Memorandum UCB/ERL M95/3, Electronics Research Lab., University of California, Berkeley
January, 1995

Two Complementary Heuristics for Translating Graphical DSP Programs into Minimum Memory Implementations

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

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 compare 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.

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., 179 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. Here, we focus mainly on programs that are represented as *acyclic* SDF graphs; a large class of important DSP applications (some examples will be given in the sequel) can be implemented with such programs.

Numerous DSP design environments, including a number of commercial tools, support SDF or closely related models [10, 12, 15, 16, 17]. In SDF, a program is represented by a directed graph in which each vertex (**actor**) represents a computation, and an edge specifies a FIFO communication channel. In SDF, each actor produces (consumes) a fixed number of data values (**tokens**) onto (from) each output (input) edge per invocation.

Figure 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 on this edge. 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 (when there is no ambiguity, we do not distinguish between the FIFO buffer associated with and edge and the edge itself). Also, $prod(e)$ denotes the number of tokens produced by $src(e)$, and $cons(e)$ denotes the number of tokens consumed 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 period schedule**, or simply a “valid schedule.” Corresponding to each actor in the schedule

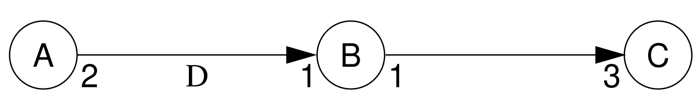

Figure 1. A simple SDF graph.

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 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 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

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

or equivalently, from (1),

$$TNSE(e) = \mathbf{q}_G(snk(e)) \times cons(e)\,. \tag{3}$$

Thus, $TNSE(e)$ is the total number of tokens produced onto (consumed from) e in any minimal, valid schedule for G.

For Figure 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 Figure 1 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. 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."

Note that in the valid schedule $B(2AB)CA(3B)C$, B is allowed to fire first because of the unit delay on the edge (A, B).

A more compact valid schedule for Figure 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 [2].

Given an SDF graph G, a valid schedule S, and an edge e in G, we define $max\_tokens(e, S, G)$ to denote the maximum number of tokens that are queued on e during an execution of S. If G is understood, then we may write $max\_tokens(e, S)$ in place of $max\_tokens(e, S, G)$. For example if for Figure 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, $buffer\_memory(S_1) = 7 + 6 = 13$, while for the "nested" schedule S_2, we have $buffer\_memory(S_2) = 3 + 6 = 9$.

In the model of buffering implied by our "buffer memory requirement" measure, each buffer is mapped to a contiguous and independent 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 any 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]. The model of buffering discussed in this paper is used in the SDF-based code generation environments described in [10, 12, 17].

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. In this paper, we call such a schedule an **optimal schedule**. 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 [3] 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]. We call this top-down heuristic *Recursive Partitioning Based on Minimum Cuts (RPMC).* On all of the applications that we considered, both heuristics produced excellent results and consistently outperformed randomly generated schedules. However, we have found applications where RPMC significantly outperforms APGAN, and others where APGAN significantly outperforms RPMC. Furthermore, on a diverse collection of large randomly-generated SDF graphs, we have found that RPMC outperforms APGAN by a margin of over 10% on 45% of the random graphs, while APGAN outperforms RPMC by over 10% on 23% of the random graphs. The conclusions that we postulate based on our study are that 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.

2 Background

For reference, much of the terminology that is introduced in this and subsequent sections is summarized in the glossary 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 usually 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 $G = (V, E)$, we say that 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 we say that 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 called 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 we say that X is an **ancestor** of Y, and Y is a **descendant** of X. If X is neither a descendant nor an ancestor of Y, we say that X is **independent of** Y. A path that is directed from a vertex to itself is called 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** $(snk(e) = v_j)) \Rightarrow (i < j)$. Given an SDF graph and an actor X in this graph, $ancs(X)$ denotes the set of ancestors of X, and $desc(X)$ denotes the set of descendants of X.

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 $E' = \{\text{the delayless version of } e \mid e \in E\}$. 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, we say that S_0 is **contained in** S, and we say that 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, we define $inv(S_0, S)$ to be 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 will occasionally need to refer to the relative lexical positions of actors in a single appearance schedule. For this purpose, we define $position(X, S)$ to be the number of actors that lexically precede X in the single appearance schedule S. Observe that no ambiguity arises in this definition since we apply it only to single appearance schedules. For example, if $S = (2(3B)(5C))(7A)$, then $position(A, S) = 2$, $position(B, S) = 0$, and $position(C, S) = 1$. Formally, we define the **lexical ordering** of a single appearance schedule S, denoted $lexorder(S)$, to be 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 $position(X, S) < position(Y, S)$.

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 $subgraph(Z, G)$. 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 $subgraph(Z, G)$, 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 2: 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 $\rho_G(Z) \equiv gcd\left(\{ \mathbf{q}_G(A) \mid A \in Z \} \right)$, and we refer to this quantity as the **repetition count** of Z. We show below (Fact 3(a)) that the repetition count of Z can be viewed as the number of times that a minimal schedule for G invokes the "subsystem" corresponding to Z.

We will extensively apply the concept of "clustering" a subgraph in an SDF graph, which was introduced in [11]. 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 $E' = E - (\{e \mid (src(e) \in Z) \textbf{ or } (snk(e) \in Z)\}) + E^*$, 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

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

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 $cluster(Z, G, \Omega)$, or simply $cluster(Z, G)$. Intuitively, an invocation of Ω in $cluster(Z, G, \Omega)$ corresponds to an invocation of a minimal valid schedule for $subgraph(Z)$ in G. We say that Z is **clusterable** if $cluster(Z, G)$ is consistent, and if G is acyclic, we say that Z **introduces a cycle** if *cluster* (Z, G) contains one or more cycles. Figure 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 the repetitions vector of the original SDF graph. The proofs of both parts can be found in [2].

Fact 3: (a). If $G = (V, E)$ is a connected, consistent SDF graph, $Z \subseteq V$, and $G' = \mathit{cluster}(Z, G, \Omega)$, 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 3(a) together with the definition of clustering immediately yields

Fact 4: If G and G' are as in Fact 3(a), then for each edge e in G', $\mathit{TNSE}(e, G') = \mathit{TNSE}(e', G)$, where e' is the edge in G that corresponds to e.

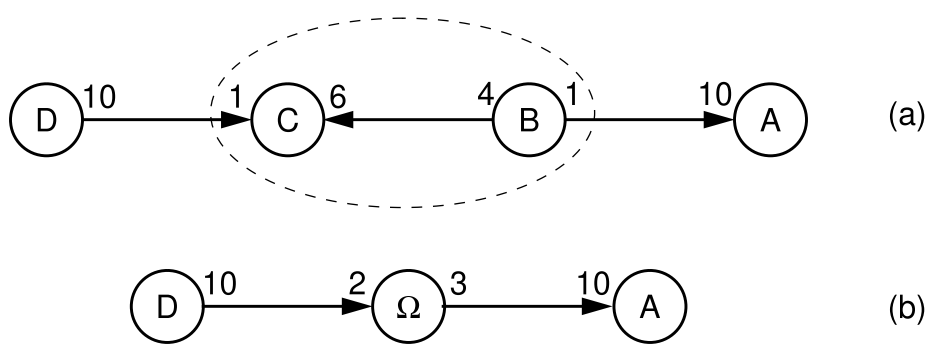

Figure 2. An example of clustering. In (b), we have $\mathit{cluster}(\{B, C\}, G, \Omega)$, where G denotes the SDF graph in (a). Here, $\mathbf{q}_G(A, B, C, D) = (3, 30, 20, 2)$, and thus, $\rho_G(\{B, C\}) = 10$.

We will use following fact, which is developed in [2]. This fact provides a simple test for the validity a schedule transformation called the *factoring transformation*. In [2] it is shown that this transformation can significantly reduce the buffer memory requirement in a single appearance schedule.

Fact 5: Suppose that S is a valid schedule for an SDF graph G, and suppose that $L = (m\,(n_1 S_1)\;(n_2 S_2)\;\ldots\;(n_k S_k))$ is a schedule loop in S of any nesting depth such that $(1 \le i < j \le k) \Rightarrow \mathit{actors}\,(S_i) \cap \mathit{actors}\,(S_j) = \varnothing$. Suppose also that γ is any positive integer that divides $n_1, n_2, \ldots, n_k$; let L' denote the schedule loop $\left(\gamma m\left(\gamma^{-1} n_1 S_1\right)\left(\gamma^{-1} n_2 S_2\right)\ldots\left(\gamma^{-1} n_k S_k\right)\right)$; and let S' denote the schedule that results from replacing L with L' in S. Then

(a). S' is a valid schedule for G; *and*

(b). $\mathit{max\_tokens}\,(e, S') \le \mathit{max\_tokens}\,(e, S)$, for each edge e in G.

If Λ is either a schedule loop or a looped schedule, we say that Λ is **non-coprime** if all iterands of Λ are schedule loops and there exists an integer $j > 1$ that divides all of the iteration counts of the iterands of Λ. If Λ is not non-coprime, we say that Λ is **coprime**. The distinction between the conditions for a schedule loop and a looped schedule arise because our convention in manipulating looped schedules is to drop the outermost loop (mS) (usually $m = \infty$) that encapsulates a valid schedule S in the final implementation. If we retain the outermost loop, then it is equivalent to say that S is a coprime looped schedule if (mS) is a coprime schedule loop. For example, the schedule loops $(3\,(4A)\;(2B))$ and $(10\,(7C))$ are both non-coprime, while the loops $(5\,(3A)\;(7B))$ and $(70C)$ are coprime. Similarly, the looped schedules $(4AB)$ and $(6AB)\;(3C)$ are both non-coprime, while the schedules $A\,(7B)\;(7C)$ and $(2A)\;(3B)$ are coprime.

From our discussion of Fact 5, we know that non-coprime schedules or loops may result in much higher buffer memory requirements than their factored counterparts. Given a single appearance schedule S, we say that S is **fully reduced** if S is coprime and every schedule loop contained in S is coprime.

In [2], it is shown that every fully reduced schedule has unit blocking factor. This result is easily generalized to yield the following fact, which we will use in Section 3.

Fact 6: Suppose that S is a valid, fully reduced schedule, and $L = (i_L B_L)$ is a schedule loop in S. Then for each $A \in actors(B_L)$, $inv(A, B_L) = \dfrac{\mathbf{q}(A)}{gcd\left(\{\mathbf{q}(A') \mid (A' \in actors(B_L))\}\right)}$.

