Memorandum UCB/ERL M94/93, Electronics Research Lab, University of California, Berkeley, November 29, 1994

COMBINED CODE AND DATA MINIMIZATION FOR SYNCHRONOUS DATAFLOW PROGRAMS.

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

November 20, 1994

ABSTRACT

In this paper, we formally develop techniques that minimize the memory requirements of a target program when synthesizing software from dataflow descriptions of multirate signal processing algorithms. The dataflow programming model that we consider is the *synchronous dataflow* (*SDF*) model [17], which has been used heavily in DSP design environments over the past several years. We first focus on the restricted class of *well-ordered* SDF graphs. We show that while extremely efficient techniques exist for constructing minimum code size schedules for well-ordered graphs, the number of distinct minimum code size schedules increases combinatorially with the number of vertices in the input SDF graph, and these different schedules can have vastly different data memory requirements. We develop a dynamic programming algorithm that computes the schedule that minimizes the data memory requirement from among the schedules that minimize code size, and we show that the time complexity of this algorithm is cubic in the number of vertices in the given well-ordered SDF graph. We present several extensions to this dynamic programming technique to more general scheduling problems, and we present a heuristic that often computes near-optimal schedules with quadratic time complexity. We then show that finding optimal solutions for arbitrary acyclic graphs appears to be difficult and present heuristic techniques that jointly minimize code and data size requirements. We present a practical example and simulation data that demonstrate the effectiveness of these techniques.

This work is 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, Cadence, 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 Background

The synchronous dataflow (SDF) model has been used widely as a foundation for block-diagram programming of digital signal processing (DSP) systems (see, for example, [15, 16, 18, 20, 21]). In this model, as in other forms of dataflow, a program is specified by a directed graph in which the vertices, called **actors**, represent computations, and the edges represent FIFO queues that store data values, called **tokens**, as they pass between computations. We refer to the FIFO queue associated with each edge as a **buffer**.

SDF imposes the restriction that the number of tokens produced and consumed by each actor is fixed and known at compile time. Figure 1 shows an example of an SDF graph. Each edge is annotated with the number of tokens produced (consumed) by each invocation of the source (sink) actor. Given an SDF edge α, we denote the source actor of α by $source(\alpha)$ and the sink actor of α by $sink(\alpha)$. We denote the number of tokens produced onto α per each invocation of $source(\alpha)$ by $produced(\alpha)$, and similarly, we denote the number of tokens consumed from α per each invocation of $sink(\alpha)$ by $consumed(\alpha)$. Each edge in a general SDF graph also has associated with it a non-negative integer *delay.* A unit of delay represents an initial token on an edge. For clarity, in this paper, we will usually assume that the edges in an SDF graph all have zero delay; however, we will explain how to extend our main techniques to handle delays.

In this paper, we focus initially on SDF graphs that are **chain-structured**. An m-vertex directed graph is chain-structured if it has $m-1$ edges, and there are orderings $(v_1, v_2, \ldots, v_m)$ and $(\alpha_1, \alpha_2, \ldots, \alpha_{m-1})$ for the vertices and edges, respectively, such that each α_i is directed from v_i to v_{i+1}. Figure 1 is an example of a chain-structured SDF graph. The major results that we present for chain-structured SDF graphs can be extended to the somewhat more general class

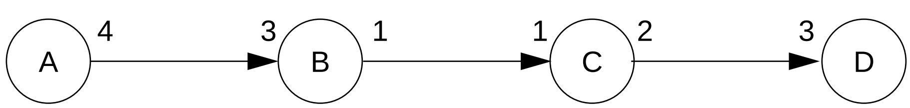

Figure 1. A chain-structured SDF graph.

of **well-ordered** graphs, but for clarity, we develop our techniques in the context of chain-structured graphs. A directed graph is well-ordered if it has only one ordering of the vertices such that for each edge α, $source(\alpha)$ occurs earlier in the ordering than $sink(\alpha)$. We will discuss the extensions of our techniques to well-ordered SDF graphs in Section 6.

A **schedule** is a sequence of actor firings. We compile a properly-constructed SDF graph by first constructing a finite schedule σ that fires each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each buffer. When such a schedule σ is repeated infinitely, we call the resulting infinite sequence of actor firings a **valid periodic schedule**, or simply a "valid schedule", and we say that σ is the **body** of this valid schedule. Corresponding to each actor in the schedule body σ, we insert 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 valid schedule.

