In the Design Automation for Embedded Systems Journal, Vol 2, No. 1, January 1997.

APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations

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

May 17, 1996

ABSTRACT

Dataflow has proven to be an attractive computational model for graphical DSP design environments that support the automatic conversion of hierarchical signal flow diagrams into implementations on programmable processors. The synchronous dataflow (SDF) model is particularly well-suited to dataflow-based graphical programming because its restricted semantics offer strong formal properties and significant compile-time predictability, while capturing the behavior of a large class of important signal processing applications. When synthesizing software for embedded signal processing applications, critical constraints arise due to the limited amounts of memory. In this paper, we propose a solution to the problem of jointly optimizing the code and data size when converting SDF programs into software implementations.

We consider two approaches. The first 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 to acyclic graphs significantly reduces the complexity of the general PGAN algorithm, and we present a formal study of our modified PGAN technique that rigorously establishes its optimality for a certain class of applications. The second approach that we consider is a top-down technique, based on a generalized *minimum-cut* operation, that was introduced recently in [14]. We present the results of an extensive experimental investigation on the performance of our modified PGAN technique and the top-down approach and on the trade-offs between them. Based on these results, we conclude that these two techniques complement each other, and thus, they should both be incorporated into SDF-based software implementation environments in which the minimization of memory requirements is important. We have implemented these algorithms in the Ptolemy software environment [5] at UC Berkeley.

A portion of this research was undertaken as part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U. S. Air Force (under the RASSP program, contract F33615-93-C-1317), Semiconductor Research Corporation (project 94-DC-008), National Science Foundation (MIP-9201605), Office of Naval Technology (via Naval Research Laboratories), the State of California MICRO program, and the following companies: Bell Northern Research, Dolby, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, Rockwell, Sony, and Synopsys.

S. S. Bhattacharyya is with the Semiconductor Research Laboratory, Hitachi America, Ltd., 201 East Tasman Drive, San Jose, California 95134, USA.

P. K. Murthy and E. A. Lee are with the Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, California 94720, USA.

1 Motivation

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 [11, 12, 15, 16, 17]. In SDF, a program is represented by a directed graph in which each vertex (**actor**) represents a computation, edges specify FIFO communication channels, and each actor produces (consumes) a fixed number of data values (**tokens**) onto (from) each output (input) edge per invocation.

A key property of the SDF model is that static schedules can be constructed at compile time. This removes the overhead of dynamic scheduling and is thus useful for real-time DSP programs where throughput requirements are often severe. Another constraint that programmable DSPs used in embedded systems have is the extremely limited amount of on-chip memory. Typically, these processors might only have around 1000 bytes of program memory and 1000 bytes of data memory. Off-chip memory is usually undesirable because it often entails a speed penalty, increased system cost, and power consumption. Hence, it is imperative that the target code fit inside the on-chip memory whenever possible. While the SDF model is natural for expressing a large class of multirate signal processing algorithms, care must be taken while scheduling to avoid code and data size blowup. This paper considers the following combinatorial optimization problem in SDF scheduling: Given an acyclic SDF graph, amongst the set of possible schedules for this graph, there is a class of schedules that minimizes code size (in terms of metrics that will be defined shortly). We would like to pick those schedules from this code-optimal class that also minimize the data memory required for the buffers on the edges connecting the actors. It should be emphasized that we concentrate on uniprocessor scheduling in this paper.

Fig. 1 shows a simple SDF graph. This graph contains three actors, labeled A, B and C. Each edge is annotated with number of tokens produced (consumed) by its source (sink) actor, and the "D" on the edge from A to B specifies a unit delay. Given an SDF edge e, we denote the source actor and sink actor of e by $src(e)$ and $snk(e)$, and we denote the delay on e by $delay(e)$. Each unit of delay is implemented as an initial token on the edge. Also, $prod(e)$ and $cons(e)$ respectively denote the number of

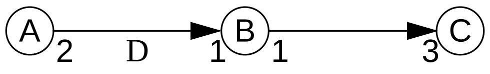

Figure 1. A simple SDF graph.

tokens produced onto e by $src(e)$, and the number of tokens consumed from e by $snk(e)$.

A **schedule** is a sequence of actor firings. We compile a properly-constructed SDF graph by first constructing a finite schedule S that fires each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each edge. We call such a schedule a **valid schedule**. Corresponding to each actor in the schedule S, we instantiate a code block that is obtained from a library of predefined actors, and 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 [13], efficient algorithms are presented to determine whether or not a given SDF graph G 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 row vector $\mathbf{q}_G$, indexed by the actors in G, and we refer to $\mathbf{q}_G$ as the **repetitions vector** of G. We often suppress the subscript if G is understood from context. More precisely, the repetitions vector gives the minimum positive integer solution $\mathbf{x} = \mathbf{q}_G$ to the system of *balance equations*

$$\mathbf{x}(src(e))prod(e) = \mathbf{x}(snk(e))cons(e), \text{ for each edge } e \text{ in } G. \tag{1}$$

A valid schedule is any schedule that does not deadlock, and that invokes each actor A exactly $k\mathbf{q}_G(A)$ times for some positive integer k. This positive integer is called the **blocking factor** of the valid schedule, and it is denoted by J or by $J(S)$, where S is schedule. A schedule that has $J = 1$ is called a **minimal** schedule.

Given an edge e in G, we define the *total number of samples exchanged* on e, denoted $TNSE_G(e)$, by

$$TNSE_G(e) \equiv \mathbf{q}_G(src(e)) \times prod(e) = \mathbf{q}_G(snk(e)) \times cons(e). \tag{2}$$

Thus, $TNSE_G(e)$ is total number of tokens produced onto (consumed from) e in any minimal, valid schedule for G. Note that the equality of the two products in (2) follows from the definition of $\mathbf{q}_G$.

For Fig. 1, $\mathbf{q} = \mathbf{q}(A, B, C) = (3, 6, 2)$, and $TNSE((A, B)) = TNSE((B, C)) = 6$. Note that we adopt the convention of indexing vectors using functional notation rather than subscripts.

One valid schedule for Fig. 1 is $B(2AB)CA(3B)C$. Note that B is allowed to fire first because of the unit delay on the edge (A, B). 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. Observe that this notation naturally accommodates the representation of nested loops. We refer to each parenthesized term $(nS_1S_2\ldots S_k)$ as a **schedule loop** having **iteration count** n and **iterands** $S_1S_2\ldots S_k$.

A **looped schedule** is a finite sequence $(V_1, V_2, \ldots, V_k)$, represented as $V_1V_2\ldots V_k$, where each V_i is either an actor or a schedule loop. Thus, the "looped" qualification indicates that the schedule in question may be expressed in terms of schedule loops. Since a looped schedule is usually executed repeatedly, we refer to each V_i as an **iterand** of the associated looped schedule. Henceforth in this paper, by a "schedule" we mean a "looped schedule."

A more compact valid schedule for Fig. 1 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. This approximation neglects second order affects such as loop overhead and the efficiency of data transfers between actors [3].

Given an SDF graph G, a valid schedule S, and an edge e in G, we define $max\_tokens_G(e, S)$ (we may suppress the subscript if G is understood) to denote the maximum number of tokens that are queued on e during an execution of S. For example if for Fig. 1, $S_1 = (3A)(6B)(2C)$ 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, denoted $buffer\_memory(S)$, by $buffer\_memory(S) = \sum_{e \in E} max\_tokens(e, S)$, where E is the set of edges in G. Thus, $buffer\_memory(S_1) = 7 + 6 = 13$, and $buffer\_memory(S_2) = 3 + 6 = 9$.

In the model of buffering implied by our "buffer memory requirement" measure, each buffer is mapped to an independent contiguous block of memory. Although perfectly valid target programs can be generated without this restriction, it can be shown that having a separate buffer on each edge is advantageous because it permits full exploitation of the memory savings attainable from nested loops, and it accommodates delays without complication [14]. Another advantage of this model is that by favoring the generation of nested loops, the model also favors schedules that have lower latency than single appearance schedules that are constructed to optimize various alternative cost measures [14]. Combining the analysis and techniques that we develop in this paper with methods for sharing storage among multiple buffers is a useful direction for further study. Existing techniques for sharing buffers usually do not take the scheduling into account; for example, the common buffer sharing strategy of combining liveness analysis and graph coloring is used for a given schedule. Also, most existing techniques assume that every buffer being implemented is of the same size. They also do not apply to SDF graphs, where the presence of rate changes complicates matters further. Fabri [7] has studied schemes for overlaying buffers when the buffer sizes are not all identical but even these techniques only apply to a given schedule, and do not attempt to optimize over all possible schedules as done in this paper. Finally, as shown in [14], naive techniques for buffer-sharing can result in sub-optimal schedules, and can be awkward to implement.

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**. From the discussion above, it should be clear that this scheduling problem of minimizing memory requirements even for a single processor is a challenging, non-trivial problem. We focus on acyclic graphs. We introduce a customization to acyclic graphs of a bottom-up scheduling technique, called *pairwise grouping of adjacent nodes (PGAN),* that was proposed in an earlier paper [4] for general SDF graphs. We call this customization *Acyclic PGAN (APGAN)*. We show that APGAN significantly reduces the time and space complexity of the general PGAN algorithm; we rigorously establish that APGAN performs optimally for a certain class of SDF graphs; and we give examples of practical applications that fall within the class of graphs for which APGAN produces optimal results. We present experimental data on practical applications that verifies that our implementation of APGAN performs optimally for graphs that fall within the specified class, and suggests that it often performs very well for graphs that lie outside the class.