Proof: Observe that for each $A \in actors(B_L)$, we have $J(S)\,\mathbf{q}(A) = k \times inv(A, B_L)$, where $k = inv(B_L, S)$. Thus, it suffices to show that $gcd\left(\{inv(A', B_L) \mid A' \in actors(B_L)\}\right) = 1$.

First suppose that not all iterands of L are schedule loops. Then $inv(A, B_L) = 1$ for some $A \in actors(B_L)$, and we are done.

Now suppose that all iterands of L are schedule loops, and suppose that j is an arbitrary integer that is greater than one. Then, since S is fully reduced, j does not divide at least one of the iteration counts associated with the iterands of L. Define $i_0 = 1$ and let L_1 denote one of the iterands of L whose iteration count i_1 is not divisible by $j = j/\left(gcd\left(\{j, i_0\}\right)\right)$. Again, since S is fully reduced, if all iterands of L_1 are schedule loops, then there exists an iterand L_2 of L_1 such that $j = j/\left(gcd\left(\{j, i_0 i_1\}\right)\right)$ does not divide the iteration count i_2 of L_2. Similarly, if all iterands of L_2 are schedule loops, there exists an iterand L_3 of L_2 whose iteration count i_3 is not divisible by $j = j/\left(gcd\left(\{j, i_0 i_1 i_2\}\right)\right)$.

Continuing in this manner, we get a sequence $(L_1, L_2, L_3 \ldots)$ such that the iteration count i_k of each L_k is not divisible by $j = j/\left(gcd\left(\{j, i_0 i_1 \ldots i_{k-1}\}\right)\right)$. Since S contains a finite number of loops, we cannot continue this process indefinitely — for some $m \geq 1$, not all iterands of L_m are schedule loops. Thus, there is an actors A that is an iterand of L_m. Since S is a single appearance schedule,

$$inv(A, B_L) = inv(L_2, L_1)\, inv(L_3, L_2) \ldots inv(L_{m-1}, L_m)\, inv(A, L_m) = i_0 i_1 \ldots i_m. \qquad (4)$$

By our selection of the L_k's, $j / \left(gcd\left(\{ j, i_0 i_1 \ldots i_{m-1} \} \right) \right)$ does not divide i_m, and thus, from (4), j does not divide $inv(A, B_L)$.

We have shown that given any integer $j > 1$, there exists an $A \in actors(B_L)$ such that $inv(A, B_L)$ is not divisible by j. It follows that $gcd\left(\{ inv(A', B_L) \mid A' \in actors(B_L) \} \right) = 1$.

Q.E.D.

Repeated application of the factoring transformation can be used to construct a valid fully reduced single appearance schedule from an arbitrary valid single appearance such that the amount of memory required to implement each edge in the fully reduced schedule is less than or equal to the amount of memory required for the same edge in the original schedule. This is established by the following fact.

Fact 7: Suppose that $G = (V, E)$ is a consistent, connected SDF graph, and S is a single appearance schedule for G. Then there exists a valid, fully reduced schedule S' such that $lexorder(S') = lexorder(S)$, and $max\_tokens(e, S') \leq max\_tokens(e, S)$, for each $e \in E$.

Proof: We prove this fact by construction. Given a looped schedule Ψ, we denote the set of schedule loops in Ψ that are not coprime by $non\text{-}coprime(\Psi)$. Now suppose that S is a valid single appearance schedule for G, and let $\lambda_1 = (m\,(n_1 \Psi_1)\,(n_2 \Psi_2) \ldots (n_k \Psi_k))$ be any innermost member of $non\text{-}coprime(S)$ — that is, λ_1 is non-coprime, but every schedule loop nested within λ_1 is coprime. From Fact 5, replacing λ_1 with

$\lambda_1' = \left(\gamma m\left(\gamma^{-1} n_1 \Psi_1\right)\left(\gamma^{-1} n_2 \Psi_2\right)\ldots\left(\gamma^{-1} n_k \Psi_k\right)\right)$, where $\gamma = gcd\left(\{n_1, n_2, \ldots, n_k\}\right)$, yields another valid single appearance schedule S_1 such that $max\_tokens(e, S_1) \le max\_tokens(e, S)$, for all $e \in E$. Furthermore, λ_1' is coprime, and since every schedule loop nested within λ_1 is coprime, every loop nested within λ_1' is coprime as well. Now let λ_2 be any innermost member of $non\text{-}coprime(S_1)$, and observe that λ_2 cannot equal λ_1'. Fact 5 guarantees a replacement λ_2' for λ_2 in S_1 that leads to another valid single appearance schedule S_2 such that $max\_tokens(e, S_2) \le max\_tokens(e, S)$, for all $e \in E$. If we continue this process, it is clear that no replacement loop λ_k' ever replaces one of the previous replacement loops $\lambda_1', \lambda_2', \ldots, \lambda_{k-1}'$, since these loops and the loops nested within these loops are already coprime. Also, no replacement changes the total number of schedule loops in the schedule. It follows that we can continue this process only a finite number of times — eventually, we will arrive at an S_n such that $non\text{-}coprime(S_n)$ is empty.

Now if S_n is a coprime looped schedule, we are done. Otherwise, S_n is of the form $(p_1 T_1)\ (p_2 T_2) \ldots (p_m T_m)$, where $\gamma' \equiv gcd\left(\{p_1, p_2, \ldots, p_m\}\right) > 1$. Applying Fact 5 to the schedule $(1 S_n) = (1\ (p_1 T_1)\ (p_2 T_2) \ldots (p_m T_m))$, we have that

$$\left(\gamma'\left((\gamma')^{-1} p_1 T_1\right)\left((\gamma')^{-1} p_2 T_2\right)\ldots\left((\gamma')^{-1} p_m T_m\right)\right)$$

is a valid schedule for G. From the definition of a valid schedule, it follows that

$$S_n' \equiv \left((\gamma')^{-1} p_1 T_1\right)\left((\gamma')^{-1} p_2 T_2\right)\ldots\left((\gamma')^{-1} p_m T_m\right)$$

is also a valid schedule, and by our construction of S_n and S_n', S_n' is a coprime single appearance schedule, and all schedule loops in S_n' are coprime. Thus, S_n' is a valid fully reduced schedule for G. Furthermore, since $(1S_n)$ generates the same invocation sequence as S_n clearly $max\_tokens(e, (1S_n)) = max\_tokens(e, S_n)$ for all $e \in E$. From Fact 5, $max\_tokens(e, S_n') \le max\_tokens(e, (1S_n))$ for all $e \in E$, and thus $max\_tokens(e, S_n') \le max\_tokens(e, S)$ for all $e \in E$.

It is easily verified that none of the transformations in our derivation of S_n' affect the lexical ordering, and thus $lexorder(S_n') = lexorder(S)$. *Q.E.D.*

As a consequence of Fact 7, we have that given a valid single appearance schedule, there is a single appearance schedule for any blocking factor such that the memory required for each edge is no greater than the memory required for the same edge with the original schedule. This is established by the following fact.

Fact 8: 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' such that $J(S') = k$, $lexorder(S') = lexorder(S)$, and $max\_tokens(e, S') \le max\_tokens(e, S)$, for each $e \in E$.

Proof: From Fact 7, there is a valid, fully reduced schedule S'' such that $lexorder(S'') = lexorder(S)$, and $max\_tokens(e, S'') \le max\_tokens(e, S)$, for each $e \in E$. Clearly, since S'' is fully reduced, $(1S'')$ is also fully reduced. Thus, applying Fact 6 with $L = (1S'')$, we have that $J(S'') = 1$. Thus (kS'') is a valid schedule that has blocking factor k, and has the same lexical ordering as S. Furthermore, since S'' is a valid schedule, clearly $max\_tokens(e, (kS'')) = max\_tokens(e, S'')$ for all $e \in E$, and thus, $max\_tokens(e, (kS'')) \le max\_tokens(e, S)$ for all $e \in E$. *Q.E.D.*

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

We call a valid single appearance schedule S an **R-schedule** if S satisfies the R-condition, and every schedule loop contained in S satisfies the R-condition.

In [2], it is shown that in a chain-structured SDF graph, whenever a valid single appearance schedule exists, an R-schedule can be derived whose buffer memory requirement is no greater than that of the original schedule. This result is easily generalized to give the following theorem for arbitrary consistent SDF graphs.

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 $lexorder(S_R) = lexorder(S)$.

Proof: We prove this theorem by construction.We use the following notation here: given a schedule loop L and a looped schedule S', we define $nonR(S')$ to be the set of schedule loops in S' that do not satisfy the R-condition; $I(L)$ to be the number of iterands of L; and $C(L)$ to be the iteration count of L. Also, we define $\hat{I}(S') \equiv \sum_{L' \in nonR(S')} I(L')$.

First observe that from Fact 7, there exists a valid fully reduced schedule S_0 for G such that $max\_tokens(e, S_0) \le max\_tokens(e, S)$ for all $e \in E$. Now let $L_0 = (nT_1T_2 \ldots T_m)$ be an innermost loop in $(1S_0)$ [1] that does not satisfy the R-condition; that is, L_0 does not satisfy the R-condition, but all loops nested in L_0 satisfy the R-condition. If $m = 1$, then since S_0 is fully reduced, $L_0 = (n(1T'))$, where $(1T')$ satisfies the R-condition. Let S^* denote the schedule

1. This is the schedule loop whose iteration count is one and whose body is S_0. Any schedule loop of the form (mS_0) is acceptable for the purposes of this proof, and we have chosen $m = 1$ only for simplicity.

that results from replacing L_0 with (nT') in $(1S_0)$. Then clearly, S^* is also valid and fully reduced, and S^* generates the same invocation sequence as S_0, so $max\_tokens(e, S^*) = max\_tokens(e, S_0)$ for all $e \in E$. Also, replacing L_0 with (nT') reduces the number of non-R loops by one, and does not increase the number of iterands of any loop, and thus, $\hat{I}(S^*) < \hat{I}((1S_0))$.

