Proc. 1995 IEEE International Workshop on Rapid Systems Prototyping,
Chapel Hill, North Carolina, June, 1995.

Converting Graphical DSP Programs into Memory Constrained Software Prototypes

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

Abstract

Since software prototypes of DSP applications are most efficient when their code and data space requirements can be accommodated entirely within the on-chip memory of the target processor, it is crucial to employ efficient memory-minimizing compilation techniques in a DSP software prototyping system. In this paper, we introduce two techniques for the combined minimization of code and data when compiling graphical programs that are based on the synchronous dataflow (SDF) model.

The first method is a customization to acyclic graphs of a bottom-up technique, called Pairwise Grouping of Adjacent Nodes (PGAN), that was proposed earlier for general SDF graphs. We show that our customization significantly reduces the complexity of the general PGAN algorithm and performs optimally for a certain class of applications. The second approach is a top-down technique, called Recursive Partitioning by Minimum Cuts (RPMC), that is based on a generalized minimum cut operation. From an extensive experimental study, we conclude that RPMC and our customization of PGAN are complementary, and both should be incorporated into SDF-based prototyping environments in which the minimization of memory requirements is important.

1: Introduction

For the past several years, programmable digital signal processors have been popular in rapid prototyping environments for DSP. The limited on-chip memory of these processors, together with significant speed penalties for off-chip memory access, often render it critical for a high level software synthesis tool to produce a lean target program, in the dimensions of both code and data. In this paper, we present efficient techniques to compile graphical DSP programs based on the synchronous dataflow (SDF) model into software implementations that require a minimum amount of memory for code and data. Numerous DSP design environments, including a number of commercial tools, support SDF or closely related models [9, 12, 13, 14]. Here, we focus on programs that are represented as *acyclic* SDF graphs.

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.

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 [10], efficient algorithms are presented to determine whether or not a given SDF graph

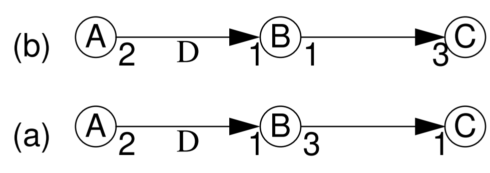

Figure 1. Examples of SDF graphs.

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).

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 (consumed from) 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 $B(2AB)CA(3B)C$. Here, a parenthesized term $(nS_1S_2\ldots S_k)$ specifies n successive firings of the "subschedule" $S_1S_2\ldots S_k$, and we may translate such a term into a loop in the target code. This notation naturally accommodates nested loops. Each parenthesized term $(nS_1S_2\ldots S_k)$ is called a **schedule loop**. A **looped schedule** is a finite sequence $V_1V_2\ldots V_k$, where each V_i is either an actor or a schedule loop. Henceforth in this paper, by a "schedule" we mean a "looped schedule."

A more compact valid schedule for Fig. 1(a) is $(3A)(2(3B)C)$. We call this schedule a **single appearance schedule** since it contains only one lexical appearance of each actor. To a good first approximation, any valid single appearance schedule gives the minimum code space cost for in-line code generation.

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 $buffer\_memory(S_2) = 3 + 6 = 9$.

In this paper we address the problem of computing a valid single appearance schedule that minimizes the buffer memory requirement over all valid single appearance schedules. We call such a schedule an **optimal schedule**.

2: Background

Given an SDF graph $G = (V, E)$, actor X is a **predecessor** of actor Y if there is an $e \in E$ such that $src(e) = X$ and $snk(e) = Y$, and X is a **successor** of Y if Y is a predecessor of X. Two actors X, Y are **adjacent** if X is a predecessor or successor of Y, and if X, Y are distinct, then $\{X, Y\}$ is an **adjacent pair**. If there is a path from $X \in V$ to $Y \in V$, then X is an **ancestor** of Y, and Y is a **descendant** of X. If X is neither a descendant nor an ancestor of Y, X is **independent of** Y. Also, $ancs(X)$ denotes the set of ancestors of X, $desc(X)$ denotes the set of descendants of X, and if $Z \subseteq V$, then $subgraph(Z)$ denotes the subgraph associated with Z.

