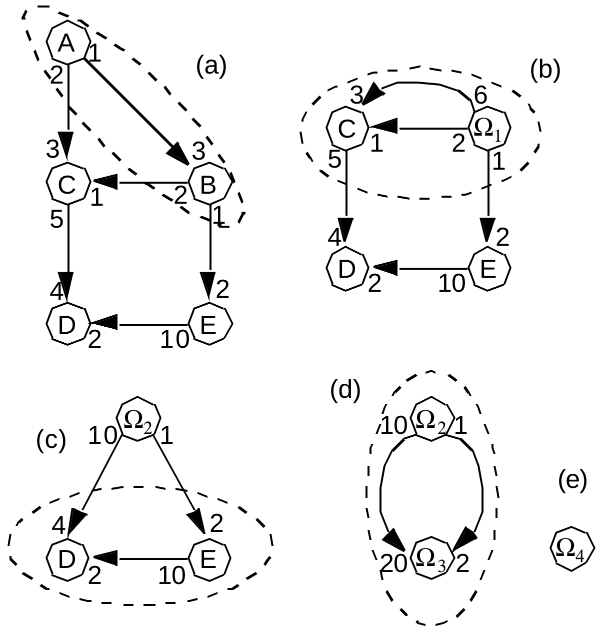


Figure 6. An illustration of APGAN.

The APGAN approach, as we have defined it here, does not uniquely specify the sequence of clusterizations that will be performed. The APGAN technique together with an unambiguous protocol for deciding between adjacent pairs that are tied for the highest repetition count form an **APGAN instance**, which generates a unique schedule for a given graph. We say that an adjacent pair is an **APGAN candidate** if it does not introduce a cycle, and its repetition count is greater than or equal to that of all other adjacent pairs that do not introduce cycles. Thus, an APGAN instance is any algorithm that takes a consistent, acyclic SDF graph, repeatedly clusters APGAN candidates, and then outputs the schedule corresponding to a recursive traversal of the resulting cluster hierarchy.

It is shown in [2] that APGAN is optimal for a class of acyclic SDF graphs in the following sense:

Theorem 3: If $G = (V, E)$ is a connected, acyclic SDF graph that has a BMLB schedule, $d(e) < \eta(e)$ for all $e \in E$, and P is an APGAN instance, then the schedule obtained by applying P to G is a BMLB schedule for G.

Hence, whenever the achievable lower bound on the buffer memory (that is, the buffer memory requirement of the single appearance schedule having the lowest possible buffer memory requirement) coincides with the BMLB, and inequality in the statement of Theorem 3 holds, APGAN will always find a BMLB schedule. If the achievable lower bound is greater than the BMLB, then the schedule returned by APGAN could have a buffer memory requirement greater than the achievable lower bound.

Many practical systems, such as QMF filter banks, fall into the category of SDF graphs that satisfy the conditions of Theorem 3 [2].

12. Recursive Partitioning by Minimum Cuts

APGAN constructs a single appearance schedule in a bottom-up fashion by starting with the innermost loops and working outward. An alternative approach, called **Recursive Partitioning by Minimum Cuts (RPMC)**, computes a schedule by recursively partitioning the SDF graph in such a way that outer loops are constructed before the inner loops. Each partition is constructed by finding the cut (partition of the set of actors) across which the minimum amount of data is transferred. The cut that is produced must have the property that all edges that cross the cut have the same direction. This is to ensure that all actors on the left side of the partition can be scheduled before any on the right side are scheduled. A constraint that the partition be fairly evenly sized is also imposed. This is to increase the possibility of having gcd's that are greater than unity for the repetitions of the actors in the subsets produced by the partition, thus reducing the buffer memory requirement [22].