We compare APGAN to a top-down heuristic based on recursively partitioning the input graph using a generalized minimum cut operation, which was introduced recently in [14]. This top-down heuristic is called *Recursive Partitioning Based on Minimum Cuts (RPMC).* We report on an extensive experimental study in which the performance of both scheduling techniques is evaluated on several practical applications, and on a diverse collection of complex random graphs. The conclusions that we derive are that techniques should be investigated for efficiently combining the methods of RPMC and APGAN, and that 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. An algorithm based on APGAN has in fact been implemented by the Alta group at Cadence Design Systems Inc. in their Signal Processing WorkSystem programming environment. We have implemented APGAN and RPMC in the Ptolemy programming environment [5] at UC Berkeley and will be making these algorithms available in the next release.

The paper is organized as follows. In Section 2 we first review some graph concepts and establish notation that will be used throughout the paper. We then prove some facts about clustering in SDF graphs that will be useful in the development of the APGAN algorithm. We also discuss the problem of constructing a buffer-optimal loop hierarchy for a given lexical ordering of nodes and present a polynomial-time algorithm that computes it optimally. In Section 3 we prove a simple lower bound on the memory requirement (called BMLB) of any single appearance schedule for an acyclic SDF graph and in Section 4 we describe the APGAN algorithm. In Section 5 we develop a concept called *proper clustering*. Section 6 develops one of the main results of this paper; namely, the optimality of APGAN for a particular class of SDF graphs. Even though this class appears restrictive at first, it is shown in Section 8 that a wide variety of practical systems fall into this class and hence it is a useful class. Section 7 briefly discusses a different heuristic that was proposed in [14]; this discussion is given primarily to facilitate the comparison between the two heuristics in Section 8. Finally we discuss some related work and present our conclusions.

2 Background

For reference, a glossary of terminology can be found at the end of the paper.

Given a finite set H, we denote the number of elements in H by $|H|$. If x and y are positive integers, we say that x *divides* y if $y = kx$ for some positive integer k. If the members of H are positive integers, then by $gcd(H)$ we mean the largest positive integer that divides all members of H.

Precisely speaking, SDF graphs, as we use them in this paper, are directed multigraphs rather than directed graphs, since we allow two or more edges to have the same source and sink vertices. However, we often ignore this distinction. Thus, when there is no ambiguity, we may refer to an edge e as the ordered pair $(src(e), snk(e))$. We frequently represent an SDF graph G by an ordered pair (V, E), where V is the set of vertices and E is the set of edges. By a **subgraph** of G, we mean the directed graph formed by any $V' \subseteq V$ and the set of edges $\{e \in E | src(e), snk(e) \in V'\}$. We denote the subgraph associated with the vertex subset V' by $subgraph(V')$. A **connected component** of G is a subset $V' \subseteq V$ such that $subgraph(V')$ is connected, and no subset of V that properly contains V' induces a connected subgraph.

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**. A **path** in G from X to Y is a finite, nonempty sequence $(e_1, e_2, \ldots, e_n)$ such that each e_i is a member of E, $X = src(e_1)$, $Y = snk(e_n)$, and $snk(e_1) = src(e_2)$, $snk(e_2) = src(e_3)$, $\ldots$, $snk(e_{n-1}) = src(e_n)$. If $(p_1, p_2, \ldots, p_k)$ is a finite sequence of paths such that $p_i = (e_{i,1}, e_{i,2}, \ldots, e_{i,n_i})$, for $1 \le i \le k$, and $snk(e_{i,n_i}) = src(e_{i+1,1})$, for $1 \le i \le (k-1)$, then we define

$$\langle (p_1, p_2, \ldots, p_k) \rangle \equiv (e_{1,1}, \ldots, e_{1,n_1}, e_{2,1}, \ldots, e_{2,n_2}, \ldots, e_{k,1}, \ldots, e_{k,n_k}) .$$

Clearly, $\langle (p_1, p_2, \ldots, p_k) \rangle$ is a path from $src(e_{1,1})$ to $snk(e_{k,n_k})$. 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. A path that is directed from a vertex to itself is a **cycle**. If G is acyclic, a **topological sort** for (V, E) is an ordering $(v_1, v_2, \ldots, v_{|V|})$ of the members of V such that for each $e \in E$, $((src(e) = v_i)$ **and** $(snk(e) = v_j)) \Rightarrow (i < j)$.

If e is an SDF edge, then the **delayless version** of e is an edge e' such that $e' = e$ if $delay(e) = 0$, and if $delay(e) \neq 0$, then e' is the edge defined by $src(e') = src(e)$, $snk(e') = snk(e)$, and $delay(e') = 0$. If $G = (V, E)$ is an SDF graph, then G is **delayless** if $delay(e) = 0$ for all $e \in E$, and the **delayless version** of G is the SDF graph defined by (V, E'), where $E' = \{\text{the delayless version of } e | e \in E\}$. In words, the delayless version of G is the graph that results from setting the delays on all edges to zero.

A contiguous sequence of actors and schedule loops in a looped schedule S is called a **subschedule** of S. For example, the schedules $(3AB)C$, $(2D(3AB)C)$, and $(4E)(2D(3AB)C)$ are all subschedules of $(4E)(2D(3AB)C)$. If S_0 is a subschedule of S, then S_0 is **contained in** S, and S_0 is **nested** in S if S_0 is contained in S and $S_0 \neq S$.

We denote the set of actors that appear in a single appearance schedule S by $actors(S)$, and given an $A \in actors(S)$, we define $inv(A, S)$ to be the number of times that S invokes A. Similarly, if S_0 is a subschedule of S, $inv(S_0, S)$ is the number of times that S invokes S_0. For example, if $S = (2(3B(2CD)))(5E)$, then $inv(E, S) = 5$, and $inv((2CD), S) = 6$.

We define $position(X, S)$ to be the number of actors that lexically precede X in the single appearance schedule S. Thus, if $S = (2(3B)(5C))(7A)$, then $position(A, S) = 2$. Also, the **lexical ordering** of a single appearance schedule S, denoted $lexorder(S)$, is the sequence of actors $(A_1, A_2, \ldots, A_n)$ where $\{A_1, A_2, \ldots, A_n\} = actors(S)$, and $position(A_i, S) = i - 1$ for each i. Thus, $lexorder((2(3B)(5C))(7A)) = (B, C, A)$. We will apply the following obvious fact about lexical orderings.

Fact 1: If S is a valid single appearance schedule for a delayless SDF graph, then whenever X is an ancestor of Y, we have $position(X, S) < position(Y, S)$.

We will also apply the following fact, whose proof can be found in [2].

Fact 2: Suppose that $G = (V, E)$ is a consistent, connected SDF graph, S is a single appearance schedule for G, and k is any positive integer. Then there exists a valid single appearance schedule S' for G

such that $J(S') = k$, $lexorder(S') = lexorder(S)$, and $max\_tokens(e, S') \le max\_tokens(e, S)$, for each $e \in E$.

Suppose that S is a looped schedule for an SDF graph G and Z is a set of actors. If we remove from S all actors that are not in Z, and then we repeatedly remove all null loops (loops that have empty bodies) until no null loops remain, we obtain another looped schedule, which we call the **projection** of S onto Z, denoted $projection(S, Z)$. For example, $projection((2(2B)(5A)), \{A, C\}) = (2(5A))$. Clearly, $projection(S, Z)$ fully specifies the sequence of token populations occurring on each edge in $subgraph(Z)$. More precisely, for any $A \in Z$, any $i \in \{1, 2, \ldots, inv(A, S)\}$, and any input edge e of A contained in $subgraph(Z)$, the number of tokens queued on e just before the ith invocation of A in S equals the number of tokens queued on e just before the ith invocation of A in an execution of $projection(S, Z)$. Thus, we have the following fact.

Fact 3: If S is a valid looped schedule for an SDF graph $G = (V, E)$, and $Z \subseteq V$, then $projection(S, Z)$ is a valid looped schedule for $subgraph(Z)$, and $max\_tokens(e, projection(S, Z)) = max\_tokens(e, S)$, for each edge e in $subgraph(Z)$.

If Z is a subset of actors in a connected, consistent SDF graph G, we define $\rho_G(Z) \equiv gcd(\{\mathbf{q}_G(A) | A \in Z\})$, and we refer to this quantity as the **repetition count** of Z. The subscript may be dropped if G is understood from context.

2.1 Clustering

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 $E' = E - (\{e | (src(e) \in Z) \textbf{ or } (snk(e) \in Z)\}) + E^*$, where 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

$$src(e') = \Omega,\ snk(e') = snk(e),$$

$$delay(e') = delay(e),\ prod(e') = prod(e) \times (\mathbf{q}_G(src(e))/\rho_G(Z)), \text{ and } cons(e') = cons(e);$$

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

$$src(e') = src(e),\ snk(e') = \Omega$$

$$delay(e') = delay(e),\ prod(e') = prod(e), \text{ and}$$

$$cons(e') = cons(e) \times (\mathbf{q}_G(snk(e))/\rho_G(Z)),$$

then, we can specify E^* by

$$E^* = \{e' | (src(e) \in Z \textbf{ and } snk(e) \notin Z) \textbf{ or } (snk(e) \in Z \textbf{ and } src(e) \notin Z)\}.$$

For each $e' \in E^*$, 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 $cluster_G(Z, \Omega)$, or simply $cluster(Z)$. Intuitively, an invocation of Ω in $cluster_G(Z, \Omega)$ corresponds to an invocation of a minimal valid schedule for $subgraph(Z)$ in G. We say that Z is **clusterable** if $cluster_G(Z, \Omega)$ is consistent, and if G is acyclic, we say that Z **introduces a cycle** if $cluster_G(Z, \Omega)$ contains one or more cycles. Fig. 2 gives an example of clustering. Here, edge (D, Ω) corresponds to (D, C) (and vice versa), and (Ω, A) corresponds to (B, A).

The following fact relates the repetitions vector of an SDF graph obtained by clustering a subgraph to that of the original SDF graph. The proofs of both parts can be found in [3].