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

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

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

Given a connected, consistent SDF graph $G = (V, E)$, a subset $Z \subseteq V$, and an actor $\Omega \notin V$, **clustering** Z **into** Ω means generating the new SDF graph (V', E') such that $V' = V - Z + \{\Omega\}$ and $E' = E - (\{e | (src(e) \in Z) \textbf{ or } (snk(e) \in Z)\}) + \hat{E}$, where $\hat{E}$ is a "modification" of the set of edges that connect actors in Z to actors outside of Z. If for each $e \in E$ such that $src(e) \in Z$ and $snk(e) \notin Z$, we define e' by

$$src(e') = \Omega, snk(e') = snk(e), d(e') = d(e) ,$$
$$p(e') = \mathbf{q}_G(src(e))\, p(e) / \rho_G(Z), c(e') = c(e) ,$$

and similarly for each $e \in E$ such that $snk(e) \in Z$ and $src(e) \notin Z$, we define e' by

$$snk(e') = \Omega, src(e') = src(e), d(e') = d(e) ,$$
$$p(e') = p(e), c(e') = \mathbf{q}_G(snk(e))\, c(e) / \rho_G(Z) ,$$

then,

$$\hat{E} \equiv \{e' | (src(e) \in Z, snk(e) \notin Z) \textbf{ or } (snk(e) \in Z, src(e) \notin Z)\} .$$

For each $e \in \hat{E}$, we say that e' **corresponds to** e and vice versa (e corresponds to e'). The graph that results from clustering Z into Ω in G is denoted $clust_G(Z, \Omega)$, or simply $clust_G(Z)$. Intuitively, an invocation of Ω in $clust_G(Z, \Omega)$ corresponds to one invocation of the subsystem $subgraph(Z)$. We say that Z is **clusterable** if $clust_G(Z)$ is consistent, and if G is acyclic, then Z **introduces a cycle** if $clust_G(Z)$ contains one or more cycles. Fig. 2 gives an example of clustering. Here, edge (D, Ω) corresponds to (D, C) and vice versa.

In [2], a dynamic programming algorithm called GDPPO is developed for post-optimizing single appearance schedules to reduce the buffer memory requirement. In the remainder of this paper, we discuss two heuristics for constructing single appearance schedules, and we report on an experimental study that compares these heuristics — with their schedules post-processed by GDPPO

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

— against each other and against randomly generated schedules that are post-processed by GDPPO.

3: The Buffer Memory Lower Bound

In [2] we derive the following lower bound on $max\_tokens(e, S)$, 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.

In Fig. 1(a), $BMLB((A, B)) = 3$, and $BMLB((B, C)) = 3$. Thus, a valid single appearance schedule for Fig. 1(a) is a BMLB schedule iff its buffer memory requirement is 6. It is easily verified that no such single appearance schedule exists for this graph. In contrast, for Fig. 1(b), it is easily verified that $A(2B(3C))$ is a BMLB schedule (here $\mathbf{q}(A, B, C) = (1, 2, 6)$).

4: PGAN for Acyclic Graphs

In the original *Pairwise Grouping of Adjacent Nodes (PGAN)* technique, developed in [3], 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 clusterable adjacent pairs.

To check whether or not an adjacent pair is clusterable, PGAN maintains the cluster hierarchy on the *acyclic precedence graph (APG)* [10]. Each vertex of the APG corresponds to an actor invocation, and each edge (x, y) signifies that at least one token produced by x is consumed by y in a valid schedule. PGAN determines whether or not an adjacent pair is clusterable by checking whether or not its consolidation introduces a cycle in the APG. This check is performed efficiently by applying a *reachability matrix*, which indicates for any two APG vertices x, y, whether or not there is a path from x to y.