Suppose that $G = (V, E)$ is a connected, consistent SDF graph. A **cut** of G is a partition of the actor set V into two disjoint sets V_L and V_R. The cut is **legal** if for all edges e crossing the cut (that is, all edges that have one incident actor in V_L and the other in V_R), we have $src(e) \in V_L$ and $snk(e) \in V_R$. Given a bounding constant $K \le |V|$, the cut results in bounded sets if it satisfies

$$|V_R| \le K, \; |V_L| \le K. \tag{10}$$

The weight of edge e is defined as $w(e) \equiv TNSE(e)$.

The weight of the cut is the total weight of all the edges crossing the cut. The problem then is to find the minimum weight legal cut into bounded sets for the graph. This problem is believed to be NP-complete, although a proof has not been discovered [22]. Kernighan and Lin [11] devised a heuristic procedure for computing cuts into bounded sets but they considered only undirected graphs. Methods based on network flows [8] do not work because the minimum cut given by the max-flow-min-cut theorem may not be legal and may not be bounded [22]. Hence, a heuristic solution is needed for finding legal minimum cuts into bounded sets.

One technique is to use the max-flow-min-cut theorem [8] to generate a minimum cut. Legality can be ensured by adding reverse edges (v, u) for each edge (u, v). The capacities of the reverse edges are set to infinity, ensuring that any edge that crosses the cut in the reverse direction is an edge of infinite capacity [19]. However, this cut may not be bounded. One way to make this cut bounded would be to simply examine actors on the side with the larger number of actors, and move those over to the other side that increase the cost the least, until the bound is satisfied.

Another technique for constructing legal minimum cuts into bounded sets is to examine the set of cuts produced by taking an actor and all of its descendants as the actor set V_R and the set of cuts produced by taking an actor and all of its ancestors as the set V_L. For each such cut, an optimization step is applied that attempts to improve the cost of the cut. Consider a cut produced by setting

$$V_L = (ancs(v) \cup \{v\}), V_R = V \setminus V_L \tag{11}$$

for some actor v, and let $T_R(v)$ be the set of independent, *boundary actors* of v in V_R. A **boundary** actor in V_R is an actor that is not the predecessor of any other actor in V_R. Following Kernighan and Lin [11], for each of these actors, we can compute the cost difference that results if the actor is moved into V_L. This cost difference for an actor a in $T_R(v)$ is defined to be the difference between the total weight of all input edges of a and the total weight of output edges of a. We then move those actors across that reduce the cost. We apply this optimization step for all cuts of the form $(ancs(v) \cup \{v\})$ and $(desc(v) \cup \{v\})$ for each actor v in the graph and take the best one as the minimum cut. Since there are $|V|$ actors in the graph, $2|V|$ cuts are examined. Moreover, the cut produced will have bounded sets since cuts that produce unbounded sets are discarded.

RPMC now proceeds by partitioning the graph by computing the legal minimum cut and forming the schedule $(\rho_G(V_L)S_L)(\rho_G(V_R)S_R)$, where S_L, S_R are schedules for G_L and G_R respectively that are obtained recursively by partitioning G_L and G_R. It can be shown that the running time of RPMC for sparse SDF graphs, including post-optimization by GDPPO, is $O(|V|^3)$ [22].

13. Application to general SDF topologies

The APGAN and RPMC algorithms work on acyclic SDF graphs, and thus are suitable for use as the acyclic scheduling algorithm in the scheduling framework described in Section 7. In this manner, we can obtain single appearance schedules for cyclic graphs that minimize buffer memory costs to a limited extent. In particular, if buffer-memory considerations are not taken into account in either the subindependence partitioning algorithm or the tight scheduling algorithm, there is no guarantee that the resulting schedule will be optimal or even near-optimal with respect to the buffer memory requirement. Combining buffer-memory considerations into the latter two components of the scheduling framework is an important topic for future work.