SDF graphs for which valid schedules exist are called **consistent** graphs. Systematic techniques exist to efficiently determine whether or not a given SDF graph is consistent [17]. Also, given a consistent SDF graph, the minimum number of times that each actor must execute in the body of a valid schedule can be computed efficiently [17]. We represent these minimum numbers of firings by a 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). For Figure 1, $\mathbf{q} = \mathbf{q}(A, B, C, D) = (9, 12, 12, 8)^T$.[1]

For example, $(\infty(2ABC)DABCDBC(2ABCD)A(2BC)(2ABC)A(2BCD))$ represents a valid schedule for Figure 1. 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 translate such a term into a loop in the target code. Note 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** $S_1, S_2, \ldots, S_k$.We say that a schedule for an SDF graph is a **looped schedule** if it contains

1. We adopt the convention of indexing vectors and matrices using functional notation rather than subscripts or superscripts. Also, we denote the transpose of a vector $\mathbf{x}$ by $\mathbf{x}^T$.

zero or more schedule loops. Thus, the "looped" qualification indicates that the schedule in question may be expressed in terms of schedule loops. Given a valid looped schedule S, we refer to each iterand of the outermost schedule loop (the loop that has infinite iteration count) as an *iterand* of S.

A more compact valid schedule for figure 1 is $(\infty (3 (3A) (4B)) (12C) (8D))$. We call this schedule a **single appearance schedule** since it contains only one 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 loop overhead and other second-order effects, such as the efficiency of data transfers between actors [4].

In general, a schedule of the form $(\infty (\mathbf{q}(N_1) N_1) (\mathbf{q}(N_2) N_2) \ldots (\mathbf{q}(N_K) N_K))$ is called a **naive** single appearance schedule. For the graph in figure 1, the schedule $(\infty (9A) (12B) (12C) (8D))$ is a naive single appearance schedule.

1.1 Buffering Costs

The amount of memory required for buffering may vary greatly between different schedules. We define the **buffer memory requirement** of a schedule S, denoted $buffer\_memory(S)$, as $\sum max\_tokens(\alpha, S)$, where the sum is taken over all edges α, and $max\_tokens(\alpha, S)$ denotes the maximum number of tokens that are simultaneously queued on α during an execution of S. For example, the schedule $(\infty (9A) (12B) (12C) (8D))$ has a buffer memory requirement of $36 + 12 + 24 = 72$, and the schedule $(\infty (3 (3A) (4B)) (4 (3C) (2D)))$ has a buffer memory requirement of $12 + 12 + 6 = 30$.

In the model of buffering implied by our "buffer memory requirement" measure, each buffer is mapped to a contiguous and independent block of memory. This model is convenient and natural for code generation, and it is the model used, for example, in the SDF-based code generation environments described in [12, 19, 20]. However, perfectly valid target programs can be generated without these restrictions. For example, another model of buffering is to use a shared buffer of size $max\left(\{ \mathbf{q}(N_i) \times produced(N_i) \mid 1 \le i < K \} \right)$ which gives the maximum amount of data

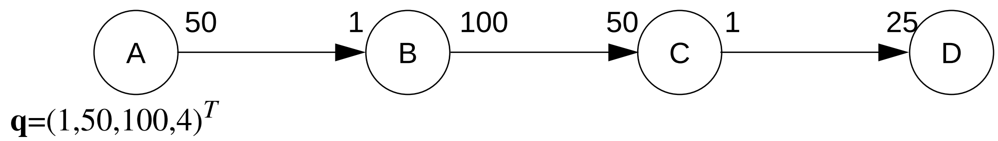

Figure 2. Example to illustrate the inefficiency of using shared buffers.

transferred on any edge in one *period* (one iteration of the outermost loop) of the naive single appearance schedule, $(\infty\,(\mathbf{q}\,(N_1)\,N_1)\;(\mathbf{q}\,(N_2)\,N_2)\;\ldots\;(\mathbf{q}\,(N_K)\,N_K))$, where K is the number of nodes in the graph. Assuming that there are no delays on the graph edges, it can be shown that via proper management of pointers, such a buffer suffices. For the example graph above, this would imply a buffering requirement of 36 since on edge AB, 36 samples are exchanged in the schedule $(\infty\,(9A)\;(12B)\;(12C)\;(8D))$, and this is the maximum over all arcs. Moreover, the implementation of this schedule using a shared buffer would be much simpler than the implementation of a more complicated nested schedule. But there are two problems with buffer-sharing that prevent its use as the model for evaluating the buffering cost of single appearance schedules. Consider the graph in Figure 2. The shared-buffer cost for the naive schedule for this graph is given by $max(\{1 \times 50, 50 \times 100, 100 \times 50, 4 \times 25\}) = 5000$. However, with a buffering model where we have a buffer on each edge, the schedule $(\infty A\,(50B\,(2C))\;(4D))$ requires total buffering of only 250 units. Of-course, we could attempt sharing buffers in this nested looped schedule as well, but the implementation of such sharing could be awkward.

