Proceedings of the ASAP 97 conference in Zurich, Switzerland, July 1997.

Optimized Software Synthesis for Synchronous Dataflow

Shuvra S. Bhattacharyya, Hitachi America
Praveen K. Murthy, University of California at Berkeley
Edward A. Lee, University of California at Berkeley

1/21/1997

Abstract

This paper reviews a set of techniques for compiling dataflow-based, graphical programs for embedded signal processing applications into efficient implementations on programmable digital signal processors. This is a critical problem because programmable digital signal processors have very limited amounts of on-chip memory, and the speed and power penalties for using off-chip memory are often prohibitively high for the types of applications, typically embedded systems, where these processors are used. Moreover, off-chip memory typically needs to be static, increasing the system cost considerably.

The compiling techniques described in the paper are developed for the synchronous dataflow model of computation, a model that has found widespread use for specifying and prototyping DSP systems.

1: Introduction

Rapid prototyping environments such as those described in [6], [12], [19], and [18] support code-generation for programmable digital signal processors (PDSP) used in embedded systems. Traditionally, PDSPs 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 on such a PDSP is severely limited, it is imperative that the generated code be parsimonious in its memory usage. Adding off-chip memory is often infeasible 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 [25]. 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 (SDF) [15] to specify the system, and to do code-generation starting from this specification. One reason that a compiler for a block diagram language is likely to give better performance than a compiler for an imperative language is because the underlying model of computation often imposes restrictions on the control flow of the specification, and this can be profitably exploited by the compiler.

SDF [15] 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 residing on that arc (called **delays**).

The code-generation strategy followed in many block diagram 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 inserts the machine instructions necessary for the computation specified by each actor it encounters; these instructions are obtained from a predefined library of actor code blocks. We 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. By “compile an SDF graph”, we mean exactly the strategy described above for generating a software implementation from an SDF graph specification of the system in the block diagram environment.

A key problem that arises in this strategy is code-size explosion since 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 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.

The predefined actor code blocks in the library can either be hand-optimized assembly language (feasible since the actors are usually small, modular components), or it can be an imperative language specification that is compiled by a compiler. As already mentioned, a compiler for an imperative language cannot usually exploit the restrictions in the overall control flow of the system. However, the code blocks within an actor are usually much simpler, and may even correspond to basic blocks that compilers *are* adept at handling. Hence, compiling an SDF graph using the methods we describe in this paper does not preclude the use of a good imperative language compiler; we expect this hybrid approach to eventually produce code competitive to hand-written code, as compiler technology improves. However, in this paper, we only consider the code and buffer memory optimization possible at the SDF graph level.

2: Synchronous dataflow (SDF)

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 and $snk(e)$.

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 [15], 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 often suppress the subscript if G is understood).

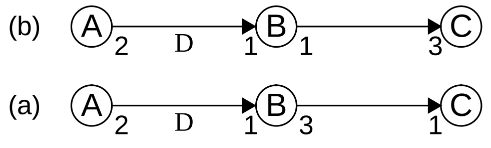


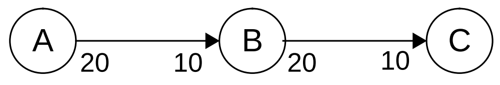
Figure 1. Examples of SDF graphs.

3: Constructing memory-efficient loop structures

This section informally outlines the interaction between the construction of periodic 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 periodic 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 periodic schedule, B must be invoked twice for every invocation of A. Simi-

(a)



Periodic Schedules

(b)

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.

larly, the mismatch on the right edge implies that we must invoke C twice for every invocation of B.

Figure 2(b) shows four possible periodic 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 overhead is small, particularly in many programmable DSPs, 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 arcs 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 arc, are 50, 40, 60, and 50 respectively.

Note that this model of buffering — maintaining a separate memory buffer for each data flow arc — is convenient and natural for code generation, and it is the model used, for example, in the SDF-based code generation described in [6], [12], [19]. More technical advantages of this buffering model are elaborated on in [17].

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 further 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. A practical example to illustrate this will be given in the final paper.