Fact 4: (a). If $G = (V, E)$ is a connected, consistent SDF graph, $Z \subseteq V$, and $G' = cluster_G(Z, \Omega)$, then $\mathbf{q}_{G'}(\Omega) = \rho_G(Z)$, and for each $A \in (V - Z)$, $\mathbf{q}_{G'}(A) = \mathbf{q}_G(A)$.

(b). If G is a connected, consistent SDF graph and $G' = (V', E')$ is a connected subgraph of G, then for each $A \in V'$, $\mathbf{q}_{G'}(A) = \mathbf{q}_G(A)/\rho_G(V')$.

Fact 4(a) together with the definition of clustering immediately yields

Fact 5: If G and G' are as in Fact 4(a), then for each edge e in G', $TNSE_{G'}(e) = TNSE_G(e')$, where e' is the edge in G that corresponds to e.

2.2 R-schedules

If Λ is either a schedule loop or a looped schedule, we say that Λ satisfies the **R-condition** if one

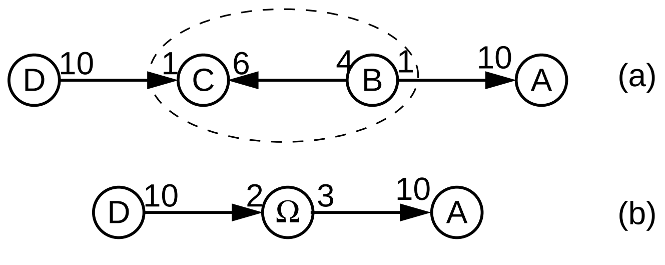

Figure 2. An example of clustering. In (b), we have $cluster_G(\{B, C\}, \Omega)$, where G denotes the SDF graph in (a). Here, $\mathbf{q}_G(A, B, C, D) = (3, 30, 20, 2)$.

of the following two conditions holds: (a) Λ has a single iterand, and this single iterand is an actor, *or* (b) Λ has exactly two iterands, and these two iterands are schedule loops having coprime iteration counts.

A valid single appearance schedule S is an **R-schedule** if S satisfies the R-condition, and every schedule loop contained in S satisfies the R-condition. The following result on R-schedules is established in [2].

Theorem 1: Suppose that $G = (V, E)$ is a consistent SDF graph and S is a valid single appearance schedule for S. Then there exists an R-schedule S_R for S such that $max\_tokens(e, S_R) \le max\_tokens(e, S)$ for all $e \in E$, and $lexorder(S_R) = lexorder(S)$.

2.3 Optimally Reparenthesizing a Single Appearance Schedule

In [14], a dynamic programming algorithm is developed that constructs an optimal schedule for a *well-ordered* SDF graph (a graph that has only one topological sort) in $O(v^3)$ time, where v is the number of actors. An adaptation of this technique is also presented for general, delayless, consistent SDF graphs[1] that computes a single appearance schedule that has minimum buffer memory requirement from among the single appearance schedules that have a given lexical ordering. We refer to this adaptation as **Dynamic Programming Post Optimization (DPPO)** for single appearance schedules. DPPO can be extended efficiently to handle delays and arbitrary topologies [3]. We refer to the extension that we have developed as **Generalized DPPO (GDPPO)**.

GDPPO gives a post-optimization for any scheduler for general SDF graphs that constructs single appearance schedules. Applying GDPPO to a single appearance schedule S yields a schedule that has a buffer memory requirement that is less than or equal to the buffer memory requirement of every valid single appearance schedule that has the same lexical ordering as S. In the remainder of this paper, we discuss two heuristics for constructing single appearance schedules, and we present an experimental study that compares these heuristics — with their schedules post-processed by GDPPO — against each other and against randomly generated schedules that are post-processed by GDPPO. To enhance our analysis of these heuristics, we first develop a fundamental lower bound on the buffer memory requirement of a single appearance schedule.

1. Note that for consistent SDF graphs, *delayless* implies *acyclic*, and thus, we are referring here to the class of consistent, acyclic — but not necessarily well-ordered — SDF graphs such that the delay on each edge is zero.

3 A Lower Bound on the Buffer Memory Requirement

Given a consistent SDF graph G, there is an efficiently computable upper and lower bound on the buffer memory requirement over all valid single appearance schedules. Our lower bound can be derived easily by examining a generic two-actor SDF graph, as shown in Fig. 3(a). From the balance equations (see (1)), it is easily verified that the repetitions vector for this graph is given by $\mathbf{q}(A, B) = \left(\frac{q}{g}, \frac{p}{g}\right)$, where $g \equiv gcd(\{p, q\})$, and that if $d < \frac{pq}{g}$, then the only R-schedule for this graph is $S_1 = \left(\frac{q}{g}A\right)\left(\frac{p}{g}B\right)$. From Theorem 1 it follows that if $d < \frac{pq}{g}$, then $max\_tokens((A, B), S_1) = \left(\frac{pq}{g} + d\right)$ is a lower bound for the buffer memory requirement of the graph in Fig. 3(a). Similarly, if $d \geq \frac{pq}{g}$, then there are exactly two R-schedules — S_1 and $S_2 = \left(\frac{p}{g}B\right)\left(\frac{q}{g}A\right)$. Since $max\_tokens((A, B), S_2) = d$, we obtain d as a lower bound for the buffer memory requirement. Thus, given a valid single appearance schedule S for Fig. 3(a), we have that

$$\left(d < \frac{pq}{g}\right) \Rightarrow \left(max\_tokens((A, B), S) \geq \left(\frac{pq}{g} + d\right)\right), \text{ and}$$

$$\left(d \geq \frac{pq}{g}\right) \Rightarrow (max\_tokens((A, B), S) \geq d). \qquad (3)$$

Furthermore, if (A, B) is an edge in a general SDF graph, we know from Fact 3 that the projection of a valid schedule S onto $\{A, B\}$, which is a valid schedule for $subgraph(\{A, B\})$, always satisfies

$$max\_tokens((A, B), projection(S, \{A, B\})) = max\_tokens((A, B), S). \qquad (4)$$

It follows that the lower bound defined by (3) holds whenever (A, B) is an edge in a consistent SDF graph

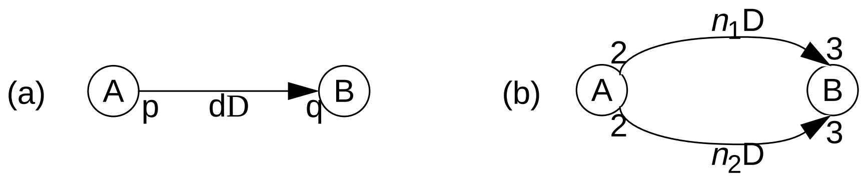

Figure 3. Examples used to develop the buffer memory lower bound.

G, S is a valid single appearance schedule for G, $(prod((A, B)) = p)$, $(cons((A, B)) = q)$, and $g = gcd(\{p, q\})$. We have motivated the following definition.

Definition 1: Given an SDF edge e, we define the **buffer memory lower bound (BMLB)** of e, denoted $BMLB(e)$, by

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

$$\eta(e) = \frac{prod(e)cons(e)}{gcd(\{prod(e), cons(e)\})}.$$

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, we see that $BMLB((A, B)) = 3$, and $BMLB((B, C)) = 3$. Thus, a valid single appearance schedule for Fig. 1 is a BMLB schedule if and only if its buffer memory requirement equals 6. It is easily verified that only two R-schedules for Fig. 1 exist — $(3A(2B))(2C)$, and $(3A)(2(3B)C)$; the associated buffer memory requirements are $3 + 6 = 9$ and $7 + 3 = 10$, respectively. Thus, a BMLB schedule does not exist for Fig. 1.

In contrast, the SDF graph shown in Fig. 4 has a BMLB schedule. This graph results from simply interchanging the production and consumption parameters of edge (B, C) in Fig. 1. Here, $\mathbf{q}(A, B, C) = (1, 2, 6)$, the BMLB values for both edges are again identically equal to 3, and $A(2B(3C))$ is a valid single appearance schedule whose buffer memory requirement achieves the sum of these BMLB values.

The following fact is a straightforward extension of our development of the BMLB.

Fact 6: Suppose that G is an SDF graph that consists of two vertices A, B and $n \geq 1$ edges $e_1, e_2, \ldots, e_n$ directed from A to B. Then (a). if $delay(e_i) \geq \eta(e_i)$ for all $i \in \{1, 2, \ldots, n\}$, then

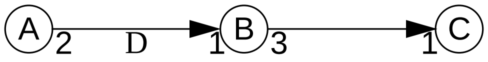

Figure 4. An SDF graph that has a BMLB schedule.

$(\mathbf{q}_G(B)B)(\mathbf{q}_G(A)A)$ is a BMLB schedule for G; (b) otherwise, $(\mathbf{q}_G(A)A)(\mathbf{q}_G(B)B)$ is an optimal schedule — that is, it minimizes the buffer memory requirement over all valid single appearance schedules — for G, and it is a BMLB schedule if and only if $delay(e_i) < \eta(e_i)$ for $1 \le i \le n$.

Fact 7: If $G = (V, E)$ is a connected, consistent, acyclic SDF graph, and $delay(e) < \eta(e)$ for all $e \in E$, then S is a BMLB schedule for the delayless version of G if and only if S is a BMLB schedule for G.

Proof: Let G' denote the delayless version of G. If S is a BMLB schedule for G', then S is a valid schedule for G that satisfies $max\_tokens_G(e, S) = max\_tokens_{G'}(e, S) + delay(e)$ for all $e \in E$. It follows from Definition 1 that S is BMLB schedule for G. Similarly, if S a BMLB schedule for G, then S is a valid schedule for G', and $max\_tokens_{G'}(e, S) = max\_tokens_G(e, S) - delay(e)$. Again, from Definition 1, S must be a BMLB schedule for G'. *Q.E.D.*

A proof of the following fact can be found in [2].