Consider also the effect of having delays on the arcs. In the model where we have a buffer on every edge, having delays does not affect the ease of implementation. For example, if we introduce d delays on edge BC in the graph in Figure 2, then we merely augment the amount of buffering required on that edge by d. This is fairly straightforward to implement. On the other hand, having delays in the shared buffer model causes complications because there is often no logical place in the buffer to place the delays since the entire buffer might be written over by the time we reach the actor that consumes the delays. For instance, consider the graph in figure 3. The repetitions vector for this graph is given by $(147, 49, 28, 32, 160)^T$. Suppose that we were to use the shared-buffer implementation for the naive schedule. We find that we need a buffer of size 224.

After all of the invocations of A have been fired, the first 147 locations of the buffer are filled. Since B writes more samples than it reads, it starts writing at location 148 and writes 196 samples. When C begins execution, it starts reading from location 148 and starts writing from location 120 (120 = (148+196) mod 224). Actor C then writes 224 samples into the buffer. When D is invoked, it starts reading from location 120. Hence, if there were a delay on edge CD for instance, the logical thing to do would be to have a buffer of size 225 (meaning that D would start reading from location 119) and place the delay in location 119. However, location 119 would have been written over by A; hence, it is not a safe location. This shows that handling delays in the shared buffer model can be quite awkward, and would probably involve copying over data from a "delay" buffer of some sort. Therefore, in this paper we focus mainly on the buffering model associated with the "buffer memory requirement" measure, although, in Section 6, we present an extension of our techniques to combine the above simple model of buffer sharing with the non-shared model. The buffer sharing model will only be used whenever it is feasible to do so (whenever there are no delays, and the size of the shared buffer is lower). There are also other ways in which sharing can be done; thoroughly combining the advantages of nested loops and these other ways of sharing buffers is a topic for further study.

We note briefly that nested-schedules have a lower latency than naive single appearance schedules. The latency is defined to be the time at which the sink node fires for the first time in the schedule. In a naive schedule $(\infty\,(\mathbf{q}\,(N_1)\,N_1)\;(\mathbf{q}\,(N_2)\,N_2)\;\ldots\;(\mathbf{q}\,(N_K)\,N_K)\,)$, the latency is given by $(\mathbf{q}\,(N_1)-1)\,T+E_1+\sum_{i=2}^{K-1}\mathbf{q}\,(N_i)\,E_i$, where T is the sample period of the source actor, and E_i is the execution time of actor N_i. All these times are assumed to be in number of instruction cycles of the processor. A nested-schedule will usually have a latency less than this because if the

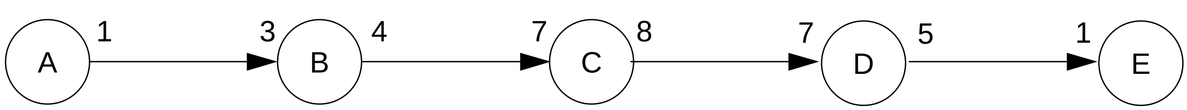

Figure 3. Example to illustrate the difficulty of using shared buffers with delays.

sink actor is part of a nested loop body, then all of the invocations of actors upstream do not have to occur before the sink actor fires for the first time. In section 4, we illustrate this by an example.

In this paper, we discuss the problem of computing a single appearance schedule that minimizes the buffer memory requirement over all valid single appearance schedules. Thus, given our model of buffer implementation, we wish to construct a software implementation that minimizes the data memory requirement over all minimum code-size implementations. As we will show in Section 2, for chain-structured SDF graphs, the number of distinct valid single appearance schedules increases combinatorially with the number of actors, and thus exhaustive evaluation is not, in a general, a feasible means to find the single appearance schedule that minimizes the buffer memory requirement. In Section 3, we show that the problem of finding a valid single appearance schedule that minimizes the buffer memory requirement for a chain-structured SDF graph is similar to the problem of most-efficiently multiplying a chain of matrices, for which a cubic-time dynamic programming algorithm exists [10]. We show that this dynamic programming technique can be adapted to our problem to give an algorithm with time complexity $O\left(m^3\right)$, where m is the number of actors in the input chain-structured SDF graph.

In Section 4, we illustrate the relevance of our dynamic programming solution through a practical example — a sample-rate conversion system to convert between the output of a compact disk player and the input of a digital audio tape player. In Section 5, we discuss an alternative solution to the problem of minimizing the buffer memory requirement over all single appearance schedules for a chain-structured SDF graph. This is a heuristic approach whose worst-case time complexity is $O\left(m^2\right)$; our experimental data suggests that this heuristic often performs quite well. In Section 6, we discuss how the dynamic programming technique of Section 4 can be applied to other problems in the construction of efficient looped schedules. Through Section 6 we are concerned primarily with chain-structured SDF graphs. In Section 7, we discuss solutions that we have developed for general acyclic SDF graphs, and present simulation data that demonstrates the efficacy of these methods. Finally, in Section 8, we discuss closely related work of other researchers.