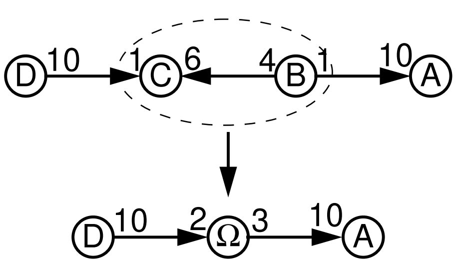

Figure 2. An example of clustering.

Unfortunately, the cost to compute and store the APG reachability matrix can be prohibitively high for some applications [2]. Since a large proportion of DSP applications that are amenable to the SDF model can be represented as acyclic SDF graphs, we propose an adaptation of PGAN to acyclic graphs, called **Acyclic PGAN (APGAN)**, that maintains the cluster hierarchy and reachability matrix directly on the input SDF graph rather than on the APG.

In an acyclic SDF graph G, it is easily verified that a subset Z of actors is not clusterable only if Z introduces a cycle. This condition is easily checked given a reachability matrix for G [2]. Since the existence of a cycle in $clust_G(Z, \Omega)$ is not a sufficient condition for Z not to be clusterable, the clusterability test that we apply in APGAN is not *exact*; it must be viewed as a conservative test. For some graphs, this imprecision can prevent APGAN from attaining optimal results [2]. In exchange for some degree of suboptimality in these cases, our clusterability test attains a large computational savings over the exact test based on the reachability matrix of the APG.

Fig. 3 illustrates the operation of APGAN. Fig. 3(a) shows the input SDF graph. Here $\mathbf{q}(A, B, C, D, E) = (6, 2, 4, 5, 1)$, and for

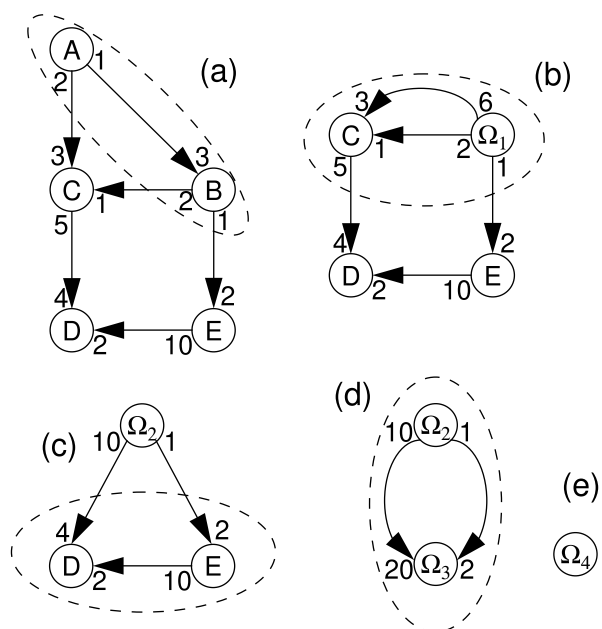

Figure 3. An illustration of APGAN.

$i = 1, 2, 3, 4$, 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. Clearly, $\{A, C\}$ introduces a cycle, 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. 3(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. 3(c) shows the resulting graph.

Fig.s 3(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. Thus, for example, the subgraph corresponding to Ω_2 consists of actors Ω_1 and C, and the two edges directed from Ω_1 to C. A valid single appearance schedule for Fig. 3(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. The subgraph G_i corresponding to each Ω_i consists of only 2 actors X_i and Y_i, and all edges in G_i are directed from X_i to Y_i. If each edge e in G_i satisfies $d(e) \geq \eta(e)$, then we construct the schedule

$$\left(\mathbf{q}_{G_i}(Y_i)\, Y_i\right)\left(\mathbf{q}_{G_i}(X_i)\, X_i\right),$$

and otherwise we construct

$$\left(\mathbf{q}_{G_i}(X_i)\, X_i\right)\left(\mathbf{q}_{G_i}(Y_i)\, Y_i\right).$$

It is straightforward to show that the resulting schedule for the subgraph corresponding to Ω_i is always optimal [2]. In Fig. 3, 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.