14. Experimental observations

APGAN and RPMC (with the second of the two legal bounded cut heuristics mentioned) have been tested on many practical examples, as well as randomly generated graphs. Many practical systems, such as QMF filter banks fall into the category of SDF graphs having BMLB schedules; hence, on these APGAN performs optimally. It is interesting to note that on non-uniform filter bank structures, the BMLB cannot be achieved, and on such structures, RPMC gives significantly better schedules than APGAN. Also, RPMC outperforms APGAN by almost 2 to 1 on random SDF graphs. Details of these experiments can be found in [2, 21]. It would be interesting to see the impact of using the first heuristic (based on the network flow formulation) for generating legal minimum cuts into bounded sets on RPMC performance; we have not done these experiments yet.

15. Application to multidimensional SDF graphs

The synchronous dataflow model suffers from the limitation that its streams are one-dimensional. For multidimensional signal processing algorithms, it is necessary to have a model in which this restriction is not present, so that effective use can be made of the inherent data-parallelism that exists in such systems. As for one-dimensional systems, the specification model for multidimensional systems should expose to the compiler or hardware synthesis tool as much static information as possible so that run-time decision making is avoided as much as possible, and so that effective use can be made of both functional and data parallelism. Most multidimensional signal processing systems also have a predictable flow of control, like one-dimensional systems, and for this reason, an extension of SDF, called multidimensional synchronous dataflow was proposed in [18].

Although a multidimensional stream can be embedded within a one dimensional stream, it may be awkward to do so [7]. In particular, compile-time information about the flow of control may not be immediately evident. The multidimensional SDF (MDSDF) model is a straightforward extension of one-dimensional SDF. Figure 7 shows a trivially simple two-dimensional SDF graph. The numbers of tokens produced and consumed are now given as M-tuples. Instead of one balance equation for each edge, there are now M. The balance equations for Figure 7 are

$$r_{A,1}O_{A,1} = r_{B,1}I_{B,1}, \; r_{A,2}O_{A,2} = r_{B,2}I_{B,2} \tag{12}$$

These equations should be solved for the smallest integers $r_{X,i}$, which then give the number of repetitions of each actor X in each dimension i.

As a simple application of MDSDF, consider a portion of an image coding system that takes a 40×48 pixel image and divides it into 8×8 blocks on which it computes a DCT. At the top level of the hierarchy, the dataflow graph is shown in Figure 8. The solution to the balance equations is given by

$$r_{A,1} = r_{A,2} = 1, \; r_{DCT,1} = 5, \; r_{DCT,2} = 6. \tag{13}$$

A segment of the index space for the stream on the edge connecting actor A to the DCT is shown in the Figure. The segment corresponds to one firing of actor A. The space is divided into regions of tokens that are consumed on each of the five vertical firings of each of the 6 horizontal

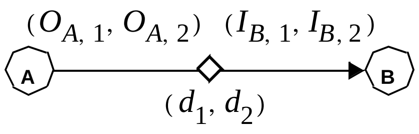


Figure 7. A simple MD-SDF graph.

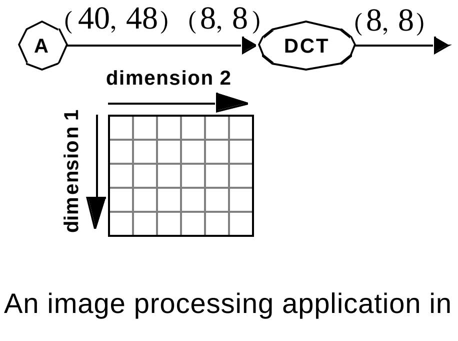


Figure 8. An image processing application in MD-SDF.

firings. The precedence graph constructed automatically from this shows that the 30 firings of the DCT are independent of one another, and hence can proceed in parallel. Distribution of data to these independent firings can be automated.

A delay in MD-SDF is associated with a tuple as shown in Figure 7. It can be interpreted as specifying boundary conditions on the index space. Thus, for 2D-SDF, as shown in the Figure, it specifies the number of initial rows and columns. It can also be interpreted as specifying the direction in the index space of a dependence between two single assignment variables, much as done in reduced dependence graphs [12].