We will use the following definitions in this paper

• Given an SDF graph G, we denote the set of actors in G by $actors(G)$, and the set of edges in G by $edges(G)$.

• By a **subgraph** of an SDF graph, we mean the SDF graph formed by any $V \subseteq actors(G)$ together with the set of edges $\{\alpha \in edges(G) \mid (source(\alpha), sink(\alpha) \in V)\}$. We denote the subgraph associated with the set of actors V by $subgraph(V, G)$.

• Given a finite set P of positive integers, we denote by $gcd(P)$ the greatest common divisor of P — the largest positive integer that divides all members of P.

• Given a finite set Z, we denote the number of elements in Z by $|Z|$.

• Given a connected, consistent SDF graph G, and a subset $V \subseteq actors(G)$, we define $q_G(V) \equiv gcd\left(\{\mathbf{q}_G(A) \mid (A \in V)\}\right)$. In [4], we discuss that $q_G(V)$ can be viewed as the number of times that a periodic schedule for G invokes the subgraph associated with V.

When discussing the complexity of algorithms, we will use the standard O, Ω and Θ notation. A function $f(x)$ is $O(g(x))$ if for sufficiently large x, $f(x)$ is bounded above by a positive real multiple of $g(x)$. Similarly, $f(x)$ is $\Omega(g(x))$ if $f(x)$ is bounded below by a positive real multiple of $g(x)$ for sufficiently large x, and $f(x)$ is $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$.

Also, we will use a number of facts that are proved in [4]. The first fact relates the repetitions vector of a connected SDF subgraph to that of an enclosing SDF graph.

Fact 1: If G is a connected, consistent SDF graph and R is a connected subgraph of G, then for each $A \in actors(R)$, $\mathbf{q}_G(A) = q_G(actors(R))\,\mathbf{q}_R(A)$.

The next fact is related to the **factoring transformation** for looped schedules that was introduced in [5]. As an example of the factoring transformation, consider the valid schedule $S_1 \equiv (\infty(3(3A)(4B))(2(6C)(4D)))$, and observe that the iteration counts of the two loops that are nested in the loop $(2\,(6C)\ (4D))$ have a common divisor of 2. Fact 2 guarantees that if we "factor" this common divisor from the iteration counts of these two inner loops into the iteration count of the enclosing loop, then the resulting schedule, $S_2 \equiv (\infty\,(3\,(3A)\ (4B))\ (4\,(3C)\ (2D)))$ is valid and that its buffer memory requirement does not exceed the buffer memory requirement of the original schedule. It is easy (although a bit tedious) to verify that S_2 is indeed a valid schedule, and we see that $\mathit{buffer\_memory}(S_2) = 12 + 12 + 6 = 30$, while $\mathit{buffer\_memory}(S_1) = 12 + 12 + 12 = 36$, and thus for this example, the factoring transformation has reduced the buffer memory requirement by 17%.

Fact 2: 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). $\mathit{buffer\_memory}(S') \le \mathit{buffer\_memory}(S)$.

The factoring transformation is closely related to the *loop fusion* transformation, which has been used for decades in compilers for procedural languages to reduce memory requirements and increase data locality [1, 22]. In compilers for procedural languages, tests for the validity of loop fusion include analysis of array subscripts to determine whether or not for each iteration n of the (lexically) second loop, this iteration depends only on iterations $1, 2, \ldots, n$ of the first loop [23]. These tests are difficult to perform comprehensively due to the complexity of exact subscript analysis [3], and due to complications such as data-dependent subscript values, conditional branches, and input/output statements. In contrast, Fact 2 gives a simple test for the validity of the factoring transformation that is applicable to a broad class of looped schedules, including all single appearance schedules.

Before we state Fact 3, we need to introduce a few more definitions.

- If Λ is either a schedule loop or a looped schedule, we say that Λ is **coprime** if not all iterands of Λ are schedule loops, or if all iterands of Λ are schedule loops, and there does not exist an integer $j > 1$ that divides all of the iteration counts of the iterands of Λ.

- We say that a single appearance schedule S is **fully reduced** if S is coprime and every schedule loop contained in S is coprime.

For example, the schedule loops $(5(3A)(7B))$ and $(70C)$ are coprime, while $(3(4A)(2B))$ and $(10(7C))$ are not coprime; similarly, the looped schedules $(\infty A(7B)(7C))$ and $(\infty(2A)(3B))$ are coprime, while the looped schedules $(\infty(4AB))$ and $(\infty(6AB)(3C))$ are not. From our discussion of Fact 2, we know that non-coprime schedules or loops may result in significantly higher buffer memory requirements than their factored counterparts. It is shown in [5] that given a valid single appearance schedule, we can repeatedly apply the factoring transformation to derive from it a valid fully reduced schedule. As a consequence, we have the following fact.