Fact 8: If G is a connected, consistent SDF graph and e is an edge in G, then

$$\eta(e) = \frac{TNSE_G(e)}{\rho_G(\{src(e), snk(e)\})}.$$

4 PGAN for Acyclic Graphs

In the original *Pairwise Grouping of Adjacent Nodes (PGAN)* technique, developed in [4], 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 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)* [13]. 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.

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 $cluster_G(Z, \Omega)$ is not a sufficient condition for Z not to be clusterable, the clusterizeability 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 clusterization test attains a large computational savings over the exact test based on the reachability matrix of the APG, and this is our main reason for adopting it.

Fig. 5 illustrates the operation of APGAN. Fig. 5(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. Each edge corresponds to a different adjacent pair; 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 the one of the three adjacent pairs $\{A, B\}$, $\{A, C\}$, or $\{B, C\}$ for its first clusterization step. Examination of the reachability matrix yields that $\{A, C\}$ introduces a cycle due to the path $((A, B), (B, C))$, 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. 5(b) shows the graph that results from clustering $\{A, B\}$ into the hierarchical actor Ω_1. Here, $\mathbf{q}(\Omega_1, C, D, E) = (2, 4, 5, 1)$, and $\{\Omega_1, C\}$ 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. 5(c) shows the resulting graph.

In Fig. 5(c), we have $\mathbf{q}(\Omega_2, D, E) = (2, 5, 1)$, and thus all three adjacent pairs have $\rho = 1$.

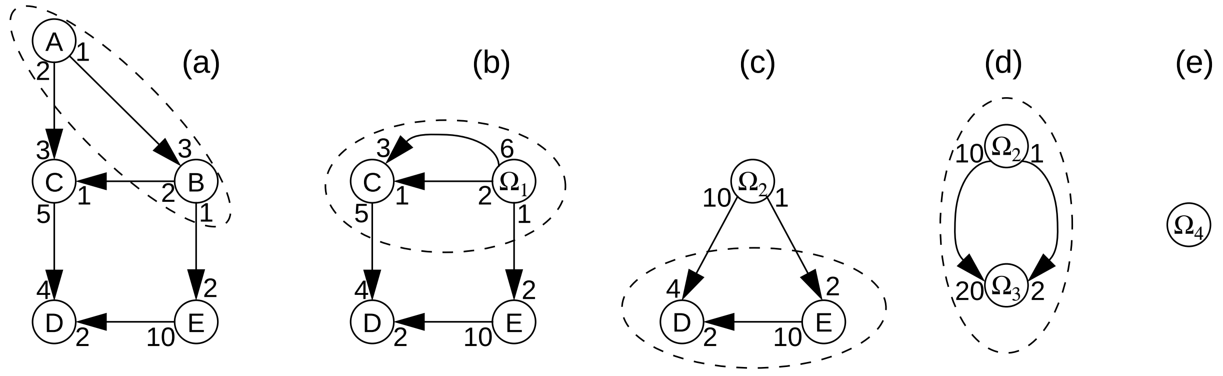

Figure 5. An illustration of APGAN.

Among these, clearly, only $\{\Omega_2, E\}$ and $\{E, D\}$ do not introduce cycles, so APGAN arbitrarily selects among these two to determine the third clusterization pair. Fig. 5(d) shows the graph that results when $\{E, D\}$ is chosen. This graph contains only one adjacent pair $\{\Omega_2, \Omega_3\}$, and APGAN will consolidate this pair in its final clusterization step to obtain the single-vertex graph in Fig. 5(e).

Figs. 5(b-e) specify the sequence of clusterizations performed by APGAN when applied to the graph of Fig. 5(a). 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. 5(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. The subgraph G_i corresponding to each Ω_i consists of only two actors X_i and Y_i, such that all edges in G_i are directed from X_i to Y_i. Thus, from Fact 6, it is clear how an optimal schedule can easily be constructed for the subgraph corresponding to each Ω_i: if each edge e in G_i satisfies $delay(e) \geq \eta(e)$, then we construct the schedule $(\mathbf{q}_{G_i}(Y_i)Y_i)(\mathbf{q}_{G_i}(X_i)X_i)$, and otherwise we construct $(\mathbf{q}_{G_i}(X_i)X_i)(\mathbf{q}_{G_i}(Y_i)Y_i)$. In Fig. 5, 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 $(5D)E$. Since this subgraph contains no hierarchical actors, $(5D)E$ is immediately returned as the "flattened" schedule for the subgraph corresponding to Ω_3. 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 $\Omega_1(2C)$ for the corresponding subgraph. 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 $(3A)B(2C)$. This gets substituted for Ω_2 in the top-level schedule to yield the schedule $S_p \equiv (2(3A)B(2C))(5D)E$ for Fig. 5(a).

From S_p and Fig. 5(a) it is 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. 5(a), are identically equal to 43, and thus in the execution of APGAN illustrated in Fig. 5, a BMLB schedule is constructed.

As seen in the above example, the APGAN approach, as we have defined it here, does not uniquely specify the sequence of clusterizations that will be performed, and thus, it does not in general, result in a unique schedule for a given SDF graph. APGAN 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. For example, one tie-breaking protocol that can be used when actors are labelled alphabetically, as in Fig. 5, is to choose that adjacent pair that maximizes the sum of the "distances" of the actor labels from the letter "A". If this protocol is used to break the tie between $\{A, B\}$ ("distance sum" is $0 + 1 = 1$) and $\{B, C\}$ (distance sum is $1 + 2 = 3$) in the first clusterization of step of Fig. 5, then $\{B, C\}$ is chosen.

If an efficient data structure is used to maintain the list of pairwise clustering candidates, then it can be shown that APGAN instances exist with running times that are $O(V^2 E)$.

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 all other adjacent pairs that do not introduce cycles. Thus, an APGAN instance is any algorithm that takes a consistent, acyclic SDF graph as input, repeatedly clusters APGAN candidates, and then outputs the schedule corresponding to a recursive traversal of the resulting cluster hierarchy.

In the following two sections, we show that for a consistent, acyclic SDF graph (V, E) that has a BMLB schedule, and that satisfies $delay(e) < \eta(e)$ for each $e \in E$, any APGAN instance is guaranteed to obtain a BMLB schedule when applied to this graph.

The following fact, which is easily understood from our discussion of the example in Fig. 5, is fundamental to developing our result on the optimality of APGAN instances.

Fact 9: Suppose G is a connected, consistent, acyclic SDF graph such that $delay(e) < \eta(e)$ 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 in the subgraphs corresponding to the hierarchical actors $\{\Omega_i\}$ instantiated by P.

For the example of Fig. 5, E_Ω is the set of six edges that are enclosed by dashed ovals in Fig. 5(a-d). It is easily seen that the BMLB values for these edges are 3, 6, 2, 10, 2, and 20. Thus, Fact 9 states that the schedule obtained from the sequence of clusterizations shown in Fig. 5 has a buffer memory requirement equal to $3 + 6 + 2 + 10 + 2 + 20 = 43$, which we know is correct from the discussion above.

There are two main parts in the development of our optimality result. First, we define a certain class of "proper" clusterizations; we show that for delayless graphs, such clusterizations have the property that they do not increase the BMLB values on any edge; and we show that under the assumption that a BMLB schedule exists, a clustering operation performed by any APGAN instance is guaranteed to fall in the class of proper clusterizations. Then we show that clustering an APGAN candidate cannot transform a graph that has a BMLB schedule into a graph that does not have a BMLB schedule. From these three developments and Facts 7 and 9, the desired result can be derived easily.

5 Proper Clustering

Definition 2: If G is a connected, 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. 5(a) $\mathbf{q}(A, B, C, D, E) = (6, 2, 4, 5, 1)$, and $\rho(\{B, C\}) = 2$ is divisible by $\rho(\{A, C\}) = 2$, $\rho(\{A, B\}) = 2$, $\rho(\{C, D\}) = 1$, and $\rho(\{B, E\}) = 1$, and thus, $\{B, C\}$ satisfies the proper clustering condition. Conversely, $\rho(\{B, E\})$ is not divisible by $\rho(\{B, C\})$, so $\{B, E\}$ does not satisfy the proper clustering condition.

The motivation for Definition 2 is given by Theorem 2 below, which establishes that when the proper clustering condition is satisfied, clustering $\{X, Y\}$ does not change the BMLB on any edge, and that when the proper clustering condition is not satisfied, clustering $\{X, Y\}$ increases the BMLB on at least one edge. Thus, a clustering operation that does not satisfy the proper clustering condition cannot be used to derive a BMLB schedule.

To establish Theorem 2, we will use the following simple fact about greatest common divisors, which we state here without proof.

Fact 10: Suppose that a, b, c are positive integers. If $gcd(\{a, b\})$ divides $gcd(\{a, c\})$, then $gcd(\{a, b, c\}) = gcd(\{a, b\})$; otherwise, $gcd(\{a, b, c\}) < gcd(\{a, b\})$.

Theorem 2: Suppose that G is a consistent, connected, delayless SDF graph, and $\{X, Y\}$ is a clusterable adjacent pair in G. If $\{X, Y\}$ satisfies the proper clustering condition, then for each edge e in $G_c \equiv cluster_G(\{X, Y\})$, $BMLB(e') = BMLB(e)$, where e' is the edge in G that corresponds to e. If $\{X, Y\}$ does not satisfy the proper clustering condition, then there exists an edge e in G_c such that $BMLB(e') < BMLB(e)$.