If on the other hand $m \geq 2$, we define $S_a \equiv (1T_1)$ if T_1 is an actor and $S_a \equiv T_1$ if T_1 is a schedule loop. Also, if $T_2, T_3, \ldots, T_m$ are all schedule loops, we define

$$S_b \equiv \left(\gamma\left(\frac{C(T_2)}{\gamma}B_2\right)\left(\frac{C(T_3)}{\gamma}B_3\right)\ldots\left(\frac{C(T_m)}{\gamma}B_m\right)\right),$$

where $\gamma = gcd\Big(\{C(T_2), C(T_3), \ldots, C(T_m)\}\Big)$, and $B_2, B_3, \ldots, B_m$ are the bodies of $T_2, T_3, \ldots, T_m$, respectively; if $T_2, T_3, \ldots, T_m$ are not all schedule loops, we define $S_b \equiv (1T_2T_3\ldots T_m)$. Let S^* be the schedule that results from replacing L_0 with $L_0' = (nS_aS_b)$ in $(1S_0)$. Now, because S_0 is fully reduced, the iteration counts of S_a and S_b must be coprime. Thus, it is easily verified that S^* is a valid, fully reduced schedule and that L_0' satisfies the R-condition, and with the aid of Fact 5, it is also easily verified that $max\_tokens(e, S^*) \leq max\_tokens(e, S_0)$ for all $e \in E$.

Furthermore, observe that S_a and L_0' satisfy the R-condition, but S_b may or may not satisfy the R-condition, depending on L_0. Thus, replacing L_0 with L_0' either reduces the number of loops that do not satisfy the R-condition by one, or it leaves the number of loops that do not satisfy the R-condition unchanged, and we see that either $\hat{I}(S^*) = \hat{I}((1S_0)) - I(L_0)$, or $\hat{I}(S^*) = \hat{I}((1S_0)) - I(L_0) + I(S_b)$. Since $I(S_b) = I(L_0) - 1 < I(L_0)$, we again conclude that $\hat{I}(S^*) < \hat{I}((1S_0))$.

Thus, from $(1S_0)$, we have constructed a valid, fully reduced schedule S^* such that $max\_tokens(e, S^*) \leq max\_tokens(e, S_0) \leq max\_tokens(e, S)$ for all $e \in E$, and $\hat{I}(S^*) < \hat{I}((1S_0))$. Also, since S^* is derived from S_0 by replacing a single loop that has iteration count n with another loop that has the same iteration count, it is easily verified that S^* is of the form $S^* = (1S_1)$. Clearly, if $\hat{I}((1S_1)) \neq 0$, we can repeat the above process to obtain a valid, fully reduced schedule $(1S_2)$ such that $max\_tokens(e_2, (1S_2)) \leq max\_tokens(e_2, (1S_1))$ for all $e \in E$, and $\hat{I}((1S_2)) < \hat{I}((1S_1))$. Continuing in this manner, we obtain a sequence of valid, fully reduced schedules $((1S_0), (1S_1), (1S_2), (1S_3), \ldots)$ such that for each S_i in the sequence with $i > 0$, $max\_tokens(e, (1S_i)) = max\_tokens(e, S_i) \leq max\_tokens(e, S)$ for all $e \in E$, and $\hat{I}((1S_i)) < \hat{I}((1S_{i-1}))$. Since $\hat{I}((1S_0))$ is finite, we cannot go on generating S_i's indefinitely — eventually, we will arrive at an S_n, $n \geq 0$, such that $\hat{I}((1S_n)) = 0$. Thus, all schedule loops in S_n satisfy the R-condition, and S_n satisfies the R-condition, and we have that S_n is an R-schedule.

From Fact 7 and from the observation that the factoring transformation does not affect the lexical ordering, it is easily verified that none of the transformations applied in deriving S_n from S change the lexical ordering. Thus, $lexorder(S_n) = lexorder(S)$. *Q.E.D.*

3 Optimally Reparenthesizing a Single Appearance Schedule

In [14], a dynamic programming algorithm was 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 an **order-optimal** schedule — 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. Given a single appearance schedule S, DPPO computes a single appearance schedule that minimizes the buffer memory requirement over all schedules in the set $\{S' \mid (lexorder(S') = lexorder(S))\}$. Thus,

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 is zero.

DPPO can be used as a post-optimization to any scheduling technique for delayless, acyclic SDF graphs. In this section, we elaborate on this technique and present an efficient extension to handle delays and general SDF graphs.

Suppose that G is a connected, consistent, delayless SDF graph, S is valid single appearance schedule for G, $lexorder(S) = (A_1, A_2, \ldots, A_n)$, and S_{oo} is an order optimal schedule for $(G, lexorder(S))$. Assuming that G contains at least two actors, we know from Theorem 1 that there is a valid schedule of the form $S_R = (i_L B_L)(i_R B_R)$ such that $buffer\_memory(S_R) = buffer\_memory(S_{oo})$ and for some $p \in \{1, 2, \ldots, (n-1)\}$, $lexorder(B_L) = (A_1, A_2 \ldots, A_p)$ and $lexorder(B_R) = (A_{p+1}, A_{p+2}, \ldots, A_n)$. Furthermore, from the order-optimality of S_{oo}, clearly, $(i_L B_L)$ and $(i_R B_R)$ must also be order optimal.

From this observation, we can efficiently compute an order-optimal schedule for G if we are given an order optimal schedule $S_{a,b}$ for the subgraph corresponding to each proper subsequence $A_a, A_{a+1}, \ldots, A_b$ of $lexorder(S)$ such that (1). $(b-a) \le (n-2)$ *and* (2). $a = 1$ or $b = n$. Given these schedules, an order-optimal schedule for G can be derived from a value of x, $1 \le x < n$ that minimizes

$$buffer\_memory(S_{1,x}) + buffer\_memory(S_{x+1,n}) + \sum_{e \in E_s} TNSE(e)\text{, where}$$

$$E_s = \{e | (src(e) \in \{A_1, A_2, \ldots, A_x\} \textbf{ and } snk(e) \in \{A_{x+1}, A_{x+2}, \ldots, A_n\})\}$$ is the set of edges that "cross the split" if the schedule parenthesization is split between A_x and A_{x+1}.

DPPO is based on repeatedly applying this idea in a bottom-up fashion to the given lexical ordering $lexorder(S)$. First, all two actor subsequences (A_1, A_2), (A_2, A_3), …, (A_{n-1}, A_n) are examined and the minimum buffer memory requirements for the edges contained in each subsequence is recorded. This information is then used to determine an optimal parenthesization split and the minimum buffer memory requirement for each three actor subsequence (A_i, A_{i+1}, A_{i+2}); the minimum requirements for the two- and three-actor subsequences are used to determine the optimal split and minimum buffer memory requirement for each four actor subsequence; and so on, until an optimal split is derived for the original n-actor sequence

$lexorder(S)$. An order-optimal schedule can easily be constructed from a recursive, top-down traversal of the optimal splits [14].

In the rth iteration of this bottom up approach, we have available the minimum buffer memory requirement $b[p, q]$ for each subsequence $(A_p, A_{p+1}, \ldots, A_q)$ that has less than or equal to r members. To compute the minimum buffer memory requirement $b[i, j]$ associated with an $r+1$-actor subchain $(A_i, A_{i+1}, \ldots, A_j)$, we determine a value of $k \in \{i, i+1, \ldots, j-1\}$ that minimizes

$$b[i, k] + b[k+1, j] + c_{i,j}[k] , \tag{5}$$

where $b[x, x] = 0$ for all x and $c_{i,j}[k]$, the memory cost at the split if we split the subsequence between A_k and A_{k+1} is given by (see Fact 6).

$$c_{i,j}[k] = \frac{\sum_{e \in E_s} TNSE(e)}{gcd(\mathbf{q}_G(A_x) \mid (i \le x \le j))} , \tag{6}$$

where

$$E_s = \{e | \left(src(e) \in \{A_i, A_{i+1}, \ldots, A_k\} \textbf{ and } snk(e) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\} \right)\} \tag{7}$$

is the set of edges that cross the split.

This technique can easily be extended to handle graphs that are not necessarily delayless, although a few additional considerations arise. We refer to our extension as **Generalized DPPO (GDPPO)**. First, if delays are present, then Fact 1 does not apply, and $lexorder(S)$, the lexical ordering of the input schedule, is not necessarily a topological sort. As a consequence, generally not all parenthesizations of the input schedule will be valid. For example, suppose that we are given the valid schedule $S = (6A)(5(2C)(3B))$ for Figure 3. Then $lexorder(S) = (A, C, B)$ clearly is not a topological sort, and it is easily verified that the schedule that corresponds to splitting the outermost parenthesization between C and B —

$(2(3A)(5C))(15B)$ — is not a valid schedule since there is not sufficient delay on the edge (B, C) to fire 10 invocations of C before a single invocation of B.

Thus, we see that when delays are present, the set E_s defined in (7) no longer generally gives all of the edges that cross the parenthesization split. We must also examine the set of *back edges*

$$E_b = \{e | (snk(e) \in \{A_i, A_{i+1}, \ldots, A_k\} \text{ and } src(e) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\})\} . \quad (8)$$

Each $e \in E_b$ must satisfy

$$delay(e) \ge \frac{TNSE(e)}{gcd(\mathbf{q}_G(A_x) \mid (i \le x \le j))} , \quad (9)$$

otherwise the given parenthesization split will give a schedule that is not valid. To take into account any nonzero delays on members in E_s, and the memory cost of each of the back edges, the cost expression of (5) for the given split gets replaced with

$$b[i, k] + b[k+1, j] + \frac{\sum_{e \in E_s} TNSE(e)}{gcd(\mathbf{q}_G(A_x) \mid (i \le x \le j))} + \sum_{e \in E_s} delay(e) + \sum_{e \in E_b} delay(e) . \quad (10)$$

Expression (10) gives the cost of spliting the subsequence $(A_i, A_{i+1}, \ldots, A_j)$ between A_k and A_{k+1} assuming that the subsequence $(A_i, A_{i+1}, \ldots, A_k)$ precedes $(A_{k+1}, A_{k+2}, \ldots, A_j)$ in the lexical order of the schedule that will be implemented. However, if (9) is satisfied *for all* "forward edges" $e \in E_s$, it may be advantageous to interchange the lexical

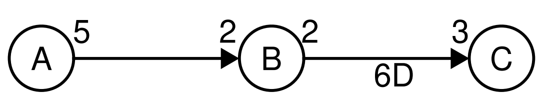

Figure 3. An SDF graph used to illustrate GDPPO applied to SDF graphs that have nonzero delay on one or more edges. Here $\mathbf{q}(A, B, C) = (6, 15, 10)$.

order of $(A_i, A_{i+1}, \ldots, A_k)$ and $(A_{k+1}, A_{k+2}, \ldots, A_j)$. Such a reversal will be advantageous whenever the *reverse split cost* defined by

$$b[i,k] + b[k+1,j] + \frac{\sum_{e \in E_b} TNSE(e)}{gcd(\mathbf{q}_G(A_x) \mid (i \le x \le j))} + \sum_{e \in E_b} delay(e) + \sum_{e \in E_s} delay(e) \qquad (11)$$

is less than the *forward split cost* computed from (10).