Next, we recursively descend one level in cluster hierarchy to the subgraph corresponding to Ω_3, and we obtain the schedule $(5D)\,E$. This “flattened schedule” then replaces its corresponding hierarchical actor in the top-level schedule, and the top-level schedule becomes $(2\Omega_2)\ (5D)\,E$. Next, descending to Ω_2, we construct the schedule $\Omega_1\,(2C)$. We then examine the subgraph corresponding to Ω_1 to obtain the schedule $(3A)\,B$. Substituting this for Ω_1, the schedule for the subgraph corresponding to Ω_2 becomes $(3A)\,B\,(2C)$. Finally, this schedule gets substituted for Ω_2 in the top-level schedule to yield the valid single appearance schedule $S_p \equiv (2\,(3A)\,B\,(2C))\ (5D)\,E$ for Fig. 3(a).

From S_p and Fig. 3(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. 3(a), are identically equal to 43, and thus in the execution of APGAN illustrated in Fig. 3, 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.

The following lemma is easily understood from our discussion of Fig. 3.

Lemma 1: Suppose G is a connected, consistent, acyclic SDF graph such that $d(e) < \eta(e)$ for each $e \in E$; P is an APGAN instance; and S is the schedule that results when P is applied to G. Then

$$buffer\_memory(S) = \sum_{e' \in E_{\Omega}} BMLB(e'),$$

where E_{Ω} is the set of edges that are contained the subgraphs corresponding to the hierarchical actors instantiated by P.

5: Deriving BMLB Schedules with APGAN

If G is a consistent SDF graph, and $\{X, Y\}$ is an adjacent pair in G that does not introduce a cycle, we say that $\{X, Y\}$ satisfies the **proper clustering condition** in G if for each actor $Z \notin \{X, Y\}$ that is adjacent to a member of $\{X, Y\}$, we have that $\rho(\{Z, P\})$ divides $\rho(\{X, Y\})$, for each $P \in \{X, Y\}$ that Z is adjacent to.

In Fig. 3(a) $\mathbf{q}(A, B, C, D, E) = (6, 2, 4, 5, 1)$, and $\rho(\{B, C\}) = 2$ is divisible by $\rho(\{A, C\}) = 2$, $\rho(\{A, B\}) = 2$, $\rho\{C, D\} = 1$, $\rho(\{B, E\}) = 1$, and thus, $\{B, C\}$ satisfies the proper clustering condition.

Theorem 1: [2] Suppose G is a consistent, connected, delayless SDF graph, and $\{X, Y\}$ is an adjacent pair that satisfies the proper clustering condition. Then for each edge e in $clust_G(\{X, Y\})$, $BMLB(e') = BMLB(e)$, where e' is the edge in G that corresponds to e.

The following theorem guarantees that whenever an APGAN instance performs a clustering operation on a graph that has a BMLB schedule, the adjacent pair selected satisfies the proper clustering condition.

Theorem 2: [2] Suppose G is a connected, delayless SDF graph; G has a BMLB schedule; and $\{X, Y\}$ is an APGAN candidate in G. Then $\{X, Y\}$ satisfies the proper clustering condition in G.

Theorem 1 guarantees that clustering an adjacent pair that satisfies the proper clustering condition does not change the BMLB on an edge. However, to derive a BMLB schedule whenever one exists, it is not sufficient to simply ensure that each clusterization step selects an adjacent pair that satisfies the proper clustering condition, since although clustering an adjacent pair that satisfies the proper clustering condition preserves the BMLB values on each edge, it does not necessarily preserve the existence of a BMLB schedule [2]. Fortunately, however, the assumption that the adjacent pair being clustered has maximum repetition count is sufficient to preserve the existence of a BMLB schedule.

Theorem 3: [2] Suppose G is a connected, delayless SDF graph; G has a BMLB schedule; and $\{X, Y\}$ is an APGAN candidate in G. Then $cluster(\{X, Y\}, G)$ has a BMLB schedule.