For example, in Fig. 6(a), $BMLB((A, B)) = 2$, $BMLB((B, C)) = 3$, and $\mathbf{q}(A, B, C) = (1, 2, 6)$. Figs. 6(b) and 6(c) respectively show $cluster_G(\{A, B\}, \Omega)$ and

$cluster_G(\{B, C\}, \Omega)$, where G denotes the graph of Fig. 6(a). In Fig. 6(b), we see that if $e' = (B, C)$, then $e = (\Omega, C)$, and $BMLB(e) = 6$, while $BMLB(e') = 3$, and thus, $BMLB(e) > BMLB(e')$. In contrast, in Fig. 6(c), we see that if $e' = (A, B)$, then $e = (A, \Omega)$, and $BMLB(e) = BMLB(e') = 2$. These observations are consistent with Theorem 2 since $\{B, C\}$ satisfies the proper clustering condition, while $\{A, B\}$ does not.

Proof of Theorem 2: First, suppose that $\{X, Y\}$ satisfies the proper clustering condition. Let e be an edge in G_c, and let e' be the corresponding edge in G. If $src(e), snk(e) \neq \Omega$, then $e' = e$, so from Definition 1, it follows that $BMLB(e) = BMLB(e')$.

If $src(e) = \Omega$, observe that $snk(e) = snk(e')$ and $src(e') \in \{X, Y\}$, and observe from Fact 4(a) that $\rho_{G_c}(\{src(e), snk(e)\}) = gcd(\{\mathbf{q}_G(X), \mathbf{q}_G(Y), \mathbf{q}_G(snk(e))\})$. Thus, since $\{X, Y\}$ satisfies the proper clustering condition, it follows from Fact 10 that $\rho_{G_c}(\{src(e), snk(e)\}) = \rho_G(\{src(e'), snk(e')\})$. From Facts 5 and 8, we conclude that $BMLB(e) = BMLB(e')$. A symmetric argument can be constructed for the case $(snk(e) = \Omega)$. Thus, we have that $BMLB(e) = BMLB(e')$ whenever $\{X, Y\}$ satisfies the proper clustering condition.

If $\{X, Y\}$ does not satisfy the proper clustering condition, then there exists an actor $Z \notin \{X, Y\}$ that is adjacent to some $P \in \{X, Y\}$ such that

$$\rho_G(\{Z, P\}) \text{ does not divide } \rho_G(\{X, Y\}). \qquad (5)$$

Without loss of generality, suppose that $P = X$ and X is a predecessor of Z (the other possibilities can be handled with symmetric arguments). Let e' be an edge directed from X to Z in G, and let e be the corresponding edge (directed from Ω to Z) in G_c. From Fact 4(a), $\rho_{G_c}(\{src(e), snk(e)\}) = gcd(\{\mathbf{q}_G(X), \mathbf{q}_G(Y), \mathbf{q}_G(snk(e))\})$, and thus from (5) and Fact 10, it follows that $\rho_{G_c}(\{src(e), snk(e)\}) < \rho_G(\{src(e'), snk(e')\})$. From Facts 5 and 8, we conclude that $BMLB(e) > BMLB(e')$. *Q.E.D.*

The following lemma establishes that if there is an adjacent pair $\{X, Y\}$, X is a predecessor of Y,

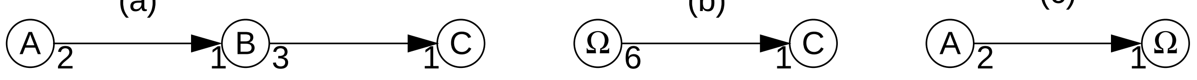

Figure 6. An example used to illustrate Theorem 2.

that introduces a cycle in a delayless SDF graph that has a BMLB schedule, then there exists an actor $V \notin \{X, Y\}$ that is a predecessor of Y and a descendant (recall the distinction between *descendant* and *successor*) of X, such that the repetition count of $\{V, Y\}$ is divisible by the repetition count of $\{X, Y\}$. A simple example is shown in Fig. 7.

Lemma 1: Suppose that G is a connected, delayless, consistent SDF graph that has a BMLB schedule, and e is an edge in G such that $\{src(e), snk(e)\}$ introduces a cycle. Then there exists an actor V in G such that V is a predecessor of $snk(e)$, V is a descendant of $src(e)$; and $\rho_G(\{src(e), snk(e)\})$ divides $\rho_G(\{V, snk(e)\})$.

Proof: Observe that from Theorem 1, there exists a BMLB schedule S_R for G that is an R-schedule; since $(\{src(e), snk(e)\})$ introduces a cycle, there is a path $(e_1, e_2, \ldots, e_n)$, $n \geq 2$, from $src(e)$ to $snk(e)$; and from Fact 1, $position(src(e), S_R) < position(src(e_n), S_R) < position(snk(e), S_R)$. Thus, there exists a schedule loop $L = (i_0(i_1B_1)(i_2B_2))$ in $(1S_R)$, where B_1 and B_2 are schedule loop bodies such that (a) B_1 contains $src(e)$, and B_2 contains both $src(e_n)$ and $snk(e)$, or (b) B_1 contains both $src(e)$ and $src(e_n)$, and B_2 contains $snk(e)$. Observe that L is simply the innermost schedule loop in $(1S_R)$ that contains $src(e)$, $src(e_n)$, and $snk(e)$.

Without loss of generality, assume that (a) applies — that is, assume that B_1 contains $src(e)$, and B_2 contains both $src(e_n)$ and $snk(e)$. Then there is a schedule loop $L' = (i_0'(i_1'B_1')(i_2'B_2'))$ contained in (i_2B_2) such that B_1' contains $src(e_n)$, and B_2' contains $snk(e)$. This is the innermost schedule loop that contains $src(e_n)$ and $snk(e)$, and this loop may be (i_2B_2), or it may be nested in (i_2B_2).

Let I be the product of the iteration counts of all schedule loops in $(1S_R)$ that contain

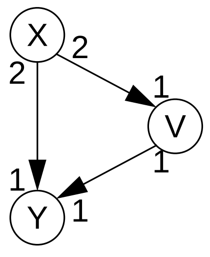

Figure 7. An illustration of Lemma 1. Here, $\mathbf{q}(V, X, Y) = (2, 1, 2)$, $X(2VY)$ is a BMLB schedule, and $\{X, Y\}$ introduces a cycle. Thus, Lemma 1 guarantees that $\rho(\{X, Y\})$ divides $\rho(\{V, Y\})$, and this is easily verified from $\mathbf{q}$.

$(i_1 B_1)(i_2 B_2)$. Similarly, let I' be the product of all schedule loops contained in $(i_2 B_2)$ that contain $(i_1' B_1')(i_2' B_2')$. Then, it is easily verified that

$max\_tokens(e, S_R) = \mathbf{q}_G(src(e))prod(e)/I = TNSE(e)/I$, and

$max\_tokens(e_n, S_R) = (\mathbf{q}_G(src(e_n))prod(e_n))/(II') = TNSE(e_n)/(II')$.

Since S_R is a BMLB schedule, we have from Fact 8 that $\rho_G(\{src(e), snk(e)\}) = I$, and $\rho_G(\{src(e_n), snk(e)\}) = II'$. Thus, $\rho_G(\{src(e), snk(e)\})$ divides $\rho_G(\{src(e_n), snk(e)\})$. Furthermore, since the path $(e_1, e_2, \ldots, e_n)$ originates at $src(e)$, we know that $src(e_n)$ is a descendant of $src(e)$. *Q.E.D.*

The following corollary to Lemma 1 states that under the hypotheses of Lemma 1 (a BMLB schedule exists and $\{src(e), snk(e)\}$ introduces a cycle), we are guaranteed the existence of an adjacent pair $\{V, snk(e)\}$ such that $\{V, snk(e)\}$ does not introduce a cycle, and the repetition count of $\{src(e), snk(e)\}$ divides the repetition count of $\{V, snk(e)\}$.

Corollary 1: Assume the hypotheses of Lemma 1. Then, there exists a predecessor $V \neq src(e)$ of $snk(e)$ such that $\{V, snk(e)\}$ does not introduce a cycle, and $\rho(\{src(e), snk(e)\})$ divides $\rho(\{V, snk(e)\})$.

Proof: Let $X = src(e)$ and $Y = snk(e)$. From Lemma 1, there exists an adjacent pair $\{W_1, Y\}$ such that (a). $\rho(\{X, Y\})$ divides $\rho(\{W_1, Y\})$, and (b). there is a path p_1 from X to W_1. If $\{W_1, Y\}$ introduces a cycle, then again from Lemma 1, we have $\{W_2, Y\}$ such that $\rho(\{W_1, Y\})$ divides $\rho(\{W_2, Y\})$, and there is a path p_2 from W_1 to W_2. Furthermore, $W_2 \neq X$, since $(W_2 = X)$ implies that $\langle(p_1, p_2)\rangle$ is a cycle, and thus that G is not acyclic.

If $(\{W_2, Y\})$ introduces a cycle, then from Lemma 1, we have $(\{W_3, Y\})$ such that $\rho(\{W_2, Y\})$ divides $\rho(\{W_3, Y\})$, and there is a path p_3 from W_2 to W_3. Furthermore $W_3 \neq X$, since otherwise $\langle(p_1, p_2, p_3)\rangle$ is a cycle in G; similarly, $W_3 \neq W_1$, since otherwise $\langle(p_2, p_3)\rangle$ is a cycle. Continuing this process, we obtain a sequence of *distinct* actors $(W_1, W_2, \ldots)$. Since the W_i s are distinct and we are assuming a finite graph, we cannot continue generating W_i s indefinitely. Thus, eventually, we will arrive at a W_n such that $(\{W_n, Y\})$ does not introduce a cycle. Furthermore, by our construction, $\rho(\{X, Y\})$ divides $\rho(\{W_1, Y\})$, and for $i \in \{1, 2, \ldots, (n-1)\}$, $\rho(\{W_i, Y\})$ divides $\rho(\{W_{i+1}, Y\})$. It follows that $\rho(\{X, Y\})$ divides $\rho(\{W_n, Y\})$. *Q.E.D.*

From Corollary 1, we obtain the following theorem, which states that given an APGAN candidate in an SDF graph that has a BMLB schedule, no adjacent pair can have higher repetition count.