4: Notation

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

$$TNSE(e) = \mathbf{q}_G(src(e)) \times p(e). \qquad \textbf{(1)}$$

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

For Fig. 1(a), $\mathbf{q}(A, B, C) = (3, 6, 2)$, $TNSE((A, B)) = 6$, and one valid schedule is $B(2AB)CA(3B)C$.

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 Fig. 1(a), if $S_1 = (3A)(6B)(2C), S_2 = (3A(2B))(2C)$, 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

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

Thus, $buffer\_memory(S_1) = 7 + 6 = 13$ 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]

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

5: Subindependence

The following useful facts have been established concerning the existence of a single appearance schedule for a given SDF graph [4].

- 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 arc α directed from a member of P_2 to a member of P_1, $d(\alpha) \ge c(\alpha)\mathbf{q}(snk(\alpha))$.

This form of precedence-independence is referred to as **subindependence**. Thus a strongly connected SDF graph has a single appearance schedule only if its nodes can be partitioned into 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 loose interdependence to single appearance schedules [3]:

Theorem 1: 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.

Thus, partitioning loosely interdependent SDF graphs defines a decomposition process for hierarchically scheduling SDF graphs that leads to single appearances schedules whenever they exist.

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 nodes in an SDF graph such that $(Z_1 \cap Z_2)$ is nonempty, 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 nonoverlapping "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 nodes 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.

6: 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.

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

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 node 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.

7: 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 was shown in [17] that the buffer-memory minimization problem is NP-complete, even for arbitrary, acyclic homogenous[1] SDF graphs. Hence, heuristic techniques have to be used. Two heuristics, along with a post-processing algorithm have been developed; these two algorithms are complimentary 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 $2n$ nodes 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 3 shows a bipartite graph with 4 nodes. 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 the schedule $(4(3A)(9B)(4D))(9C)$, with a buffer memory requirement of 120.

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

The post-processing step can be accomplished optimally by using a dynamic programming algorithm [17]. The running time of this algorithm on sparse SDF graphs is $O(|V|^3)$, where V is the set of vertices. We refer to this post-processing algorithm as **DPPO**.

7.1 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

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

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

If $G = (V, E)$ is an SDF graph, then $\left(\sum_{e \in E} BMLB(e)\right)$ 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.

7.2 APGAN

The first of the two heuristics 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

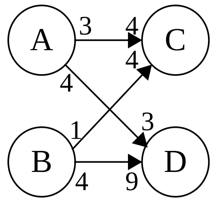


Figure 3. A bipartite SDF graph to illustrate the different buffer memory requirements possible with different topological sorts.

chosen that maximizes ρ_G over all adjacent pairs that don't introduce cycles in the graph when clustered.

Fig. 4 illustrates the operation of APGAN. Fig. 4(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 $\rho(\{A, B\}) = \rho(\{A, C\}) = \rho(\{B, C\}) = 2$, and $\rho(\{C, D\}) = \rho(\{E, D\}) = \rho(\{B, E\}) = 1$. Thus, APGAN will select one of the three adjacent pairs $\{A, B\}$, $\{A, C\}$, or $\{B, C\}$ for its first clusterization step. $\{A, C\}$ 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. 4(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. 4(c) shows the resulting graph.

Fig.s 4(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. 4(a) can easily be constructed by recursively traversing the hierarchy induced by the subgraphs corresponding to the Ω_i s. 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. 4(a).