All of the scheduling techniques discussed in the earlier sections of this paper extend to the MDSDF model. The extension of RPMC and GDPPO can be handled in a straightforward manner by simply changing the buffer cost formulation appropriately [21]. In the remainder of this section, we define an extension of the buffer memory lower bound to multidimensional systems, and we present a multidimensional version of the APGAN algorithm along with its associated optimality result (i.e. the MDSDF version of Theorem 3). In this discussion, we assume 2 dimensions for notational simplicity, unless otherwise stated. We use the notation $A_{[i,j]}$ to mean the (i,j) th invocation of actor A in a complete valid schedule. In an MDSDF schedule, a single appearance schedule such as $((4,2)A(6,4)B)$ corresponds to a loop structure of the form:

```
for x = 0 to 3
        for y = 0 to 1
            fire $A_{[x,y]}$
end fory, forx

for x = 0 to 5
        for y = 0 to 3
            fire $B_{[x,y]}$
end fory, forx.
```

15.1 The Buffer Memory Lower Bound (BMLB) for MDSDF graphs

The BMLB of an MDSDF graph can be computed in a manner similar to the SDF BMLB computation. First, we define

$$x(AB) = \frac{r_{A,1}}{gcd(r_{A,1}, r_{B,1})} O_{A,1}, y(AB) = \frac{r_{A,2}}{gcd(r_{A,2}, r_{B,2})} O_{A,2} , \qquad (14)$$

for an edge (A, B) with (d_1, d_2) delays. Then, the BMLB for the edge (A, B) can be expressed as [21]

$$BMLB(AB) = \begin{cases} (x(AB) + d_1)(y(AB) + d_2) & d_1 < x \vee d_2 < y \\ d_1 d_2 & d_1 \geq x \wedge d_2 \geq y \end{cases} . \tag{15}$$

15.2 APGAN for MDSDF graphs

APGAN can be applied to acyclic MDSDF graphs in the following manner [21]. First, define the following two quantities:

$$\rho_1(\{A, B\}) = gcd(r_{A,1}, r_{B,1}) \text{ and } \rho_2(\{A, B\}) = gcd(r_{A,2}, r_{B,2}) . \tag{16}$$

The clustering function is a tuple and is then given by

$$\rho(\{A, B\}) \equiv (\rho_1(\{A, B\}), \rho_2(\{A, B\})) . \tag{17}$$

At each step in the algorithm, we cluster the adjacent pair (A, B) that maximizes $\rho(\{A, B\})$ component-wise. This means that for any other adjacent clusterable pair $\{X, Y\}$, with $\rho'(\{X, Y\}) = (\rho'_1(\{X, Y\}), \rho'_2(\{X, Y\}))$ we should have $\rho_1 \geq \rho'_1, \rho_2 \geq \rho'_2$. If such a pair does not exist, we pick the adjacent clusterable pair $\{U, V\}$ that maximizes $\rho_1(\{U, V\})\rho_2(\{U, V\})$.

The following result extends the “APGAN optimality property” of Theorem 3 to the MDSDF version of APGAN defined above.

Theorem 4: When applied to a consistent MDSDF graph, APGAN will return a BMLB schedule whenever one exists, provided that the delay (d_1, d_2) on each edge (A, B) satisfies:

$$d_1 < x \vee d_2 < y$$

where

$$x = \frac{r_{A,1}}{gcd(r_{A,1}, r_{B,1})} O_{A,1}, y = \frac{r_{A,2}}{gcd(r_{A,2}, r_{B,2})} O_{A,2} .$$