The possibility for reverse splits introduces a fundamental difference between GDPPO and DPPO: if one or more reverse splits are found to be advantageous, then GDPPO does not preserve the lexical ordering of the original schedule. This is not a problem since in such cases the result computed by GDPPO will necessarily have a buffer memory requirement that is less than that of an order-optimal schedule for $lexorder(S)$. On the contrary, it suggests that GDPPO may be applied multiple times in succession to yield more benefit than a single application — that is, GDPPO can in general be applied iteratively, where the iterative application terminates when the schedule produced by GDPPO produces no improvement over the schedule computed in the previous iteration.

Figure 4 shows an example where multiple applications of GDPPO is beneficial. Here $\mathbf{q}(A, B, C) = (2, 1, 2)$, and the initial schedule is $S = (2A)B(2C)$, so the initial lexical ordering is (A, B, C). Upon application of GDPPO, the minimum cost for the subsequence

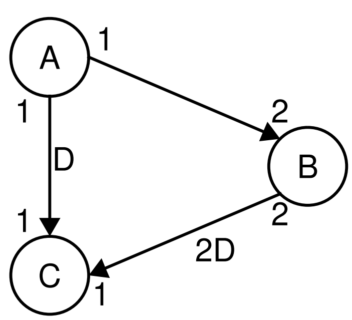

Figure 4. An illustration of iterative application of GDPPO.

(A, B) is found to be 2, and the minimum cost for the subsequence (B, C) is found to occur with a reverse split that has a cost of 2. The minimum cost for the "top-level" subsequence (A, B, C) is taken as the minimum cost over the cost if the parenthesization is split between A and B, which is equal to $0 + 2 + 5 + 0 = 7$ from (11), and the minimum cost if the split occurs between B and C, which is $2 + 0 + 7 + 0 = 9$. Thus, the former split is taken, and the result of applying GDPPO *once* to S is the schedule $S_1 = (2A)\ (1\ (2C)\ (1B))$, which has a buffer memory requirement of 7, and a lexical ordering that is different from that of S.

Since $lexorder(S_1) \neq lexorder(S)$, it is conceivable that applying GDPPO to S_1 can further reduce the buffer memory requirement. Applying GDPPO to S_1, we generate a minimum cost of 1 — which corresponds to another reverse split — for (A, C), and we generate a minimum (reverse split) cost of 2 for (C, B). Thus, we see that splitting (A, C, B) between A and C gives a cost of $0 + 2 + 5 + 0 = 7$, while splitting between C and B gives a cost of $1 + 0 + 2 + 2 = 5$. The result of GDPPO is thus the schedule $S_2 = (2CA)B$, and a buffer memory requirement of 5. It is easily verified that application of GDPPO to S_2 yields no further improvement, and thus iterative application of GDPPO terminates after three iterations.

Although the iterative application of GDPPO is conceptually interesting, we have found that for all of the practical SDF graphs that we have applied it to, termination occurred after only 2 iterations, which means that no further improvement was ever generated by a second application of GDPPO. This suggests that when compile-time efficiency is a significant issue, it may be preferable to bypass iterative application of GDPPO, and immediately accept the schedule produced by the first application.

Our extension of GDPPO can be implemented efficiently by updating forward and reverse costs incrementally. If we are examining the splits of the subsequence $(A_i, A_{i+1}, \ldots, A_j)$, and we have computed the forward and reverse split costs F_k and R_k associated with the split between A_k and A_{k+1}, $i \le k < (j-1)$, then the splits costs F_{k+1} and R_{k+1} associated with the split between A_{k+1} and A_{k+2} can easily be derived by examining the output and input edges of A_{k+1}. To ensure that we ignore reverse splits (forward splits) that fail to satisfy (9) for all $e \in E_s$

$(e \in E_b)$ a cost of $M \equiv \left(1 + \sum_{e \in E} (TNSE(e) + delay(e)) \right)$ is added to the reverse (forward) split cost for any input edge (output edge) e of A_{k+1} whose source (sink) is a member of $(A_{k+2}, A_{k+3}, \ldots, A_j)$, and that does not satisfy (9). Similarly, for each output (input) edge e of A_{k+1} whose sink (source) is contained in $(A_i, A_{i+1}, \ldots, A_k)$, and that does not satisfy (9), M is subtracted from R_{k+1} (F_{k+1}) since such an edge no longer prevents the split from being valid. Choosing M so large has the effect of "invalidating" any cost C_M that has M added to it (without a corresponding subtraction) since any minimal valid schedule has a buffer memory requirement less than M, and thus, any valid split will be chosen over a split that has cost C_M.

If forward and reverse costs are updated in this incremental fashion, then GDPPO attains a time complexity of $O\left(n_v^3\right)$ where n_v is the number of actors, if we can assume that the number of input and output edges of each actor is always bounded by some constant α. In the absence of such a bound, GDPPO has time complexity that is $O\left(n_e n_v^3\right)$, where n_e is the number of edges in the input graph.

GDPPO gives a post-optimization that can be appended to 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 discussion of these heuristics, we first develop some fundamental bounds on the buffer memory requirement of a single appearance schedule.

4 Bounds 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 Figure 5(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 Figure 5(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 Figure 5(a), we have that

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

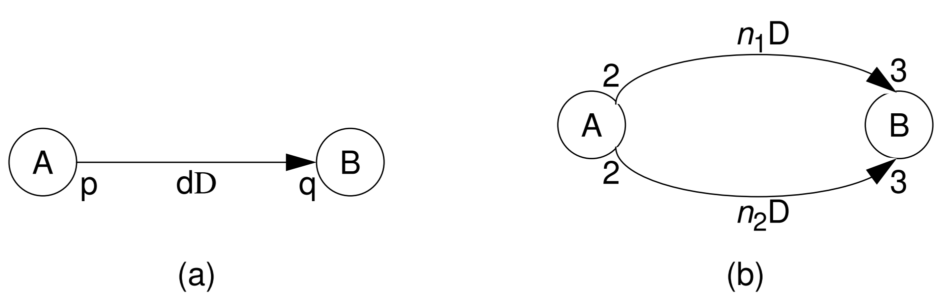

Figure 5. Examples used to develop the buffer memory lower bound.

$$\left(d \geq \frac{pq}{g}\right) \Rightarrow (max\_tokens((A, B), S) \geq d) . \tag{12}$$

Furthermore, if (A, B) is an edge in a general SDF graph, we know from Fact 2 that the projection of a valid schedule S onto $\{A, B\}$, which is a valid schedule for $subgraph(\{A, B\})$, always satisfies

$$max\_tokens((A, B), projection(S, \{A, B\})) = max\_tokens((A, B), S) . \tag{13}$$

It follows that the lower bound defined by (12) holds whenever (A, B) is an edge in a consistent SDF graph 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 Figure 1, we see that $BMLB((A, B)) = 3$, and $BMLB((B, C)) = 3$. Thus, to implement any single appearance schedule for this graph, at least three memory words will be required to implement the edge (A, B), and at least three words will be required for (B, C). Furthermore, a valid single appearance schedule for Figure 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 Figure 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 Figure 1.

In contrast, the SDF graph shown in Figure 6 has a BMLB schedule. This graph results from simply interchanging the production and consumption parameters of edge (B, C) in Figure 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 9: 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 $(\mathbf{q}_G(B)B)(\mathbf{q}_G(A)A)$ 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 $1 \leq i \leq n$.

For example, in Figure 5(b), let e_1 denote the upper edge, and let e_2 denote the lower edge. Then $\eta(e_1) = \eta(e_2) = 6$, and $(2B)(3A)$ is a BMLB schedule if $delay(e_1), delay(e_2) \geq 6$. Similarly, if $delay(e_1), delay(e_2) < 6$, then it is easily verified that $(3A)(2B)$ is a BMLB schedule. However, if $delay(e_1) < 6$ and $delay(e_2) \geq 6$, then $(3A)(2B)$ is optimal, but is not a BMLB schedule since in this case $max\_tokens(e_2, (3A)(2B)) = (delay(e_2) + 6)$, while $BMLB(e_2) = delay(e_2)$.

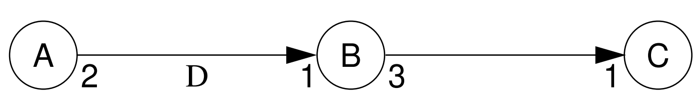

Figure 6. An SDF graph that has a BMLB schedule.

Fact 10: If $G = (V, E)$ is a connected, consistent, acyclic SDF graph, $delay(e) < \eta(e)$ for all $e \in E$, and S is a BMLB schedule for the delayless version of G, then 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 $max\_tokens(e, S, G) = max\_tokens(e, S, G') + delay(e)$ for all $e \in E$. It follows from Definition 1 that S is BMLB schedule for G. *Q.E.D.*

Fact 11: If G is a connected, consistent SDF graph and e is an edge in G, then

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

Proof: From the balance equations (1),

$$\frac{TNSE(e, G)}{\rho_G(src(e), snk(e))} = \frac{\mathbf{q}_G(src(e))\, prod(e)}{gcd\left(\{\mathbf{q}_G(src(e)), \mathbf{q}_G(snk(e))\}\right)}$$

$$= \frac{\mathbf{q}_G(src(e))\, prod(e)}{gcd\left(\{\mathbf{q}_G(src(e)), \mathbf{q}_G(src(e))\,(prod(e) / cons(e))\}\right)}.$$

Multiplying the numerator and denominator of this last quotient by $cons(e)$, and recalling that $gcd(ka, kb) = k\,gcd(a, b)$, we obtain the desired result. *Q.E.D.*

We conclude this section by defining an obvious, efficiently computable upper bound for single appearance schedules that have unit blocking factor. Clearly, if $G = (V, E)$ is a connected, consistent SDF graph, and S is a unit blocking factor single appearance schedule for G, we have $buffer\_memory(S) \le \sum_{e \in E} (TNSE(e) + delay(e))$. We refer the RHS of this inequality as the **buffer memory upper bound (BMUB)** for G.