Fact 3: Suppose that G is a consistent SDF graph and S is a valid single appearance schedule for G. Then there exists a valid single appearance schedule S' for G such that S' is fully reduced and $buffer\_memory(S') \le buffer\_memory(S)$.

2 R-schedules

Let G be a chain-structured SDF graph with actors $A_1, A_2, \ldots, A_m$ and edges $\alpha_1, \alpha_2, \ldots, \alpha_{m-1}$ such that each α_k is directed from A_k to A_{k+1}. In the trivial case, $m = 1$, we immediately obtain (∞A_1) as a valid single appearance schedule for G. Otherwise, given any $i \in \{1, 2, \ldots, m-1\}$, define

$$left\,(i) \equiv subgraph\Big(\{A_1, A_2, \ldots, A_i\}, G \Big), \text{ and}$$

$$right\,(i) \equiv subgraph\Big(\{A_{i+1}, A_{i+2}, \ldots, A_m\}, G \Big).$$

From Fact 1, if (∞S_L) and (∞S_R) are valid single appearance schedules for $left\,(i)$ and $right\,(i)$, respectively, then $(\infty\,(q_L S_L)\ (q_R S_R))$ is a valid single appearance schedule for G, where $q_L = gcd\Big(\{\mathbf{q}_G(A_j) \,\big|\, 1 \le j \le i\} \Big)$ and $q_R = gcd\Big(\{\mathbf{q}_G(A_j) \,\big|\, i < j \le m\} \Big)$.

For example, suppose that G is the SDF graph in Figure 1 and suppose $i = 2$. It is easily verified that $\mathbf{q}_{left\,(i)}(A, B) = (3, 4)^T$ and $\mathbf{q}_{right\,(i)}(C, D) = (3, 2)^T$. Thus, $(\infty S_L) = (\infty\,(3A)\ (4B))$ and $(\infty S_R) = (\infty\,(3C)\ (2D))$ are valid single appearance schedules for $left\,(i)$ and $right\,(i)$, and $(\infty\,(3\,(3A)\ (4B))\ (4\,(3C)\ (2D)))$ is a valid single appearance schedule for Figure 1.

We can recursively apply this procedure of decomposing a chain-structured SDF graph into left and right subgraphs to construct a schedule. However, different sequences of choices for i will in general lead to different schedules. For a given chain-structured SDF graph, we refer to the set of valid single appearance schedules obtainable from this recursive scheduling process as the set of **R-schedules**.

We will use the following fact, which is easily verified from the definition of an R-schedule.

Fact 4: Suppose that G is a chain-structured SDF graph, $|actors\,(G)| > 1$, and $(delay\,(\alpha) = 0), \forall(\alpha \in edges\,(G))$. Then a valid single appearance schedule S for G is an R-schedule if and only if every schedule loop L contained in S satisfies the following property:

(a). L has a single iterand, and this single iterand is an actor; that is, $L = (nA)$ for some $n \in \{1, 2, \ldots, \infty\}$ and some $A \in actors\,(G)$; *or*

(b). L has exactly two iterands, and these two iterands are schedule loops having coprime iteration counts; that is, $L = (m\,(n_1 S_1)\ (n_2 S_2))$, where $m \in \{1, 2, \ldots, \infty\}$; n_1 and n_2 are positive integers; $gcd(n_1, n_2) = 1$; and S_1 and S_2 are looped schedules.

Note that if S is an R-schedule, and $L = (m(n_1 S_1)(n_2 S_2))$ is a two-iterand loop in S, then (a). or (b). must also be satisfied for every schedule loop contained in $(n_1 S_1)$ and for every schedule loop contained in $(n_2 S_2)$; thus, it follows that (∞S_1) and (∞S_2) are also R-schedules.

If a schedule loop L satisfies condition (a) or condition (b) of Fact 4, we say that L is an **R-loop**; otherwise, we call L a **non-R-loop**. Thus, a valid single appearance schedule S is an R-schedule if and only if every schedule loop contained in S is an R-loop.