Lemma 2: If G is a connected, delayless SDF graph that has a BMLB schedule, and P is an APGAN instance, then the schedule obtained by applying P to G is a BMLB schedule for G.

Proof: By definition, P repeatedly clusters APGAN candidates until the top-level graph contains only one actor. From Theorem 2, the first adjacent pair p_1 clustered when P is applied to G satisfies the proper clustering condition, and from Theorem 3, the top level graph T_1 that results from this clustering operation has a BMLB schedule. Since T_1 has a BMLB schedule we can again apply Theorems 2 and 3 to conclude that the second adjacent pair p_2 clustered by P satisfies the proper clustering condition, and that the top-level graph T_2 obtained from clustering p_2 in T_1 has a BMLB schedule. Continuing in this manner for $p_3, p_4, \ldots, p_n$, where n is the total number of adjacent pairs clustered when P is applied to G, we conclude that each adjacent pair clustered by P satisfies the proper clustering condition. From Theorem 1, $BMLB(e') = BMLB(e)$, whenever e' and e are corresponding edges associated with a clusterization step of P. Thus, from Lemma 1,

$$buffer\_memory(S_P(G)) = \sum_{e \in E} BMLB(e).$$

Q.E.D.

The following consequence of Lemma 2 gives our general specification of the optimality of APGAN.

Theorem 4: [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.

6: 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. In this section, we propose an alternative approach, which we call *Recursive Partitioning by Minimum Cuts (RPMC)*, that 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 we can schedule all actors on the left side of the partition before scheduling any on the right side. We also impose the constraint that the partition is fairly evenly sized. 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 [2].

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

$$|V_R| \le K, \; |V_L| \le K. \tag{2}$$

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. Since the related problem of finding a minimum cut (not necessarily legal) into bounded sets is NP-complete [5], and the problem of finding an acyclic partition of a graph is NP-complete [5], we believe this problem to be NP-complete as well even though we have not discovered a proof. Kernighan and Lin [7] devised a heuristic procedure for computing cuts into bounded sets but they considered only undirected graphs. Methods based on network flows [4] do not work because the minimum cut given by the max-flow-min-cut theorem may not be legal and may not be bounded [11]. Hence, we give a heuristic solution 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 [7], 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 $O(|V|^3)$ [11].

7: Experimental Results

Table 1 shows experimental results on the performance of APGAN and RPMC that we have developed for several practical examples of multirate acyclic SDF graphs. The column titled "average random" gives the average cost obtained by considering 100 random topological sorts and applying GDPPO to each. The data for APGAN and RPMC also includes the effect of GDPPO.

We see that APGAN achieves the BMLB on 5 of the 9 examples, outperforming RPMC in these cases. Particularly interesting are the last three examples, which illustrate the performance of the two heuristics as the graph sizes are increased. The graphs represent a symmetric tree-structured QMF filterbank with differing depths. APGAN constructs a BMLB schedule for each of these systems while RPMC generates schedules that have costs about 1.2 times the optimal. Conversely, the third and fourth entries show that RPMC can outperform APGAN significantly on graphs that have more irregular rate changes. These graphs represent nonuniform filterbanks with differing depths.

We have also tested APGAN and RPMC on a large number of randomly-generated 50-actor graphs. For these graphs, RPMC outperformed APGAN 63% of the time, and we also found that both APGAN and RPMC significantly outperformed randomly-generated schedules. Our comparisons on random graphs give a worst case estimate of the performance we can expect from these heuristics.

All of our experiments show that APGAN and RPMC complement each other. For the practical examples, APGAN performs well on graphs that have relatively regular topological structures and rate changes, and RPMC performs well on graphs that are more irregular. Since large random graphs can be expected to consistently have irregular rate changes and topologies, the average performance on random graphs of RPMC is better than APGAN by a wide margin — although we have found that there is a significant proportion of random graphs for which APGAN outperforms RPMC by a margin of over 10% [2], which suggests that APGAN is a useful complement to RPMC even when mostly irregular graphs are encountered. However, the main advantage of adopting both APGAN and RPMC as a combined solution arises from complementing the strong performance of RPMC on general graphs with the formal properties of APGAN, as specified by Theorem 4, and the ability of APGAN to exploit regularity that arises frequently in practical applications.