In Figure 6, $\mathbf{q}(A, B, C) = (1, 2, 6)$, and the BMUB for this graph is 9.

5 PGAN for Acyclic Graphs

The original *Pairwise Grouping of Adjacent Nodes (PGAN)* technique was developed in [3]. In this technique, a cluster hierarchy is constructed by clustering exactly two adjacent vertices at each step. At each clusterization step, a pair of adjacent actors is chosen that maximizes $\rho(\{A, B\})$, the repetition count of the adjacent pair, over all clusterable adjacent pairs $\{A, B\}$. Recall from Section 2 that $\rho(Z)$ can be viewed as the number of times a minimal periodic schedule for the subset of actors Z is invoked in the given SDF graph, and thus, we see that the PGAN technique repeatedly clusters adjacent pairs whose associated subgraphs are invoked most frequently in a valid schedule.

To check whether or not an adjacent pair is clusterable, PGAN maintains the cluster hierarchy on a data structure called the *acyclic precedence graph (APG)*. Each vertex of the APG corresponds to an actor invocation, and there is an edge directed from the vertex corresponding to invocation x to the vertex corresponding to invocation y if and only if at least one token produced by x is consumed by y in a valid schedule. See [11] for details on the derivation of the APG that corresponds to an SDF graph.

The PGAN technique verifies whether or not an adjacent pair is clusterable by checking whether or not its consolidation introduces a cycle in the APG. It is shown that this check can be performed quickly by applying a *reachability matrix*, which indicates for any two APG vertices x and y,whether or not there is a path from x to y.

Unfortunately, the cost to compute and store the reachability matrix can be prohibitively high for multirate applications that involve large changes in sample rate. Since the number of vertices in the APG of an SDF graph (V, E) is $J \times \sum_{X \in V} \mathbf{q}(X)$, where J is the desired blocking factor, and the number of entries in the reachability matrix is quadratic in the number of APG vertices, it is easily seen that the time and space required to maintain the APG can grow exponentially with the number of actors in the given SDF graph. Although this is not a problem for the large class of practical SDF graphs for which $\sum_{X \in V} \mathbf{q}(X)$ is not much larger than the number of elements in V, practical examples can easily be constructed where the technique consumes enormous amounts of resources relative to the size of the input SDF graph. For example, for the 6-vertex SDF representation of a multi-stage sample-rate conversion system between a compact disc player and a digital audio tape player discussed in [14], $\sum_{X \in V} \mathbf{q}(X) > 600$, which means that over $360,000$ units of storage are required to implement the reachability matrix for this 6-actor SDF graph.

Since a large proportion of DSP applications that are amenable to the SDF model can be represented as acyclic SDF graphs, we propose a simple adaptation of PGAN to acyclic graphs that maintains the cluster hierarchy and reachability matrix directly on the input SDF graph rather than on the APG, and thus allows us to efficiently exploit the advantages of the bottom-up clustering approach of the original PGAN technique. We refer to this adaptation of PGAN as **Acyclic PGAN (APGAN)**. APGAN is exactly the original PGAN technique specified in [3] with the exception that the input SDF graph is assumed to be acyclic, and the cluster hierarchy and reachability matrix are maintained for the input SDF graph rather than for the APG.

In an acyclic SDF graph G, it is easily verified that a subset Z of actors is not clusterable only if $\mathit{cluster}(Z, G, \Omega)$ contains a cycle — that is, only if Z introduces a cycle. This condition is easily checked given a reachability matrix for G by examining each successor of a member of Z: $\mathit{cluster}(Z, G, \Omega)$ contains a cycle if and only if there is an $X \notin Z$ such that X is a successor of some member of Z, *and* there is a path from X to some member of Z.

Since the existence of a cycle in $\mathit{cluster}(Z, G, \Omega)$ is only a necessary — but not sufficient — condition for Z not to be clusterable, the clusterability test that we apply in our APGAN is not *exact*; it must be viewed as a conservative test. It is even inexact if we restrict ourselves to single appearance schedules. That is, it is possible for $\mathit{cluster}(Z, G, \Omega)$ to contain a cycle, and still have a valid single appearance schedule. A simple example is shown in Figure 7. Here, the

BMLB schedule $C(2AB)$ results if we first cluster $\{A, B\}$. However in APGAN, clustering $\{A, B\}$ is not permitted since the resulting graph contains a cycle. Instead, APGAN generates the schedule $(2A)C(2B)$ or the schedule $C(2A)(2B)$, neither of which is a BMLB schedule. Thus, in this example, we see that our inexact clusterization test prevents us from obtaining an optimal schedule.

In exchange for some degree of suboptimality in certain examples, 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.

Figure 8 illustrates the operation of APGAN. Figure 8(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.

Figure 8(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 $\{\Omega_1, C\}$

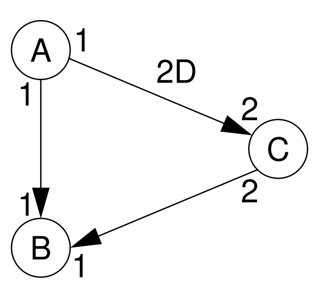

Figure 7. An example of how a clusterization operation that introduces a cycle can lead to a BMLB schedule. Here $\mathbf{q}(A, B, C) = (2, 2, 1)$.

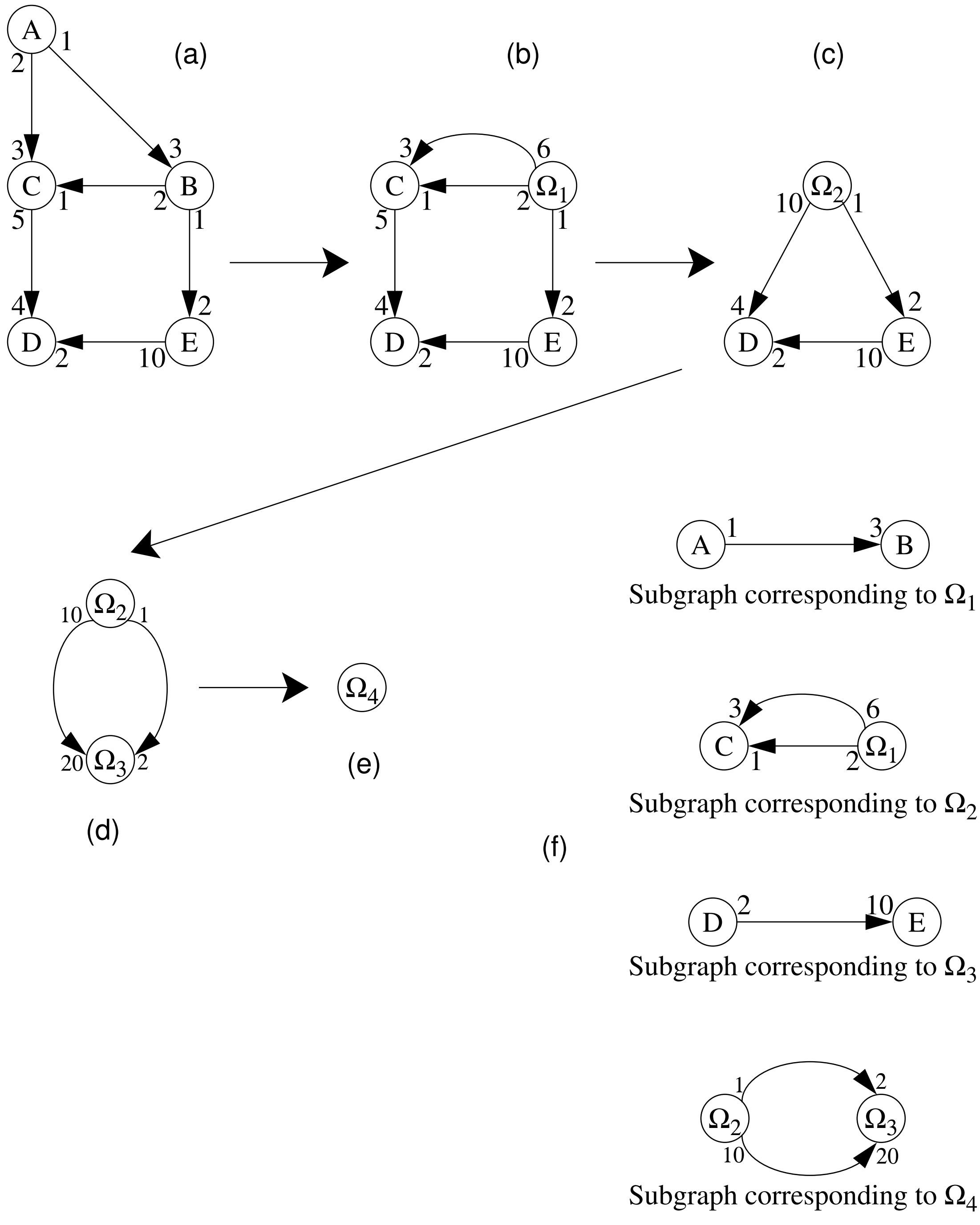

Figure 8. An illustration of APGAN.

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. Figure 8(c) shows the resulting graph.

In Figure 8(c), we have $\mathbf{q}(\Omega_2, D, E) = (2, 5, 1)$, and thus all three adjacent pairs have $\rho = 1$. 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. Figure 8(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 Figure 8(e).

Figures 8(b-e) specify the sequence of clusterizations performed by APGAN when applied to the graph of Figure 8(a). A more compact representation of this sequence is shown in Figure 8(f). A valid single appearance schedule for Figure 8(a) can easily be constructed by recursively traversing the hierarchy induced by this sequence. 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 9, 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 $(\mathbf{q}_{G_i}(Y_i) Y_i)(\mathbf{q}_{G_i}(X_i) X_i)$. In Figure 8, 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 $(3A)\,B\,(2C)$. 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 Figure 8(a).