Now let ε_n denote the number of R-schedules for an n-actor chain-structured SDF graph. Trivially, for a 1-actor graph there is only one schedule obtainable by the recursive scheduling process, so $\varepsilon_1 = 1$. For a 2-actor graph, there is only one edge, and thus only one choice for i, $i = 1$. Since for a 2-actor graph, $left(1)$ and $right(1)$ both contain only one actor, we have $\varepsilon_2 = \varepsilon_1 \times \varepsilon_1 = 1$. For a 3-actor graph, $left(1)$ contains 1 actor and $right(1)$ contains 2 actors, while $left(2)$ contains 2 actors and $right(2)$ contains a single actor. Thus,

$$\begin{aligned}
\varepsilon_3 = {} & (\text{the number of R-schedules when } (i = 1)) \\
& + (\text{the number of R-schedules when } (i = 2)) \\
= {} & (\text{the number of R-schedules for } left(1)) \\
& \times (\text{the number of R-schedules for } right(2)) \\
& + (\text{the number of R-schedules for } left(2)) \\
& \times (\text{the number of R-schedules for } right(1)) \\
= {} & (\varepsilon_1 \times \varepsilon_2) + (\varepsilon_2 \times \varepsilon_1) = 2\varepsilon_1\varepsilon_2 .
\end{aligned}$$

Continuing in this manner, we see that for each positive integer $n > 1$,

$$\varepsilon_n = \sum_{k=1}^{n-1} (\text{the number of R-schedules when } (i = k)) = \sum_{k=1}^{n-1} (\varepsilon_k \times \varepsilon_{n-k}) . \tag{1}$$

The sequence of positive integers generated by (1) with $\varepsilon_1 = 1$ is known as the set of **Catalan numbers**, and each ε_i is known as the $(i-1)$ th Catalan number. Catalan numbers arise in many problems in combinatorics; for example, the number of different binary trees with n vertices is given by the n th Catalan number, ε_n. It can be shown that the sequence generated by (1) is given by

$$\varepsilon_n = \frac{1}{n}\binom{2n-2}{n-1}, \text{ for } n = 1, 2, 3, \ldots, \tag{2}$$

where $\binom{a}{b} \equiv \frac{a(a-1)\ldots(a-b+1)}{a!}$, and it can be shown that the expression on the right hand side of (2) is $\Omega\left(4^n/n\right)$ [8].

For example, the chain-structured SDF graph in Figure 1 consists of four actors, so (2) indicates that this graph has $\frac{1}{4}\binom{6}{3} = 5$ R-schedules. The R-schedules for Figure 1 are $(\infty(3(3A)(4B))(4(3C)(2D)))$, $(\infty(3(3A)(4(1B)(1C)))(8D))$, $(\infty(3(1(3A)(4B))(4C))(8D))$, $(\infty(9A)(4(3(1B)(1C))(2D)))$, and $(\infty(9A)(4(3B)(1(3C)(2D))))$; and the corresponding buffer memory requirements are, respectively, 30, 37, 40, 43, and 45.

The following theorem establishes that the set of R-schedules always contains a schedule that achieves the minimum buffer memory requirement over all valid single appearance schedules.

Theorem 1: Suppose that G is a chain-structured SDF graph; $(delay(\alpha) = 0), \forall(\alpha \in edges(G))$; and S is a valid single appearance schedule for G. Then there exists an R-schedule S' for G such that $buffer\_memory(S') \le buffer\_memory(S)$.

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 non-R-loops in S; we define $I(L)$ to be the number of iterands of L; we define $C(L)$ to be the iteration count of L; and we define $\hat{I}(S) \equiv \sum_{L' \in \; nonR(S)} I(L')$.

First observe that all chain-structured SDF graphs are consistent so no further assumptions are required to assure that valid schedules exist for G, and observe that from Fact 3, there exists a valid fully reduced schedule S_0 for G such that $buffer\_memory(S_0) \le buffer\_memory(S)$.

Now let $L_0 = (nT_1 T_2 \ldots T_m)$ be an innermost non-R-loop in S_0; that is, L_0 is not an R-loop, but every loop nested in L_0 is an R-loop. If $m = 1$ then since S_0 is fully reduced, $L_0 = (n(1T'))$, for some iterand T', where $(1T)$ is an R-loop. Let S_1 be the schedule that results from replacing L_0 with (nT') in S_0. Then clearly, S_1 is also valid and fully reduced, and S_1 generates the same invocation sequence as S_0, so $buffer\_memory(S_1) = buffer\_memory(S_0)$. 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, $\hat{I}(S_1) < \hat{I}(S_0)$.

If on the other hand $m \ge 2$, we define $S_a \equiv (1T_1)$ if T_1 is an actor and $S_a \equiv T_1$ otherwise (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 $\gamma = gcd\left(\{ C(T_i) \mid (2 \le i \le m) \} \right)$, and $B_2, B_3, \ldots, B_m$ are the bodies of the loops $T_2, T_3, \ldots, T_m$, respectively; if $T_2, T_3, \ldots, T_m$ are not all schedule loops, we define $S_b \equiv (1T_2 \ldots T_m)$. Let S_1 be the schedule that results from replacing L_0 with $L_0' = (nS_a S_b)$ in S_0. Now, because L_0 is fully reduced, the iteration counts of S_a and S_b must be coprime. Thus, it is easily verified that S_1 is a valid, fully reduced schedule and that L_0' is an R-loop, and with the aid of Fact 2, it is also easily verified that $buffer\_memory(S_1) \le buffer\_memory(S_0)$.