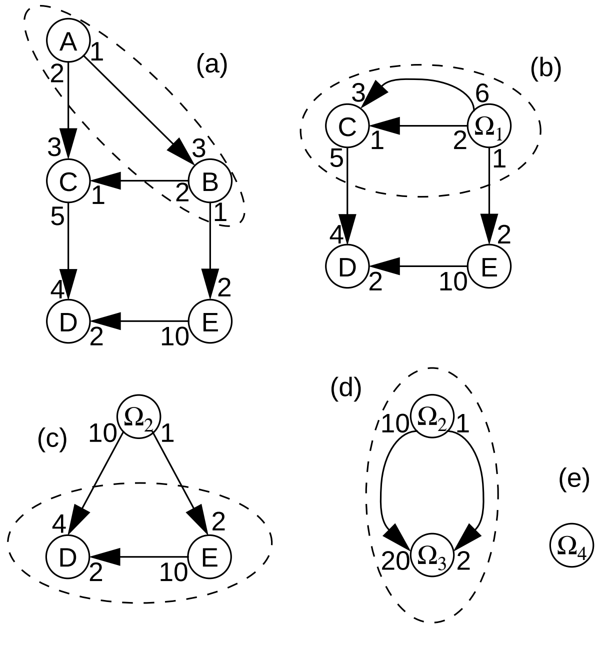


Figure 4. An illustration of APGAN.

From S_p and Fig. 4(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. 4(a), are identically equal to 43, and thus in the execution of APGAN illustrated in Fig. 4, a BMLB schedule is returned.

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 way:

Theorem 2: [2] 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 the other conditions of Theorem 2 hold, APGAN will find the 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.

7.3 RPMC

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 the 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 [17].

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

$$|V_R| \le K, |V_L| \le K. \qquad \textbf{(2)}$$

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 [17]. Kernighan and Lin [10] devised a heuristic procedure for computing cuts into bounded sets but they considered only undirected graphs. Methods based on network flows [7] do not work because the minimum cut given by the max-flow-min-cut theorem may not be legal and may not be bounded [17]. Hence, a heuristic solution is needed for finding legal minimum cuts into bounded sets.

RPMC examines 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

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

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 [10], 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 DPPO, is $O(|V|^3)$ [17].

7.4 Non-acyclic SDF graphs

The above algorithms work on acyclic SDF graphs, and thus are suitable for use as the acyclic scheduling component in the scheduling framework described in Section 6. Hence, buffer-optimal single appearance schedules for cyclic graphs can be obtained to a limited extent in this manner. But because buffer-memory is not taken into account in either the subindependence partitioning algorithm, or the tight scheduling algorithm, as described in Section 6, there is no guarantee that the resulting schedule will be buffer-memory optimal. Combining buffer-memory considerations into the latter two components in the scheduling framework is a possibility for future work.

7.5 Examples and Experiments

APGAN and RPMG have been tested on many practical examples, as well as randomly generated graphs. Many practical systems, such as QMF filterbanks 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 filterbank structures, the BMLB cannot be achieved, and on such structures, RPMC gives better schedules than APGAN. RPMC outperforms APGAN by almost 2 to 1 on random SDF graphs. Details of this study can be found in [2, 17].

8: 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 [Ritz93]. 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. Activation overhead includes saving the contents of registers that are used by the next actor to invoke, if necessary, and loading state variables and buffer pointers into registers.

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 [23] can be used to improve data locality for looped schedules of SDF graphs. Also, the use of iteration space tiling, as discussed in [22, 23], 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. Our clustering scheme uses 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.

In [1], Ade, Lauwereins, and Peperstraete develop upper bounds on the minimum buffer memory requirement for certain classes of SDF graphs. 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.

In [11], Lauwereins, Wauters, Ade, and Peperstraete present a generalization of SDF called *cyclo-static dataflow.* 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 down samplers, it is not clear whether this model can in general be used to get schedules that are as compact as single appearance schedules for pure SDF graphs but have lower buffering requirements than those arising from the techniques given 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 [9]. 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.

9: 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 arcs in the input dataflow graph. Even though some of the associated problems are NP-complete, we have described algorithms that solve subsets of these problems optimally, and have described the manner in which these 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 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 complimentary algorithms that construct single appearance schedules for acyclic SDF graphs that minimize the buffer memory requirement. These complimentary 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]. A detailed, comprehensive treatment of the techniques discussed in this paper, including complete pseudocode specifications of the algorithms, can be found in [5].

10: 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*, to appear.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing,” *IEEE Transactions on Computers*, vol.C-36, (no.1):24-35, January, 1987.

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

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

[18] J. Pino, S. Ha, E. A. Lee, and J. T. Buck, “Software Synthesis for DSP Using Ptolemy,” invited paper in *Journal of VLSI Signal Processing*, January, 1995.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

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