From S_p and Figure 8(a) it is easily verified that $\mathit{buffer\_memory}\,(S_p)$ and

$\left(\sum_{e \in E} \mathit{BMLB}\,(e)\right)$, where E is the set of edges in Figure 8(a), are identically equal to 43, and

thus in the execution of APGAN illustrated in Figure 8, 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 it does not in general, result in a unique schedule for a given SDF graph. 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. For example, one tie-breaking protocol that can be used when actors are labelled alphabetically, as in Figure 8, 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 Figure 8, then $\{B, C\}$ is chosen.

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. As a consequence, all delay-free graphs, such as that shown in Figure 8(a), for which BMLB schedules exist are handled optimally by any APGAN instance. For example, even if in a certain APGAN instance, $\{B, C\}$ is clustered instead of $\{A, B\}$ in the first clusterization step of Figure 8, we are still guaranteed that final result achieved by that APGAN instance will be a BMLB schedule. To demonstrate the relevance of this optimality result, in Section 9 we will give practical applications to which the result applies. Also, we will present experimental data that suggests that our implementation of an APGAN instance frequently produces excellent results even for applications that do not have BMLB schedules, and we show that it has exhibited encouraging performance on a large collection of complex randomly-generated SDF graphs.

The following fact, which is easily understood from our discussion of the example in Figure 8, is fundamental to developing our result on the optimality of APGAN instances.

Fact 12: Suppose that 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 $buffer\_memory(S) = \sum_{e' \notin E_\Omega} BMLB(e')$, where E_Ω is the set of edges that are contained the subgraphs corresponding to the hierarchical actors $\{\Omega_i\}$ instantiated by P.

For the example of Figure 8, E_Ω is the set of six edges that appear in Figure 8(f). It is easily seen that the BMLB values for these edges are, from top to bottom, 3, 6, 2, 10, 2, and 20. Thus, Fact 12 states that the schedule obtained from the sequence of clusterizations shown in Figure 8 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 10 and 12, the desired result can be derived easily.

If an efficient data structure, such as a heap, is used to maintain the list of pairwise clustering candidates, then it can be shown that APGAN instances exist with running times that are $O\left(V^2 E\right)$. The details are beyond the scope of this paper.

6 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 Figure 8(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 13: 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 $G_c \equiv cluster(\{X, Y\}, G)$, $BMLB(e') = BMLB(e)$, 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 Figure 6, $BMLB((A, B)) = 3$, $BMLB((B, C)) = 3$, and $\mathbf{q}(A, B, C) = (1, 2, 6)$. Figures 9(a) and 9(b) respectively show $cluster(\{A, B\}, G, \Omega)$ and $cluster(\{B, C\}, G, \Omega)$, where G denotes the graph of Figure 6. In Figure 9(a), 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 Figure 9(b), we see that if $e = (A, B)$, then $e' = (A, \Omega)$, and $BMLB(e') = BMLB(e) = 3$. These observations are consistent with Theorem 2 since $\rho_G(\{A, B\}) = 1$ divides $\rho_G(\{B, C\}) = 2$, and thus $\{B, C\}$ satisfies the proper clustering condition, while $\rho_G(\{C, B\}) = 2$ does not divide $\rho_G(\{A, B\}) = 1$, and thus $\{A, B\}$ does not satisfy the proper clustering condition.

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

Figure 9. An example used to illustrate Theorem 2.

from Fact 3(a) that $\rho_{G_c}(\{src(e), snk(e)\}) = gcd\left(\{\mathbf{q}_G(X), \mathbf{q}_G(Y), \mathbf{q}_G(snk(e))\}\right)$.

Thus, since $\{X, Y\}$ satisfies the proper clustering condition, it follows from Fact 13 that $\rho_{G_c}(\{src(e), snk(e)\}) = \rho_G(\{src(e'), snk(e')\})$. From Facts 4 and 11, 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

$$\rho_G(\{Z, P\}) \text{ does not divide } \rho_G(\{X, Y\}) . \tag{14}$$

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 3(a),

$\rho_{G_c}(\{src(e), snk(e)\}) = gcd\left(\{\mathbf{q}_G(X), \mathbf{q}_G(Y), \mathbf{q}_G(snk(e))\}\right)$, and thus from (14) and Fact 13, it follows that $\rho_{G_c}(\{src(e), snk(e)\}) < \rho_G(\{src(e'), snk(e')\})$. From Facts 4 and 11, 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, 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 of X, such that the repetition count of $\{V, Y\}$ is divisible by the repetition count of $\{X, Y\}$. One interesting consequence of this lemma is that whenever a BMLB schedule exists, the repetition count of an adjacent pair that introduces a cycle cannot exceed the repetition counts of all adjacent pairs that do not introduce cycles. An example is shown in Figure 10.

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 $(e_1, e_2, \ldots, e_n)$, $n > 2$, 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 $snk(e)$.

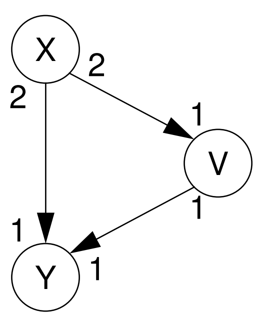

Figure 10. An illustration of Lemma 1. Here, the repetitions vector is given by $\mathbf{q}(V, X, Y) = (2, 1, 2)$, and $X(2VY)$ is a BMLB schedule. Clearly, $\{X, Y\}$ introduces a cycle. Thus, Lemma 1 guarantees that $\rho(\{X, Y\})$ divides $\rho(\{V, Y\})$, and this is easily verified from $\mathbf{q}$: $\rho(\{X, Y\}) = gcd(\{1, 2\}) = 1$, and $\rho(\{V, Y\}) = gcd(\{2, 2\}) = 2$.

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 B_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 schedule loop may be equal to (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 $(i_1B_1)(i_2B_2)$. Similarly, let I' be the product of all schedule loops contained in (i_2B_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 11 that $\rho_G(\{src(e), snk(e)\}) = I$, and $\rho_G\left(\{src(e_n), snk(e)\}\right) = II'$. Thus, $\rho_G(\{src(e), snk(e)\})$ divides $\rho_G\left(\{src(e_n), snk(e)\}\right)$. 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 $\rho\left(\{W_1, Y\}\right)$, 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 $\rho\left(\{W_1, Y\}\right)$ divides $\rho\left(\{W_2, Y\}\right)$, 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 $\left(\{W_2, Y\}\right)$ introduces a cycle, then from Lemma 1, we have $\left(\{W_3, Y\}\right)$ such that $\rho\left(\{W_2, Y\}\right)$ divides $\rho\left(\{W_3, Y\}\right)$, 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_is are distinct and we are assuming a finite graph, we cannot continue generating W_is indefinitely. Thus, eventually, we will arrive at a W_n such that $\left(\{W_n, Y\}\right)$ does not introduce a cycle. Furthermore, by our construction, $\rho(\{X, Y\})$ divides $\rho\left(\{W_1, Y\}\right)$, and for $i \in \{1, 2, \ldots, (n-1)\}$, $\rho\left(\{W_i, Y\}\right)$ divides $\rho\left(\{W_{i+1}, Y\}\right)$. It follows that $\rho(\{X, Y\})$ divides $\rho\left(\{W_n, Y\}\right)$. *Q.E.D.*

As a consequence of Corollary 1, we can be sure that given an APGAN candidate $\{X, Y\}$ in an SDF graph that has a BMLB schedule, no other adjacent pair has a higher repetition count.

As an example consider Figure 11, 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 below guarantees that under the additional assumption that Figure 11(a) has a BMLB schedule, $\rho(\{B, C\})$ is guaranteed not to exceed $\rho(\{A, B\})$.

Figure 11(b) shows a case where this additional assumption is violated. Here, $\mathbf{q}(A, B, C, D) = (2, 4, 8, 1)$. It is easily seen that 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, Figure 11(b) cannot have a BMLB schedule. It is also easily verified that among the three adjacent pairs in Figure 11(b) that do not introduce cycles, $\{A, B\}$ is the only APGAN candidate, and $\rho(\{B, C\}) = 4$, while $\rho(\{A, B\}) = 2$. Thus, the conclusion of Theorem 3 does not generally hold if we relax the assumption that the graph in question has a BMLB schedule.

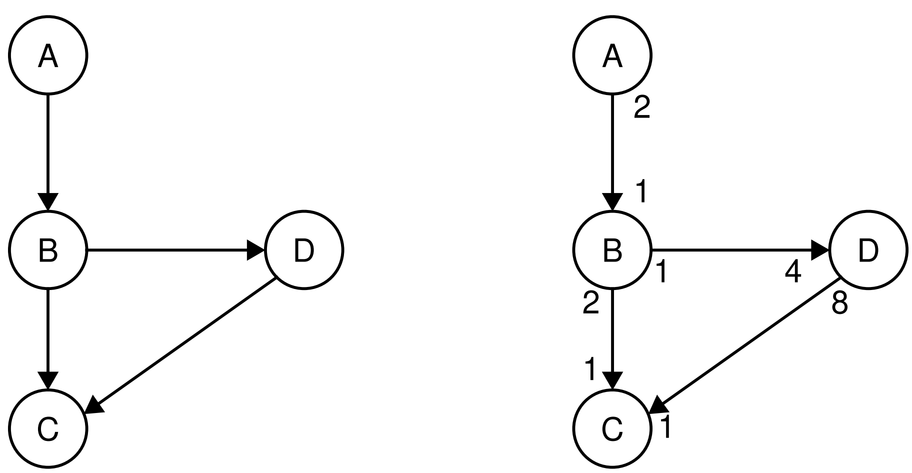

Figure 11. Examples used to illustrate Theorem 3.

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, $\rho(p) \geq \rho(p')$.