8: Related Work

In [1], Ade, Lauwereins, and Peperstraete develop upper bounds on the minimum buffer memory requirement for certain classes of SDF graphs. 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 [8], 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 significantly reduce the buffer memory requirement over corresponding (pure) SDF representations, but it is not clear whether this model can in general be used to get schedules that are as compact as single appearance schedules and have lower buffering requirements than those arising from the techniques given in this paper.

A linear programming framework for minimizing the memory requirement of an SDF graph in a parallel processing context is explored by Govindarajan and Gao [6]. Here the goal is to minimize the buffer cost without sacrificing throughput.

9: Conclusions

We have presented two scheduling techniques, APGAN and RPMC, for minimizing the memory required for code and data when mapping an acyclic SDF graph into an implementation on a programmable processor. We have shown that for a certain class of SDF graphs, APGAN is guaranteed to achieve an optimal solution. We have also reported on an experimental study in which we evaluated the performance of APGAN and RPMC. Based on this study, we have concluded that APGAN and RPMC complement each other, and thus, techniques should be investigated for efficiently combining the two methods. In the absence of such a combined solution, or of a more powerful alternative solution, both of these heuristics should be incorporated into SDF-based DSP prototyping and implementation environments in which the minimization of memory requirements is important.

References

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

[2] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, "Two Complementary Heuristics for Translating Graphical DSP Programs into Minimum Memory Software Implementations," Memorandum No. UCB/ERL M95/3, Electronics Research Laboratory, University of California at Berkeley, January, 1995.

[3] S. S. Bhattacharyya, E. A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," *Journal of VLSI Signal Processing*, December, 1993.

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

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

[6] S.R. Govindarajan, G. R. Gao, P. Desai, "Minimizing Memory Requirements in Rate-Optimal Schedules," *Proc. of the Intl. Conf. on Application Specific Array Processors*, August, 1994.

[7] B. W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, February 1970.

[8] R. Lauwereins, P. Wauters, M. Ade, J. A. Peperstraete, "Geometric Parallelism and Cyclo-Static Dataflow in GRAPE-II," *IEEE Wkshp. on Rapid System Prototyping*, June, 1994.

[9] R. Lauwereins, M. Engels, J. A. Peperstraete, E. Steegmans, J. Van Ginderdeuren, "GRAPE: A CASE Tool for Digital Signal Parallel Processing," *IEEE ASSP Magazine*, April, 1990.

[10] E. A. Lee, D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, February, 1987.

[11] P. K. Murthy, S. S. Bhattacharyya, E. A. Lee, *Combined Code and Data Minimization for Synchronous Dataflow Programs*, Memorandum No. UCB/ERL M94/93, Electronics Research Laboratory, University of California at Berkeley, November, 1994.

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

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

[14] S. Ritz, S. Pankert, H. Meyr, "High Level Software Synthesis for Signal Processing Systems," *Proc. of the Intl. Conf. on Application Specific Array Processors*, August, 1992.

Table 1. Performance of APGAN and RPMC on various applications.

System	BMLB	APGAN	RPMC	Avg. Random
Fractional decimation	47	47	52	52
Laplacian pyramid	95	99	99	102
Nonuniform filterbank (4 channels)	85	137	128	172
Nonuniform filterbank (6 channels)	224	756	589	1025
QMF nonuniform-tree filterbank	154	160	171	177
QMF filterbank (one-sided tree)	102	108	110	112
QMF analysis only	35	35	35	43
QMF tree filterbank (4 channels)	46	46	55	53
QMF tree filterbank (8 channels)	78	78	87	93
QMF tree filterbank (16 channels)	166	166	200	227