15.3 MDSDF APGAN example

Consider the example graph shown in Figure 9. The repetitions vector is given by $r(A, B, C, D) = \{(2, 8), (6, 4), (4, 2), (1, 3)\}$. The clusterable pairs are $\{A, B\}$, $\{B, C\}$, and $\{C, D\}$. The clustering function values are $\rho(\{A, B\}) = (2, 4)$, $\rho(\{B, C\}) = (2, 2)$, and $\rho(\{C, D\}) = (1, 1)$. Hence, $\{A, B\}$ is the pair chosen for clustering since its clustering function has maximum component-wise value over the three clusterable pairs. Similarly, at the next step, there are two clusterable pairs, $\{W1, C\}$ and $\{C, D\}$, and the clustering function values are $\rho(\{W1, C\}) = (2, 2)$ and $\rho(\{C, D\}) = (1, 1)$. So $\{W1, C\}$ is clustered next, and the final schedule is $(2, 2)((1, 2)((1, 2)A(3, 1)B)(2, 1)C)(1, 3)D$. It can be verified that this is indeed a BMLB schedule.

The graph in Figure 10 shows an example where there is no adjacent pair whose clustering function has the maximum-componentwise value. Hence, the graph does not have a BMLB schedule either, as is verified by looking at the two possible nested single appearance schedules. The repetitions vector is given by $\{(4, 5), (6, 15), (9, 3)\}$. The clustering function values for the two clusterable pairs are $\rho(\{A, B\}) = (2, 5)$ and $\rho(\{B, C\}) = (3, 3)$. The two possible nested single appearance schedules are

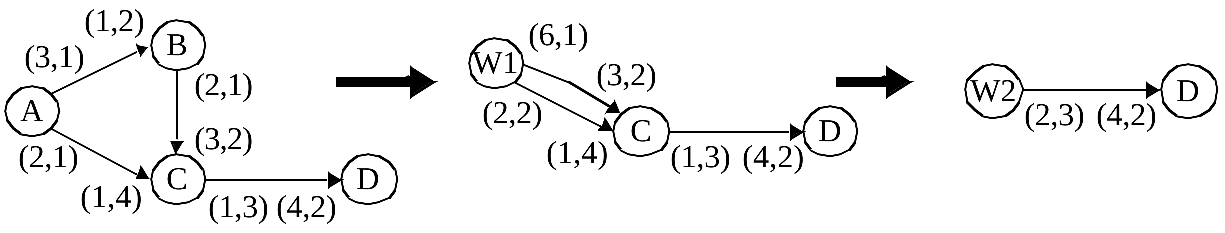


Figure 9. An MDSDF graph that has a BMLB schedule.

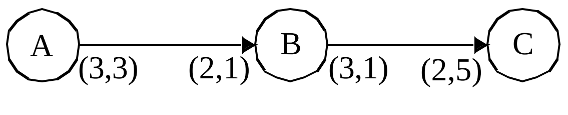


Figure 10. An example of a graph that does not have a BMLB schedule.

$(2,5)((2,1)A(3,3)B)\quad(9,3)C$ and $(4,5)A\quad(3,3)((2,5)B\quad(3,1)C)$.

Neither of these is a BMLB schedule. The APGAN algorithm in this case will choose to cluster $\{A, B\}$ first because $2 \times 5 > 3 \times 3$; this results in the first of the two schedules given above. The first schedule has higher buffering requirements than the second; hence, APGAN is not optimal when the graph does not have a BMLB schedule.

16. Alternative approaches for scheduling SDF graphs

The techniques in this paper focus on compiling SDF graphs to minimize the code size and data memory size. At the Aachen University of Technology, as part of the COSSAP software synthesis environment for DSP, Ritz et al. have investigated the minimization of code size in conjunction with a different secondary optimization criterion: minimization of the context-switch overhead, or the average rate at which **actor activations** occur [25]. An actor activation occurs whenever two distinct actors are invoked in succession; for example, the schedule $(2(2B))(5C)$ results in five activations per schedule period.

In multiprocessor computers, different iterations of a loop can be executed in parallel on different processors. To achieve this, the code for the loop is replicated across the processors. This is in contrast to our problem, which involves a uniprocessor implementation target, and in which there are no explicitly specified loops (within the schedule period). We would like to detect the opportunity to construct multiple invocations of the same firing sequence, and we wish to group these invocations successively in time so that they form successive iterations of a single loop.