Finally, observe that S_a and L_0' are R-loops, but S_b may or may not be an R-loop (depending on L_0). Thus, replacing L_0 with L_0' either reduces the number of non-R-loops by one, or it leaves the number of non-R-loops unchanged, and we see that either $\hat{I}(S_1) = \hat{I}(S_0) - I(L_0)$, or $\hat{I}(S_1) = \hat{I}(S_0) - I(L_0) + I(S_b)$. Since $I(S_b) = I(L_0) - 1 < I(L_0)$, we again conclude that $\hat{I}(S_1) < \hat{I}(S_0)$.

Thus, from S_0, we have constructed a valid, fully reduced schedule S_1 such that $buffer\_memory(S_1) \le buffer\_memory(S_0) \le buffer\_memory(S)$ and $\hat{I}(S_1) < \hat{I}(S_0)$.

Clearly, if $\hat{I}(S_1) \ne 0$, we can repeat the above process to obtain a valid, fully reduced single appearance schedule S_2 such that $buffer\_memory(S_2) \le buffer\_memory(S_1)$ and $\hat{I}(S_2) < \hat{I}(S_1)$. Continuing in this manner, we obtain a sequence of valid single appearance schedules $S_0, S_1, S_2, S_3, \ldots$ such that for each S_i in the sequence with $i > 0$, $buffer\_memory(S_i) \le buffer\_memory(S)$, and $\hat{I}(S_i) < \hat{I}(S_{i-1})$. Since $\hat{I}(S_0)$ is finite, we cannot go on generating S_i's indefinitely — eventually, we will arrive at an S_n, $n \ge 0$, such that $\hat{I}(S_n) = 0$. From Fact 4, S_n is an R-schedule. *QED*.

Theorem 1 guarantees that from within the set of R-schedules for a given chain-structured SDF graph, we can always find a single appearance schedule that minimizes the buffer memory requirement over all single appearance schedules; however, from (2), we know that in general, the R-schedules are too numerous for exhaustive evaluation to be feasible. The following section presents a dynamic programming algorithm that obtains an optimal R-schedule in cubic time.

3 Dynamic Programming Algorithm

The problem of determining the R-schedule that minimizes the buffer memory requirement for a chain-structured SDF graph can be formulated as an optimal parenthesization problem.

A familiar example of an optimal parenthesization problem is matrix chain multiplication [8, 10]. In matrix chain multiplication, we must compute the matrix product $M_1M_2\ldots M_n$, assuming that the dimensions of the matrices are compatible with one another for the specified multiplication. There are several ways in which the product can be computed. For example, with $n = 4$, one way of computing the product is $(M_1(M_2M_3))M_4$, where the parenthesizations indicate the order in which the multiplies occur. Suppose that M_1, M_2, M_3, M_4 have dimensions $10 \times 1,\ 1 \times 10,\ 10 \times 3,\ 3 \times 2$, respectively. It is easily verified that computing the matrix chain product as $((M_1M_2)M_3)M_4$ requires 460 scalar multiplications, whereas computing it as $(M_1(M_2M_3))M_4$ requires only 120 multiplications (assuming that we use the standard algorithm for multiplying two matrices).

Thus, we would like to determine an optimal way of placing the parentheses so that the total number of scalar multiplications is minimized. This can be achieved using a dynamic programming approach. The key observation is that any optimal parenthesization splits the product $M_1M_2\ldots M_n$ between M_k and M_{k+1} for some k in the range $1 \le k \le (n-1)$, and thus the cost of this optimal parenthesization is the cost of computing the product $M_1M_2\ldots M_k$, plus the cost of computing $M_{k+1}M_{k+2}\ldots M_n$, plus the cost of multiplying these two products together. In an optimal parenthesization, the subchains $M_1M_2\ldots M_k$ and $M_{k+1}M_{k+2}\ldots M_n$ must themselves be parenthesized optimally. Hence this problem has the optimal substructure property and is thus amenable to a dynamic programming solution.

Determining the optimal R-schedule for a chain-structured SDF graph is similar to the matrix chain multiplication problem. Recall the example of Figure 1. Here $\mathbf{q}(A, B, C, D) = (9, 12, 12, 8)^T$; an optimal R-schedule is $(\infty(3(3A)(4B))(4(3C)(2D)))$; and the associated buffer memory requirement is 30. Therefore, as in the matrix chain multiplication case, the optimal parenthesization (of the schedule body) contains a break in the chain at some $k \in \{1, 2, \ldots, (n-1)\}$. Because the parenthesization is optimal, the chains to the left of k and to the right of k must both be parenthesized optimally. Thus, we have the optimal substructure property.