Theorem 3: Suppose that G is a connected, delayless SDF graph that has a BMLB schedule, and p is an APGAN candidate in G. Then for all adjacent pairs p' in G, $\rho(p) \geq \rho(p')$.

As an example consider Fig. 8(a), and suppose that the SDF parameters on the graph edges are such that $(\{A, B\})$ is an APGAN candidate — that is, $(\{A, B\})$ does not introduce a cycle and maximizes $\rho(*)$ over all adjacent pairs that do not introduce cycles. Since $(\{B, C\})$ introduces a cycle, the assumption that $(\{A, B\})$ is an APGAN candidate is not sufficient to guarantee that $\rho(\{B, C\}) \leq \rho(\{A, B\})$. However, Theorem 3 guarantees that under the additional assumption that Fig. 8(a) has a BMLB schedule, $\rho(\{B, C\})$ is guaranteed not to exceed $\rho(\{A, B\})$.

Fig. 8(b) shows a case where this additional assumption is violated. Here, $\mathbf{q}(A, B, C, D) = (2, 4, 8, 1)$. Clearly, four invocations of B must fire before a single invocation of C can fire, and thus for any valid schedule S, $max\_tokens((B, C), S) \geq 4 \times 2 = 8 > BMLB((B, C))$; consequently, Fig. 8(b) cannot have a BMLB schedule. It is also easily verified that among the three adjacent pairs in Fig. 8(b) that do not introduce cycles, $\{A, B\}$ is the only APGAN candidate, and $(\rho(\{B, C\}) = 4)$, while $\rho(\{A, B\}) = 2$. Thus, Theorem 3 does not generally hold if we relax the assumption that the graph in question has a BMLB schedule.

Proof of Theorem 3: (By contraposition.) Suppose that $\rho(p') > \rho(p)$. Then since p is an APGAN candidate, p' must introduce a cycle. From Corollary 1, there exists an adjacent pair p'' such that p'' does not introduce a cycle, and $\rho(p')$ divides $\rho(p'')$. It follows that $\rho(p'') > \rho(p)$. Since p'' does not introduce a cycle, p cannot be an APGAN candidate. *Q.E.D.*

Lemma 2: Suppose that $G = (V, E)$ is a consistent, connected SDF graph, $R \subseteq V$ is a subset of actors such that $C \equiv subgraph(R)$ is connected, and $X, Y, Z \in R$. Then

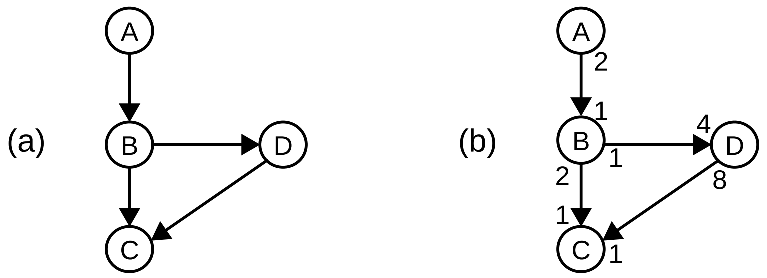

Figure 8. Examples used to illustrate Theorem 3.

$$(gcd(\{\mathbf{q}_C(X), \mathbf{q}_C(Y)\})\ divides\ gcd(\{\mathbf{q}_C(Y), \mathbf{q}_C(Z)\}))\Rightarrow$$

$$(gcd(\{\mathbf{q}_G(X), \mathbf{q}_G(Y)\})\ divides\ gcd(\{\mathbf{q}_G(Y), \mathbf{q}_G(Z)\}))\,.$$

Proof: This result is a straightforward consequence of Fact 4(b). See [2] for details.

The following lemma states that in a connected SDF graph that contains exactly three actors, and that has a BMLB schedule, the repetition count can exceed unity for at most one adjacent pair. For example, consider the three-actor graph in Fig. 9. Here, $\mathbf{q}(A, B, C) = (6, 2, 3)$, and $(2(3A)B)(3C)$ is a BMLB schedule. The two pairs of adjacent actors $\{A, B\}$ and $\{B, C\}$ have repetition counts of 2 and 1, respectively. Thus, we see that only one adjacent pair has a repetition count that exceeds unity.

Lemma 3: Suppose that (a). G is a connected, consistent, delayless SDF graph that consists of exactly three distinct actors X, Y and Z; (b). X is a predecessor of Y; (c). Z is adjacent to $P \in \{X, Y\}$; (d). $\rho_G(\{X, Y\}) \ge \rho_G(\{P, Z\})$; and (e). G has a BMLB schedule. Then, $\rho_G(\{P, Z\}) = 1$.

Proof: For simplicity, assume that $P = Y$, and that Z is a successor of Y. The other three possible cases — ($P = Y$, Z is a predecessor of Y), and ($P = X$, Z is a predecessor or successor of X) — can be handled by simple adaptations of this argument.

Let e_{xy} be an edge directed from X to Y, and let e_{yz} be an edge directed from Y to Z. From Theorem 1, there exists a BMLB R-schedule S_R for G. Since G contains only three actors, G has exactly two R-schedules, and it is easily verified that either S_R is of the form $(i_1X)(i_2(i_3Y)(i_4Z))$, or it has the form $(j_1(j_2X)(j_3Y))(j_4Z)$.

If $S_R = (i_1X)(i_2(i_3Y)(i_4Z))$, then $max\_tokens(e_{xy}, S_R) = TNSE(e_{xy})$, and thus from Fact 8, we have that $TNSE(e_{xy}) = TNSE(e_{xy})/\rho(\{X, Y\})$, which implies that $\rho(\{X, Y\}) = 1$. From Assumption (d), it follows that $\rho(\{Y, Z\}) = 1$.

Conversely, suppose that $S_R = (j_1(j_2X)(j_3Y))(j_4Z)$. Then

$max\_tokens(e_{yz}, S_R) = TNSE(e_{yz})$, so from Fact 8, we have that

$TNSE(e_{yz}) = TNSE(e_{yz})/\rho(\{Y, Z\})$, which implies the desired result. *Q.E.D.*

The following theorem guarantees that whenever an APGAN instance performs a clustering oper-

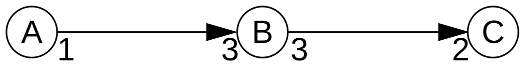

Figure 9. An illustration of Lemma 3.

ation on a top-level graph that has a BMLB schedule, the adjacent pair selected satisfies the proper clustering condition in the top-level graph. For example in Fig. 5(a), $\{A, B\}$ and $\{B, C\}$ are APGAN candidates, and it is easily verified from the repetitions vector $\mathbf{q}(A, B, C, D, E) = (6, 2, 4, 5, 1)$ that both of these adjacent pairs satisfy the proper clustering condition in Fig. 5(a). Similarly, for Fig. 5(b) we have $\mathbf{q}(\Omega_1, C, D, E) = (2, 4, 5, 1)$, and thus $\{\Omega_1, C\}$ is the only APGAN candidate. Thus, Theorem 4 guarantees that $\{\Omega_1, C\}$ satisfies the proper clustering condition in Fig. 5(b).

Theorem 4: Suppose G is a connected, delayless SDF graph that has a BMLB schedule, and $\{X, Y\}$ is an APGAN candidate in G. Then $\{X, Y\}$ satisfies the proper clustering condition in G.

Proof: Let $Z \notin \{X, Y\}$ be an actor that is adjacent to some $P \in \{X, Y\}$; let $C = \mathit{subgraph}(\{X, Y, Z\})$, and observe from Fact 3 that C has a BMLB schedule. From Theorem 3, $\rho_G(\{Z, P\}) \le \rho_G(\{X, Y\})$, and from Fact 4(b), it follows that $\rho_C(\{Z, P\}) \le \rho_C(\{X, Y\})$. Applying Lemma 3 to the three-actor graph C, we see that $\rho_C(\{Z, P\}) = 1$, and thus from Lemma 2, $\rho_G(\{Z, P\})$ divides $\rho_G(\{X, Y\})$. *Q.E.D.*

6 The Optimality of APGAN for a Class of Graphs

In this section, we use the main the results of Section 5 to show that for an acyclic SDF graph (V, E) that has a BMLB schedule, and that satisfies $\mathit{delay}(e) < \eta(e)$, for all $e \in E$, any APGAN instance is guaranteed to construct a BMLB schedule.

In Section 5, we showed 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. This is because although clustering an adjacent pair that satisfies the proper clustering condition preserves the BMLB value on each edge, it does not necessarily preserve the existence of a BMLB schedule [2].

Fortunately, the assumption that the adjacent pair being clustered has maximum repetition count is sufficient to preserve the existence of a BMLB schedule. This is established by the following theorem.

Theorem 5: Suppose that $G = (V, E)$ is a connected, consistent, delayless SDF graph with $|V| > 1$; G has a BMLB schedule; and $\{X, Y\}$ is an APGAN candidate in G. Then $\mathit{cluster}_G(\{X, Y\})$ has a BMLB schedule.

Proof: We assume without loss of generality that X is a predecessor of Y, and we prove this theorem by induction on $|V|$. Clearly, the theorem holds trivially for $|V| = 2$, since in this case, $cluster_G(\{X, Y\})$ contains no edges. Now suppose that the theorem holds for $|V| = 2, 3, \ldots, k$, and consider the case $|V| = (k+1)$.

Define $G_c = cluster_G(\{X, Y\}, \Omega)$, and let S_R be a BMLB R-schedule for G; the existence of such a schedule is guaranteed by Theorem 1. Since S_R is an R-schedule and $|V| > 2$, S_R is of the form $(i_1 B_1)(i_2 B_2)$.