Loop distribution and loop fusion [29] can be used to improve data locality for looped schedules of SDF graphs. Also, the use of iteration space tiling, as discussed in [28, 29], can be used to improve locality for code synthesized for a looped schedule of an SDF graph. However, each loop transformation and schedule rearrangement applies to a localized section of the target code. The scheduling techniques described in this paper use dataflow properties to guide a scheduler to more efficient solutions; loop transformations can then be applied to refine the resulting schedules. We believe that this will be more efficient than constructing naive schedules, and relying solely on loop transformations to achieve adequate data locality.

Ade, Lauwereins, and Peperstraete develop upper bounds on the minimum buffer memory requirement for certain classes of SDF graphs [1]. Since these bounds attempt to minimize over all valid schedules, and since single appearance schedules generally have much larger buffer memory requirements than schedules that are optimized for minimum buffer memory only, these bounds cannot consistently give close estimates of the minimum buffer memory requirement for single appearance schedules.

Lauwereins, Wauters, Ade, and Peperstraete present a generalization of SDF called cyclo-static dataflow [13]. A major advantage of cyclo-static dataflow is that it can eliminate large amounts of token traffic arising from the need to generate dummy tokens in corresponding (pure) SDF representations. Although cyclostatic dataflow can reduce the amount of buffering for graphs having certain multirate actors like explicit downsamplers, it is not clear whether this model can in general be used to derive schedules that are as compact as single appearance schedules for pure SDF graphs but have lower buffering requirements than those arising from the techniques discussed in this paper.

A linear programming framework for minimizing the memory requirement of a synchronous dataflow graph in a parallel processing context is explored by Govindarajan and Gao in [10]. Here the goal is to minimize the buffer cost without sacrificing throughput — just as one of the goals in this paper is to minimize buffering cost without sacrificing code compactness.

17. Summary

This paper has reviewed a set of techniques for mapping SDF programs for embedded digital signal processing applications into efficient implementations on programmable processors. The techniques have focused on the minimization of code size, and the minimization of the memory required for the buffers that implement the edges in the input dataflow graph. Even though some of the associated problems have been shown to be NP-complete, we have described algorithms that solve subsets of these problems optimally, and have described a framework in which these algorithms can be combined with heuristics to give a comprehensive solution.

There are two central themes that underlie the techniques discussed in this paper. These themes are based on the concept of single appearance schedules, which is a class of code-size-minimizing schedules for SDF programs. The first theme is a uniprocessor scheduling framework that operates by decomposing the input SDF graph into a hierarchy of acyclic subgraphs. The scheduling framework constructs single appearance schedules whenever they exist, and when single appearance schedules do not exist, the framework guarantees optimal code size for all actors that are not contained in a certain type of subgraph called tightly independent subgraphs. The second theme involves a pair of complementary algorithms that construct single appearance schedules for acyclic SDF graphs that minimize the buffer memory requirement. These complementary algorithms can easily be incorporated into the scheduling framework to handle the acyclic graphs that result from the decomposition process.

These techniques have all been implemented in the Ptolemy software environment [6]. Additionally, APGAN, DPPO, and the scheduling framework based on loose interdependence algorithms have been implemented by the Alta Group of Cadence in the Signal Processing Worksystem, a widely-used design environment for DSP applications. A detailed, comprehensive treatment of the techniques discussed in this paper, including complete pseudocode specifications of the algorithms, can be found in [5].

18. References

[1] M. Ade, R. Lauwereins, and J. A. Peperstraete, “Buffer Memory Requirements in DSP Applications,” presented at *IEEE Workshop on Rapid System Prototyping*, Grenoble, June, 1994.

[2] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, “APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations,” *Journal of Design Automation for Embedded Systems*, January, 1997.

[3] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, “Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms,” *IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications*, Vol. 42, No. 3, pp. 138-150, March, 1995.