Proof: (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 $C \equiv \mathit{subgraph}(R)$ is connected, and $X, Y, Z \in R$. Then

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

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

Proof: Suppose that $\left(gcd\left(\{\mathbf{q}_C(Y), \mathbf{q}_C(Z)\} \right) \right) / \left(gcd\left(\{\mathbf{q}_C(X), \mathbf{q}_C(Y)\} \right) \right) = k$, for some positive integer k. Then, from Fact 3(b),

$$\left(gcd\left(\{\mathbf{q}_G(Y), \mathbf{q}_G(Z)\} \right) \right) / \left(gcd\left(\{\mathbf{q}_G(X), \mathbf{q}_G(Y)\} \right) \right)$$

$$= \left(gcd\left(\{\rho_G(R)\mathbf{q}_C(Y), \rho_G(R)\mathbf{q}_C(Z)\} \right) \right) / \left(gcd\left(\{\rho_G(R)\mathbf{q}_C(X), \rho_G(R)\mathbf{q}_C(Y)\} \right) \right)$$

$$= \left(gcd\left(\{\mathbf{q}_C(Y), \mathbf{q}_C(Z)\} \right) \right) / \left(gcd\left(\{\mathbf{q}_C(X), \mathbf{q}_C(Y)\} \right) \right)$$

$= k$. *Q.E.D.*

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 Figure 12. Here, the repetitions vector is given by $\mathbf{q}(A, B, C) = (6, 2, 3)$, 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). $Z \notin \{X, Y\}$ is adjacent to $P \in \{X, Y\}$; (d). $\rho_G(\{X, Y\}) \geq \rho_G(\{P, Z\})$; 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 $(i_1 X)(i_2(i_3 Y)(i_4 Z))$, or it has the form $(j_1(j_2 X)(j_3 Y))(j_4 Z)$.

If $S_R = (i_1 X)(i_2(i_3 Y)(i_4 Z))$, then $max\_tokens(e_{xy}, S_R) = TNSE(e_{xy})$, and thus from Fact 11, we have that

$$TNSE(e_{xy}) = TNSE(e_{xy}) / \rho(\{X, Y\}),$$

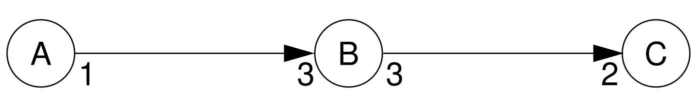

Figure 12. An illustration of Lemma 3.

which implies that $\rho(\{X, Y\}) = 1$. From Assumption (d), it follows that $\rho(\{Y, Z\}) = 1$.

Conversely, suppose that $S_R = (j_1 (j_2 X)(j_3 Y))(j_4 Z)$. Then $max\_tokens(e_{yz}, S_R) = TNSE(e_{yz})$, so from Fact 11, 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 operation 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 Figure 8(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 Figure 8(a). Similarly, for Figure 8(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 Figure 8(b).

Theorem 4: Suppose that G is a connected, consistent, delayless SDF graph; a BMLB schedule exists for G; 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 $C = subgraph(\{X, Y, Z\})$, and observe from Fact 2 that C has a BMLB schedule. From Theorem 3, $\rho_G(\{Z, P\}) \le \rho_G(\{X, Y\})$, and from Fact 3(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.*

7 The Optimality of APGAN for a Class of Graphs

In this section, we use main the results of Section 6 to show that for any acyclic SDF graph (V, E) that has a BMLB schedule, and that satisfies $delay(e) < \eta(e)$, for all $e \in E$, any APGAN instance is guaranteed to construct a BMLB schedule.

In Section 6, 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.

Consider the SDF graph in Figure 13(a) ($\mathbf{q}(A, B, C, D, E, F) = (3, 5, 10, 10, 5, 2)$). It is easily verified that $(3A)(5B(2DC)E)(2F)$ is a BMLB schedule. Also, observe that $\rho(\{A, F\}) = \rho(\{A, B\}) = \rho(\{E, F\}) = 1$, and thus, $\{A, F\}$ satisfies the proper clustering condition. Figure 13(b) shows $cluster(\{A, F\}, G, \Omega)$, where G denotes the graph of Figure 13(a). In Figure 13(b), we see that due to the path $((D, E), (E, \Omega), (\Omega, B), (B, C))$, D must

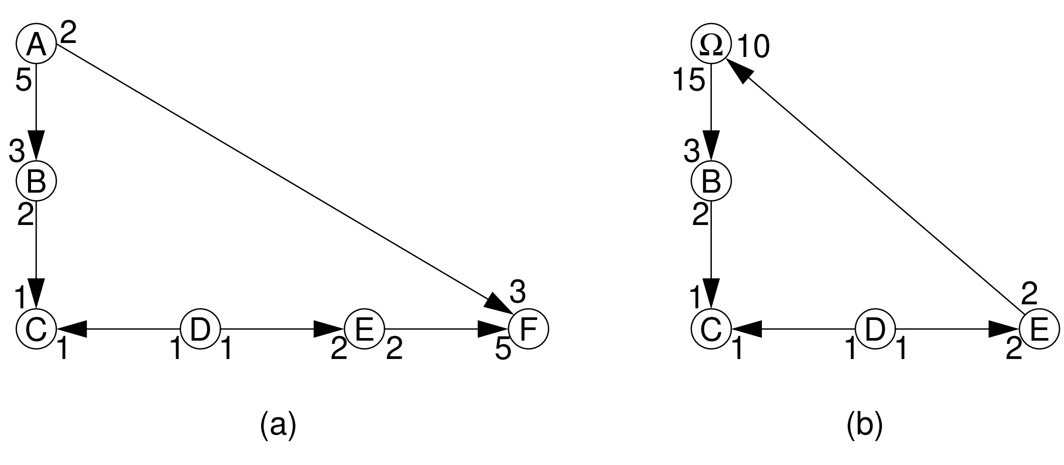

Figure 13. An example of how clustering an adjacent pair that satisfies the proper clustering condition can cancel the existence of a BMLB schedule.

fire 10 times before a single invocation of C can fire, and thus $max\_tokens((D, C), S) \geq 10 > BMLB((D, C))$, if S is a valid schedule for Figure 13(b). Thus, Figure 13(b) cannot have a BMLB schedule, and we see that even though $\{A, F\}$ satisfies the proper clustering condition in Figure 13(a), clustering this adjacent pair does not preserve the existence of a BMLB schedule.

Fortunately, the assumption that the adjacent pair being clustered has maximum repetition count is sufficient to preserve the existence of a BMLB schedule. Thus, clustering an APGAN candidate always preserves the existence of a BMLB schedule.

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 $cluster(\{X, Y\}, G)$ 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, $cluster(\{X, Y\}, G)$ contains no edges. Now suppose that the theorem holds for $|V| = 2, 3, \ldots, k$, and consider the case $|V| = (k+1)$.

Define $G_c = cluster(\{X, Y\}, G, \Omega)$, 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 2, $S_i = projection((i_1 B_1), C_i)$ is a BMLB schedule for each C_i. Let C_j denote that connected component that contains X and Y. Then, since $|C_j| \leq k$, 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 8, 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 11 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 11,

$$BMLB(e) = TNSE(e, G) \text{ for all } e \in E. \tag{15}$$

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

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

$$= \sum_{e \in E_c} TNSE(e', G) \qquad \text{(from Fact 4)}$$

$$= \sum_{e \in E_c} BMLB(e') \qquad \text{(from (15))}$$

$$= \sum_{e \in E_c} BMLB(e). \qquad \text{(from Theorems 2 and 3)}$$

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 on 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 12 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 instances.

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 returns $S_P(G)$ if the input graph is G', and returns $S_P(G_I)$ otherwise, where G_I is the input graph. Clearly P' is an APGAN instance since edge delays do not affect the repetition counts of adjacent pairs. From Lemma 4 and Fact 10, $S_{P'}(G')$ is BMLB schedule for G. But by construction,

$S_{P'}(G') = S_P(G)$. *Q.E.D.*

Figure 14 shows what can "go wrong" in trying to achieve the BMLB with APGAN when the assumption that $delay(e) < \eta(e)$ is not satisfied for all edges. In the SDF graph of (a), $\mathbf{q}(A, B, C, D) = (1, 1, 1, 1)$, and thus all adjacent pairs have the same repetition count. Thus, two possible clusterization sequences by an APGAN instance for this graph are $\{W, Y\}$ followed by $\{X, Z\}$ (shown in (b)), and $\{W, X\}$ followed by $\{Y, Z\}$ (shown in (c)). From (b) and (c), we see that the schedules resulting from these two clusterization sequences are (ignoring all one-iteration loops), respectively, $YWZX$ and $YZXW$. Here, the former schedule has a buffer memory requirement of 5 , while the latter schedule has a buffer memory requirement of 4 since the sink actor fires before the source actor for each edge that has unit delay. Thus, we see that different APGAN instances will in general produce different buffer memory requirements when applied to Figure 14(a).

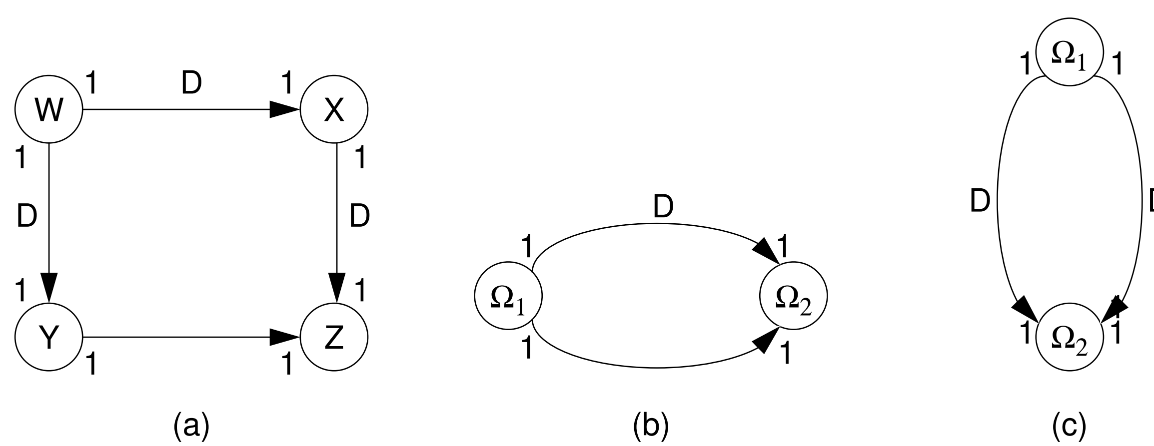

Figure 14. An example of how an APGAN instance may fail to achieve the BMLB when $delay(e) < \eta(e)$ does not hold for every edge e .

8 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 6). 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

$$|V_R| \le K \ , \ |V_L| \le K \ . \qquad (16)$$

The weight of an edge e is defined as

$$w(e) = TNSE(e) \ . \qquad (17)$$

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 in (17). Since the related problem of finding a minimum cut (not necessarily legal) into bounded sets is NP-complete [6], and the problem of finding an acyclic partition of a graph is NP-complete [6], we believe this problem to be NP-complete as well even though we have not discovered a proof. Kernighan and Lin [8] devised a heuristic procedure for computing cuts into bounded sets but they considered only undirected graphs. Methods based on network flows [5] 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.