Now suppose that $X, Y \in actors(B_1)$, and let $C_1, C_2, \ldots, C_n$ denote the connected components of $subgraph(actors(B_1))$. Observe that from Fact 3, $S_i = projection((i_1 B_1), C_i)$ is a BMLB schedule for each C_i. Let C_j denote that connected component that contains X and Y. Then, since $|C_j| \le k$, we can apply Theorem 5 with $|V| = |C_j|$ to obtain a BMLB schedule S^* for $cluster(\{X, Y\}, subgraph(C_j))$, and from Fact 2, we can assume without loss of generality that $J(S^*) = J(S_j)$. Then, it is easily verified that $S_1 S_2 \ldots S_{j-1} S^* S_{j+1} S_{j+2} \ldots S_n (i_2 B_2)$ is a BMLB schedule for G_c. A similar argument can be applied to establish the existence of a BMLB schedule for G_c when $X, Y \in actors(B_2)$.

Now suppose that $X \in actors(B_1)$ and $Y \in actors(B_2)$, and let e_{xy} be an edge directed from X to Y. Also, let E_c denote the set of edges in G_c, and for each $e \in E_c$, let e' denote the corresponding edge in G. Clearly $max\_tokens(e_{xy}, S_R) = TNSE(e_{xy})$, and thus, since S_R is a BMLB schedule, we have from Fact 8 that $\rho_G(\{X, Y\}) = 1$. From Theorem 3, it follows that $\rho_G\{X', Y'\} = 1$ for all adjacent pairs $\{X', Y'\}$ in G. Thus, from Fact 8,

$$BMLB(e') = TNSE_G(e') \text{ for all } e' \in E. \tag{6}$$

Let $(X_1, X_2, \ldots, X_n)$ be a any topological sort for G_c. Then clearly,

$S_c = (\mathbf{q}_{G_c}(X_1))(\mathbf{q}_{G_c}(X_2))\ldots(\mathbf{q}_{G_c}(X_n))$ is a valid single appearance schedule for G_c, and

$$buffer\_memory(S_c) = \sum_{e \in E_c} TNSE_{G_c}(e)$$

$$= \sum_{e \in E_c} TNSE_G(e') \qquad \text{(from Fact 5)}$$

$$= \sum_{e \in E_c} BMLB(e') \qquad \text{(from (6))}$$

$$= \sum_{e \in E_c} BMLB(e). \qquad \text{(from Theorems 2 and 4)}$$

Thus, S_c is a BMLB schedule for G_c. *Q.E.D.*

We are now able to establish our result on the optimality of APGAN.

Lemma 4: Suppose that $G = (V, E)$ is a connected, consistent, delayless SDF graph that has a BMLB schedule; P is an APGAN instance; and $S_P(G)$ is the schedule obtained by applying P to G. Then $S_P(G)$ is a BMLB schedule for G.

Proof: By definition, P repeatedly clusters APGAN candidates until the top-level graph consists of only one actor. From Theorem 4, the first adjacent pair p_1 clustered when P is applied to G satisfies the proper clustering condition, and thus from Theorem 5, the top level graph T_1 that results from the first clustering operation has a BMLB schedule. Since T_1 has a BMLB schedule we can again apply Theorems 4 and 5 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 successively 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. Thus, from Theorem 2, $BMLB(e') = BMLB(e)$, whenever e' and e are corresponding edges associated with a clusterization step of P. It follows from Fact 9 that $buffer\_memory(S_P(G)) = \sum_{e \in E} BMLB(e)$, and thus $S_P(G)$ is a BMLB schedule for G. *Q.E.D.*

The following theorem gives our general specification of the optimality of APGAN.

Theorem 6: Suppose that $G = (V, E)$ is a connected, consistent, acyclic SDF graph that has a BMLB schedule; $delay(e) < \eta(e)$ for all $e \in E$; P is an APGAN instance; and $S_P(G)$ is the schedule obtained by applying P to G. Then $S_P(G)$ is a BMLB schedule for G.

Proof: Let G' denote the delayless version of G, and let P' be the APGAN instance that begins by checking whether or not the input graph G_I is equal to G', applies P to G if $G_I = G'$, and applies P to G_I if $G_I \neq G'$. Thus, P' returns $S_P(G)$ if $G_I = G'$, and returns $S_P(G_I)$ otherwise.

Now, since edge delays do not affect the repetition counts of adjacent pairs, the sequence of adjacent pairs in G that are clustered by P' when $G_I = G'$ is a sequence of maximum repetition-count clusterizations of G'. Thus, clearly P' is an APGAN instance. From Fact 7, a BMLB schedule exists for G', and thus, from Lemma 4 and Fact 7, $S_{P'}(G')$ must be a BMLB schedule for G. But by construction, $S_{P'}(G') = S_P(G)$. *Q.E.D.*

To summarize, the BMLB bound is a lower bound on the amount of buffering required by any valid single appearance schedule for an acyclic SDF graph. However, a schedule that meets this lower bound may or may not exist. The above theorem says that whenever such a schedule exists, APGAN will find it, provided that $delay(e) < \eta(e)$. If such a schedule does not exist, then there is some schedule that minimizes the buffering requirement (and this is greater than the BMLB). However, APGAN will not necessarily find this schedule for such a graph. While the result above is of considerable intellectual interest by itself, we will show in Section 8 that there are in fact a large class of practical SDF graphs that fall into the class of graphs having BMLB schedules; for this class of graphs, APGAN gives memory-optimal schedules.

7 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 [14], we proposed an alternative top-down 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. The partitions are constructed by finding the *cut* (a partition of the set of actors) of the graph across which the minimum amount of data is transferred and scheduling the resulting halves recursively. 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. In addition, we would also like to impose the constraint that the partition that results be 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 (see Fact 4). In this section, we give an overview of the RPMC technique.

Suppose that $G = (V, E)$ is a connected, consistent SDF graph. A **cut** of G is a partition of V into two disjoint sets V_L and V_R. Define $G_L = subgraph(V_L)$ and $G_R = subgraph(V_R)$ to be the subgraphs produced by the cut. The cut is **legal** if for all edges e *crossing* the cut (that is all edges that are not contained in $subgraph(V_L)$ nor $subgraph(V_R)$), 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 $|V_R| \le K$, $|V_L| \le K$. The weight of an edge e is defined as $w(e) = TNSE(e)$.

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 with the weights defined as above. Since

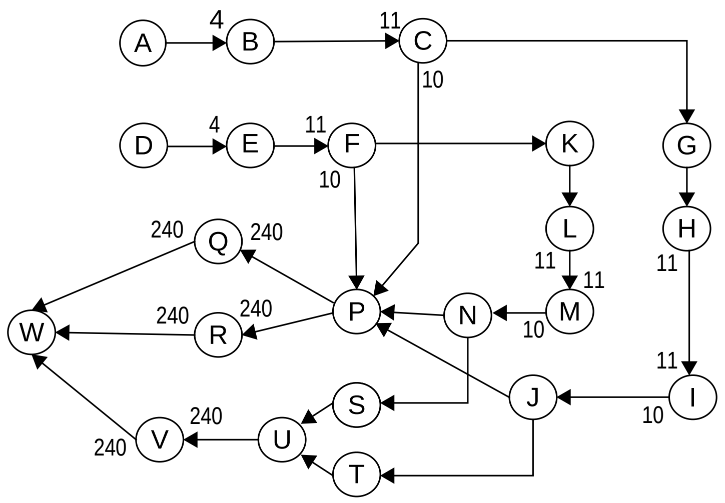

Figure 10. SDF abstraction for satellite receiver application from [Ritz95]

the related problem of finding a minimum cut (not necessarily legal) into bounded sets is NP-complete [8], and the problem of finding an acyclic partition of a graph is NP-complete [8], we believe this problem to be NP-complete as well even though we have not discovered a proof. 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 [6] do not work because the minimum cut given by the max-flow-min-cut theorem may not be legal and may not be bounded [14]. Hence, we give a heuristic solution for finding legal minimum cuts into bounded sets. See [14] for a description and pseudocode specification of the heuristic.

RPMC proceeds by partitioning the graph by computing the legal minimum cut and forming the schedule $(\rho(V_L)S_L)(\rho(V_R)S_R)$, where the schedules S_L, S_R are obtained recursively by partitioning G_L and G_R. It can be shown that the running time of RPMC is given by $O(|V|^3)$ [14].

The RPMC algorithm is easily extended to efficiently handle nonzero delays. See [14] for details.

8 Experimental Results

Figure 10 shows a practical example of a graph that is in the class of SDF graphs that have a BMLB schedule. The graph is an abstraction for a satellite receiver implementation and is taken from [18]. The graph is annotated with the produced/consumed numbers wherever they are different from unity. It is interesting to note that a shared-buffer implementation of the flat single appearance schedule for this graph would require a buffer of size 2040 [18] while APGAN generates a BMLB schedule having a total buffering requirement of 1540 (using a buffer on every edge of-course).

Table 1 shows experimental results on the performance of APGAN and RPMC that we have developed for several practical examples of acyclic, multirate SDF graphs. The column titled "Average Random" represents the average buffer memory requirement obtained by considering 100 random topological sorts and applying GDPPO (see Subsection 2.3) to each. The data for APGAN and RPMC also includes the effect of GDPPO. The "BMUB" column gives a simple upper bound on the buffer memory requirement. This bound is the sum of $(delay(e) + TNSE(e))$ taken over all edges.

All of the systems shown below are acyclic graphs. The data for APGAN and RPMC also includes the effect of GDPPO. As can be seen, APGAN achieves the BMLB on 5 of the 10 examples, outperforming RPMC in these cases. Particularly interesting are the last three examples in the table, 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 buffer memory requirements 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.