[4] S. S. Bhattacharyya and E. A. Lee, “Looped Schedules for Dataflow Descriptions of Multirate Signal Processing Algorithms,” *Journal of Formal Methods in System Design*, Vol. 5, No. 3, December, 1994.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Ma, 1996.

[6] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *International Journal of Computer Simulation*, Vol. 4, April, 1994.

[7] M. C. Chen, “Developing a Multidimensional Synchronous Dataflow Domain in Ptolemy”, MS Report, UC Berkeley, June 1994.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability-A guide to the theory of NP-completeness*, Freeman, 1979.

[10] S. R. Govindarajan, G. R. Gao, and P. Desai, “Minimizing Memory Requirements in Rate-Optimal Schedules,” *Proceedings of the International Conference on Application Specific Array Processors,* August, 1994.

[11] B. W. Kernighan and S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs,” *Bell System Technical Journal*, Vol. 49, (No.2):291-308, February, 1970.

[12] P.S. Y. Kung, *VLSI Array Processors*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[13] R. Lauwereins, P. Wauters, M. Ade, and J. A. Pererstraete, “Geometric Parallelism and Cyclo-Static Dataflow in GRAPE-II,” *IEEE Workshop on Rapid System Prototyping*, June, 1994.

[14] R. Lauwereins, M. Engels, J. A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren, “GRAPE: A CASE Tool for Digital Signal Parallel Processing,” *IEEE ASSP Magazine*, Vol. 7, (No.2):32-43, April, 1990.

[15] E. A. Lee and T. M. Parks, “Dataflow Process Networks,” *Proceedings of the IEEE*, Vol. 83, No. 5, May, 1995.

[16] 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, (No.11):1751-62, November, 1989.

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

[18] E. A. Lee, “Multidimensional Streams Rooted in Dataflow,” *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grained Parallelism*, Orlando, January, 1993.

[19] S Majumdar, S. C. Nandy, B.B.Bhattacharyya, “Partitioning VLSI Floorplans by Staircase Channels for Global Routing,” VLSI Design 1998, Chennai, India, January 1998.

[20] Marwedel, G. Goossens (editors), *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

[21] P. K. Murthy, “Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow,” Ph.D. Thesis, Technical Memorandum UCB/ERL M96/79, Electronics Research Laboratory, University of California, Berkeley, Ca 94720, December 1996.

[22] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, “Combined Code and Data Minimization for Synchronous Dataflow Programs,” *Journal of Formal Methods in System Design*, July, 1997.

[23] D. R. O’Hallaron, *The Assign Parallel Program Generator*, Memorandum CMU-CS-91-141, School of Computer Science, Carnegie Mellon University, May, 1991.

[24] S. Ritz, S. Pankert, and H. Meyr, “High Level Software Synthesis for Signal Processing Systems,” *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, pp. 679-93, August, 1992.

[25] S. Ritz, S. Pankert, and H. Meyr, “Optimum Vectorization of Scalable Synchronous Dataflow Graphs”, technical report IS2/DSP93.1a, Aachen University of Technology, Germany, January, 1993.

[26] P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, 1993.

[27] M. Veiga, J. Parera, and J. Santos, “Programming DSP Systems on Multiprocessor Architectures,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Albuquerque, pp. 965-8 Vol. 2, April, 1990.

[28] M. E. Wolf and M. S. Lam, “A Data Locality Optimizing Algorithm”, *ACM Conference on Programming Language Design and Implementation*, San Francisco, California, June, 1991.

[29] M. Wolfe, “Optimizing Supercompilers for Supercomputers”, MIT Press, 1989.

[30] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers,* ACM Press, 1990.

[31] V. Zivojnovic, H. Schraut, M. Willems, and R. Schoenen, “DSPs, GPPs, and Multimedia Applications — An Evaluation Using DSPStone,” *Proceedings of ICSPAT,* November, 1995.