The heuristic is to examine the set of cuts produced by taking a vertex and all of its descendants as the vertex set V_R and the set of cuts produced by taking a vertex 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 vertex 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 [8], 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 output edges of a and the total weight of all input 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 vertex v in the graph and take the best one as the minimum cut. For a pseudocode specification of the algorithm, see [14]. Since a greedy strategy is being used to move actors across, and only the boundary actors are considered, examples can be constructed where the heuristic will not give optimal cuts. 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 $(r_L S_L)(r_R S_R)$ where $r_L = gcd\left(\{\mathbf{q}(v) \mid v \in V_L\}\right)$, $r_R = gcd\left(\{\mathbf{q}(v) \mid v \in V_R\}\right)$ and S_L, S_R are schedules for G_L and G_R respectively. 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.

9 Experimental Results

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 to each. 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 9 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. For example, the column titled "APGAN < random" represents the number of random schedules that had a buffer memory requirement greater than that obtained by APGAN. The last two columns give the mean number 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 heuristic 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

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

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

.

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%	----	----
Nonuniform filterbank (1/3,2/3 splits) (4 channels)	100%	0%	100%	0%	----	----
Nonuniform filterbank (1/3,2/3 splits) (6 channels)	100%	0%	100%	0%	----	----
QMF nonuniform-tree filterbank	100%	0%	81%	7%	----	8
QMF filterbank (one-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

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 a simple structure topologically and regular rate changes, like the uniform QMF filterbanks, and RPMC performs well on graphs that have more irregular rate changes and irregular topologies. 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

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%

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.

In [14], we report on a variation of APGAN that achieves significantly better performance on random graphs than the original version, although still significantly worse performance as compared to RPMC. This variation arises from changing the "priority function" associated with an edge e from $\rho(\{src(e), snk(e)\})$ to the product

$$(TNSE(e) \times \rho(\{src(e), snk(e)\})) . \qquad (18)$$

In other words, the variation that we propose repeatedly clusters the source and sink vertices of edges that maximize the measure given by (18). Thus, an adjacent pair is given more weight if a large amount of data is transferred between it as compared to other adjacent pairs.

We have found that on the random graphs that were used to generate Table 3, this modification of APGAN outperforms two random schedules ("min(2 random)") 76.5 percent of the time, which indicates a level of performance intermediate to APGAN and RPMC. Furthermore, its performance equaled the performance of APGAN on all of the practical examples except the six channel nonuniform filter bank, where it achieved a buffer memory requirement of 696 (8% better than APGAN), and the four channel nonuniform filter bank, where it achieved 136 (0.7% better than APGAN).

Interestingly, however, the modification of APGAN corresponding to (18) does not preserve the formal properties specified by Theorem 6. This is easily seen from the example in Figure 15. Here, $\mathbf{q}(W, X, Y) = (2, 2, 1)$, and thus $\rho(\{W, X\}) = 2$ and $\rho(\{W, Y\}) = 1$, and if we let $\hat{\rho}$ denote the measure defined by (18), then $\hat{\rho}(\{W, X\}) = 2 \times 2 = 4$, while $\hat{\rho}(\{W, Y\}) = 6 \times 1 = 6$. We see then that APGAN clusters $\{W, X\}$ in its first clusterization step, which leads to the final schedule $(2WX)Y$, and a buffer memory requirement of 7, while in our variation of APGAN, $\{W, Y\}$ is clustered first, and the resulting schedule $(2W)Y(2X)$ gives a buffer memory requirement 8. It is easily verified the BMLB for this graph is 7, and thus, APGAN generates a BMLB schedule, while the variation generates a suboptimal result.

Thus, our variation of APGAN introduces a trade-off between provable optimality for a class of graphs, and average-case performance. Since we are proposing to complement a heuristic — RPMC — whose average case performance significantly outweighs that of both APGAN and its variation, it is intuitively more appealing to choose the original version of APGAN since it adds a feature that RPMC lacks — optimality for a restricted, but useful, class of graphs. For the practical examples that we examined, the variation of APGAN outperformed the original APGAN only in cases where RPMC outperformed both APGAN and the APGAN variation, and thus adopting the new version of APGAN does not improve the final result of any of these examples when a combined solution with RPMC is employed.

10 Related Work

In [1], Ade, Lauwereins, and Peperstraete develop upper bounds on the minimum buffer memory requirement for certain classes of SDF graphs. Since the bounds of Ade et al. 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 [9], Lauwereins, Wauters, Ade, and Peperstraete present a generalization of SDF called

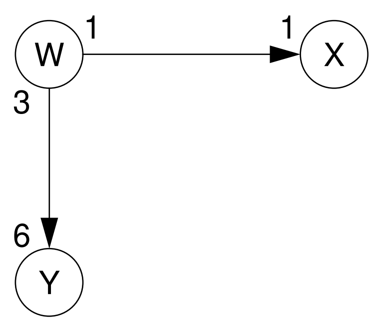

Figure 15. An example in which APGAN achieves the BMLB, but the modified version corresponding to (18) does not.

cyclo-static dataflow. A major advantage of cyclo-static dataflow is that it can eliminate large amounts of token traffic arising from the need to generate dummy tokens in corresponding (pure) SDF representations. This leads to lower memory requirements and fewer run-time operations. Although cyclostatic dataflow can reduce the amount of buffering for graphs having certain multirate actors like explicit downsamplers, it is not clear whether this model can in general be used to derive schedules that are as compact as single appearance schedules for pure SDF graphs but have lower buffering requirements than those arising from the techniques 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 [7]. Here the goal is to minimize the buffer cost without sacrificing throughput — just as the goal in this paper is to minimize buffering cost without sacrificing code compactness. Thus, the techniques of [7] address the problem of selecting a schedule that minimizes buffering cost from among the set of *rate-optimal* schedules.

11 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 discussed a generalization to handle delays and arbitrary topologies of the dynamic programming approach described in [14] for post-optimizing a single appearance schedule by reparenthesizing its lexical ordering. 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 [3]; 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.

The solutions developed in this paper have focused on acyclic SDF graphs. Single appearance schedules for general SDF graphs can be constructed efficiently by clustering the strongly connected components into a "top-level graph," constructing a single appearance schedule for the resulting (acyclic) hierarchical graph, constructing a single appearance schedule for each strongly connected component in isolation, and then replacing each hierarchical actors in the schedule for the top-level graph with the schedule for the corresponding strongly connected component [2]. Thus, the solutions presented in this paper can be exploited when scheduling general SDF graphs by applying them to the top-level graph. More thorough techniques for jointly optimizing code and data for general SDF graphs is a topic for further study.

Glossary

$\eta(e)$ — Given an SDF edge e, $\eta(e) = \dfrac{prod(e)\, cons(e)}{gcd(\{prod(e), cons(e)\})}$.

$\rho(Z)$ — Given a subset of actors Z, $\rho(Z) = gcd(\mathbf{q}(A) | A \in Z)$.

Adjacent pair — A set $\{X, Y\}$ that consists of two adjacent actors.

APGAN — Acyclic PGAN — a customization of PGAN to acyclic graphs. This is a technique for constructing single appearance schedules that repeatedly clusters adjacent pairs that have maximum repetition count over all adjacent pairs that do not introduce cycles.

Blocking factor — For each valid schedule S for a connected SDF graph, there is a positive integer k such that S invokes each actor A exactly $k\mathbf{q}(A)$ times. The constant k is the called the blocking factor of S.

BMLB — The 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.

BMUB — Given a consistent SDF graph, the BMLB is an upper bound on the buffer memory requirement over all single appearance schedules that have unit blocking factor.

$cluster(Z, G, \Omega)$

The SDF graph that results from clustering the subset of actors Z in the SDF graph G into the actor Ω. Also denoted $cluster(Z, G)$ if Ω is understood.

DPPO — Dynamic processing post optimization. A technique for computing a single appearance schedule that has minimum buffer memory requirement from among the single appearance schedules that have a given lexical ordering. The technique applies to delayless SDF graphs.

GDPPO — Generalized DPPO. A generalization of DPPO to handle delays.

Introduces a cycle

A subset of actors Z in a connected, consistent, acyclic SDF graph G introduces a cycle if $cluster(Z, G)$ contains one or more cycles.

$J(S)$ — Denotes the blocking factor of the valid schedule S.

$max\_tokens(e, S)$

Given an SDF graph G, a valid schedule S for G, and an edge e in G, we define $max\_tokens(e, S, G)$ to denote the maximum number of tokens that are queued on e during an execution of S. When G is understood, we may write $max\_tokens(e, S)$ in place of $max\_tokens(e, S, G)$.

PGAN — Pairwise grouping of adjacent nodes. A bottom-up technique for constructing looped schedules that repeatedly clusters adjacent pairs of actors that have maximum repetition count over all clusterable adjacent pairs.

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.

RPMC Recursive partitioning by minimum cuts. A top-down technique for constructing looped schedules that involves recursively computing partitions that have minimum buffering cost for all edges that cross the partitions.

$TNSE(e)$ Total number of samples exchanged on an SDF edge. Given an SDF edge e in a consistent SDF graph, $TNSE(e) = \mathbf{q}(src(e))\,prod(e)$.

References

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

[2] S. S. Bhattacharyya, *Compiling Dataflow Programs for Digital Signal Processing*, Ph. D. thesis, Memorandum No. UCB/ERL M94/52, Electronics Research Laboratory, University of California at Berkeley, July, 1994.

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

[4] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Multirate Signal Processing in Ptolemy," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Toronto, April, 1991.

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

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

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

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

[9] R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete, "Geometric Parallelism and Cyclo-

Static Dataflow in GRAPE-II," presented at *IEEE Workshop on Rapid System Prototyping*, Grenoble, June, 1994.

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

[11] E. A. Lee, *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors*, Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, May, 1986.

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

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

[14] P. K. Murthy, S. S. Bhattacharyya, and 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, December, 1994.

[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, and J. T. Buck, "Software Synthesis for DSP Using Ptolemy," invited paper in *Journal of VLSI Signal Processing*, to appear in 1994.

[17] S. Ritz, S. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, August, 1992.