Table 2 shows more detailed statistics for the performance of randomly obtained topological sorts. The column titled "APGAN < random" represents the percentage of random schedules that had a buffer memory requirement greater than that obtained by APGAN. The last two columns give the mean number

Table 1. Performance of the two heuristics on various acyclic graphs.

System	BMUB	BMLB	APGAN	RPMC	Average Random	Graph size(nodes/arcs)
Fractional decimation	61	47	47	52	52	26/30
Laplacian pyramid	115	95	99	99	102	12/13
Nonuniform filterbank (1/3,2/3 splits, 4 channels)	466	85	137	128	172	27/29
Nonuniform filterbank (1/3,2/3 splits, 6 channels)	4853	224	756	589	1025	43/47
QMF nonuniform-tree filterbank	284	154	160	171	177	42/45
QMF filterbank (one-sided tree)	162	102	108	110	112	20/22
QMF analysis only	248	35	35	35	43	26/25
QMF Tree filterbank (4 channels)	84	46	46	55	53	32/34
QMF Tree filterbank (8 channels)	152	78	78	87	93	44/50
QMF Tree filterbank (16 channels)	400	166	166	200	227	92/106

of random schedules needed to outperform these heuristics. A dash indicates that no random schedules were found that had a buffer memory requirement lower that obtained by the corresponding heuristic.

While the above results on practical examples are encouraging, we have also tested the heuristics on a large number of randomly generated 50-actor SDF graphs. These graphs were sparse, having about 100 edges on average. Table 3 summarizes the performance of these heuristics, both against each other, and against randomly generated schedules. As can be seen, RPMC outperforms APGAN on these random graphs almost two-thirds of the time. We choose to compare these heuristics against 2 random schedules because measurements of the actual running time on 50-vertex graphs showed that we can construct and examine approximately 2 random schedules in the time it takes for either APGAN or RPMC to construct its schedule and have it post-optimized by GDPPO. The comparison against 4 random schedules shows that in general, the performance of these heuristics goes down if a large number of random schedules are inspected. Of course, this also entails a proportionate increase in running time. However, as shown on practical examples already, it is unlikely that even picking a large number of schedules randomly will give better results than these heuristics since practical graphs usually have a significant amount of structure (as opposed to random graphs) that the heuristics can exploit well. Thus, the comparisons against 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 SDF graphs that we examine, APGAN performs well on graphs that have relatively regular topological

Table 2. Performance of 100 random schedules against the heuristics

Comparison with random schedules (100 trials)	APGAN < random	APGAN = random	RPMC < random	RPMC = random	avg. to beat APGAN	avg. to beat RPMC
Fractional decimation	92%	8%	54%	13%	----	3
Laplacian pyramid	74%	26%	74%	26%	----	----
Nonunif. filterbank (1/3,2/3 splits, 4, ch.)	100%	0%	100%	0%	----	----
Nonunif. filterbank (1/3,2/3 splits, 6 ch.)	100%	0%	100%	0%	----	----
QMF nonuniform-tree filterbank	100%	0%	81%	7%	----	8
QMF filterbank (1-sided tree)	100%	0%	77%	23%	----	----
QMF analysis only	99%	1%	99%	1%	----	----
QMF Tree filterbank (4 channels)	100%	0%	16%	13%	----	1.4
QMF Tree filterbank (8 channels)	100%	0%	87%	3%	----	9.1
QMF Tree filterbank (16 channels)	100%	0%	96%	1%	----	22.3

structures and rate changes, like the uniform QMF filterbanks, 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, from the last two rows of Table 3, we see that there is a significant proportion of random graphs for which APGAN outperforms RPMC by a margin of over 10%, 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 6, and the ability of APGAN to exploit regularity that arises frequently in practical applications.

9 Related Work

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

Table 3. Performance of the two heuristics on random graphs

RPMC < APGAN	63%
APGAN < RPMC	37%
RPMC < min(2 random)	83%
APGAN < min(2 random)	68%
RPMC < min(4 random)	75%
APGAN < min(4 random)	61%
min(RPMC,APGAN) < min(4 random)	87%
RPMC < APGAN by more than 10%	45%
RPMC < APGAN by more than 20%	35%
APGAN < RPMC by more than 10%	23%
APGAN < RPMC by more than 20%	14%

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 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.

10 Conclusions

In this paper, we have addressed the problem of constructing a software implementation of an SDF graph that requires minimal data memory from among the set of implementations that require minimum code size. We have developed a fundamental lower bound, called the BMLB, on the amount of data memory required for a minimum code size implementation of an SDF graph; we have presented an efficient adaptation to acyclic graphs, called APGAN, of the PGAN technique developed in [4]; and we have shown that for a certain class of graphs, which includes all delayless graphs, APGAN is guaranteed to achieve the BMLB whenever it is achievable. We have presented the results of an extensive experimental study in which we evaluate the performance of APGAN and RPMC, a top-down technique developed in [14] that is based on recursively applying a generalized minimum-cut operation. Based on this study, we have concluded that APGAN and RPMC complement each other, and thus, techniques should be investigated for efficiently combining the methods of APGAN and RPMC, and that 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. A version of APGAN has been implemented by Cadence Design Systems Inc. in their Signal Processing Worksystem and we have implemented both of these algorithms in the Ptolemy programming environment at UC Berkeley and will be making them available in the next release.

The solutions developed in this paper have focused on acyclic SDF graphs. These techniques can be applied in a limited way to general SDF graphs [2]. More thorough techniques for jointly optimizing code and data for general SDF graphs is a topic for further study.

References

[1] M. Ade, R. Lauwereins, J. A. Peperstraete, “Buffer Memory Requirements in DSP Applications,” *IEEE Wkshp. on Rapid System Prototyping*, 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 UCB/ERL M95/3, Electronics Research Laboratory, University of California at Berkeley, Jan., 1995. WWW URL: http://ptolemy.eecs.berkeley.edu/papers/PganRpmcDppo/.

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

[4] S. S. Bhattacharyya, E. A. Lee, “Scheduling Synchronous Dataflow Graphs for Efficient Looping,” *Jo. of VLSI Signal Processing*, Dec., 1993.

[5] JJ. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, “Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems”, *International Journal of Computer Simulation*, Jan. 1995.

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

[7] J. Fabri, *Automatic Storage Optimization*, UMI Research Press, 1982.

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

[9] R. Govindarajan, G. R. Gao, P. Desai, “Minimizing Memory Requirements in Rate-Optimal Schedules,” *Proc. of the Intl. Conf. on Application Specific Array Processors*, San Francisco, Aug., 1994.

[10] B. W. Kernighan, S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs,” *Bell System Technical Journal*, Feb., 1970.

[11] 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.

[12] E. A. Lee, W. H. Ho, E. Goei, J. Bier, S. S. Bhattacharyya, “Gabriel: A Design Environment for DSP,” *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Nov., 1989.

[13] E. A. Lee, D. G. Messerschmitt, “Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing,” *IEEE Trans. on Computers*, Feb., 1987.

[14] P. K. Murthy, S. S. Bhattacharyya, E. A. Lee, “Combined Code and Data Minimization for Synchronous Dataflow Programs”, Memorandum UCB/ERL M94/93, Electronics Research Laboratory, University of California at Berkeley, Nov., 1994, WWW URL: http://ptolemy.eecs.berkeley.edu/papers/jointCodeDataMinimize/.

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

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

[17] S. Ritz, S. Pankert, and H. Meyr, “High Level Software Synthesis for Signal Processing Systems,” *Proc. of the Intl. Conf. on Application Specific Array Processors*, Aug., 1992.

[18] S. Ritz, M. Willems, H. Meyr, “Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis,” *Proceedings of the ICASSP 95*, Detroit, Michigan, May 1995.

Glossary

$\eta(e)$	Given an SDF edge e, $\eta(e) = \frac{prod(e)cons(e)}{gcd(\{prod(e), cons(e)\})}$.
$\rho(Z)$	Given a subset of actors Z, $\rho(Z) = gcd(\mathbf{q}(A) \mid A \in Z)$.
Blocking factor	For each valid schedule S for a connected SDF graph, there is a positive integer J such that S invokes each actor A exactly $J\mathbf{q}(A)$ times. The constant J is the called the blocking factor of S.
BMLB	Buffer memory lower bound. Given an SDF edge e, $BMLB(e)$ is a lower bound on $max\_tokens(e, S)$ over all valid single appearance schedules for any consistent SDF graph that contains e. The BMLB of an SDF graph G is the sum of the BMLB values over all edges in G. A BMLB schedule for G is a valid single appearance schedule whose buffer memory requirement equals the BMLB of G.
$cluster_G(Z, \Omega)$	The SDF graph that results from clustering the subset of actors Z in the SDF graph G into the actor Ω. We may write $cluster_G(Z)$ when there is no ambiguity.
GDPPO	Generalized dynamic programming post optimization. Applying GDPPO to a single appearance schedule S yields a schedule that has a buffer memory requirement that is less than or equal to the buffer memory requirement of every valid single appearance schedule that has the same lexical ordering as S.
Introduces a cycle	A subset of actors Z in a connected, consistent, acyclic SDF graph G introduces a cycle if $cluster_G(Z)$ contains one or more cycles.
$J(S)$	Denotes the blocking factor of the valid schedule S.
$max\_tokens(e, S)$	Given a valid schedule S and an edge e, $max\_tokens(e, S)$ denotes the maximum number of tokens that are queued on e during an execution of S.
$\mathbf{q}$	Given a connected, consistent SDF graph G and an actor A in G, $\mathbf{q}(A)$ gives the minimum number of times that A must be invoked in a valid schedule for G.
$TNSE(e)$	Total number of samples exchanged on an SDF edge. Given an SDF edge e in a consistent SDF graph, $TNSE(e) = \mathbf{q}(src(e))prod(e)$.