Now given a chain-structured SDF graph G consisting of actors $A_1, A_2, \ldots, A_n$ and edges $\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$, such that each α_k is directed from A_k to A_{k+1}, given integers i, j in the range $1 \le i \le j \le n$, denote by $b[i, j]$ the minimum buffer memory requirement over all R-schedules for $subgraph\left(\{A_i, A_{i+1}, \ldots, A_j\}, G\right)$. Then, the minimum buffer memory requirement over all R-schedules for G is $b[1, n]$. If $1 \le i < j \le n$, then,

$$b[i,j] = min\left(\{(b[i,k] + b[k+1,j] + c_{i,j}[k]) \mid (i \le k < j)\}\right), \tag{3}$$

where $b[i, i] = 0$ for all i ,and $c_{i,j}[k]$ is the memory cost at the split if we split the chain at A_k. It is given by

$$c_{i,j}[k] = \frac{\mathbf{q}_G(A_k)\, produced(\alpha_k)}{gcd\left(\{\mathbf{q}_G(A_m) \mid (i \le m \le j)\}\right)}. \tag{4}$$

The *gcd* term in the denominator arises because from Fact 1, the repetitions vector $\mathbf{q}'$ of $subgraph\left(\{A_i, A_{i+1}, \ldots, A_j\}, G\right)$ satisfies $\mathbf{q}'(A_p) = \dfrac{\mathbf{q}_G(A_p)}{gcd\left(\{\mathbf{q}_G(A_m) \mid (i \le m \le j)\}\right)}$, for all $p \in \{i, i+1, \ldots, j\}$.

A dynamic programming algorithm derived from the above formulation is specified in Figure 4. In this algorithm, first the quantity $gcd\left(\{\mathbf{q}_G(A_m) \mid (i \le m \le j)\}\right)$ is computed for each subchain $A_i, A_{i+1}, \ldots, A_j$. Then the two-actor subchains are examined, and the buffer memory requirements for these subchains are recorded. This information is then used to determine the minimum buffer memory requirement and the location of the split that achieves this minimum for each three-actor subchain. The minimum buffer memory requirement for each three-actor sub-

procedure ScheduleChainGraph
input: a chain-structured SDF graph G consisting of actors $A_1, A_2, \ldots, A_n$
and edges $\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$ such that each α_i is directed from A_i to A_{i+1}.
output: an R-schedule body for G that minimizes the buffer memory requirement.

for $i = 1, 2, \ldots, n$ /* Compute the gcd's of all subchains */
    $\mathrm{GCD}[i, i] = \mathbf{q}_G(A_i)$
    for $j = (i+1), (i+2), \ldots, n$
        $\mathrm{GCD}[i, j] = gcd\left(\{\mathrm{GCD}[i, j-1], \mathbf{q}_G(A_j)\} \right)$

for $i = 1, 2, \ldots, n$ $\mathrm{Subcosts}[i, i] = 0$;
for $\mathrm{chain\_size} = 2, 3, \ldots, n$
    for $\mathrm{right} = \mathrm{chain\_size}, \mathrm{chain\_size} + 1, \ldots, n$
        $\mathrm{left} = \mathrm{right} - \mathrm{chain\_size} + 1$;
        $\mathrm{min\_cost} = \infty$;
        for $i = 0, 1, \ldots, \mathrm{chain\_size} - 2$
            $\mathrm{split\_cost} = (\mathbf{q}_G(A_{\mathrm{left}+i}) / \mathrm{GCD}[\mathrm{left}, \mathrm{right}]) \times produced(\alpha_{\mathrm{left}+i})$;
            $\mathrm{total\_cost} = \mathrm{split\_cost} + \mathrm{Subcosts}[\mathrm{left}, \mathrm{left}+i] + \mathrm{Subcosts}[\mathrm{left}+i+1, \mathrm{right}]$;
            if $(\mathrm{total\_cost} < \mathrm{min\_cost})$
                $\mathrm{split} = i$; $\mathrm{min\_cost} = \mathrm{total\_cost}$
        $\mathrm{Subcosts}[\mathrm{left}, \mathrm{right}] = \mathrm{min\_cost}$; $\mathrm{SplitPositions}[\mathrm{left}, \mathrm{right}] = \mathrm{split}$;
output $\mathrm{ConvertSplits}(1, n)$; /* Convert the SplitPositions array into an R-schedule */

procedure ConvertSplits(L, R)
implicit inputs: the SDF graph G and the GCD and SplitPositions arrays
of procedure *ScheduleChainGraph*.
explicit inputs: positive integers L and R such that $1 \le L \le R \le n = |actors(G)|$.
output: An R-schedule body for $subgraph\left(\{A_L, A_{L+1}, \ldots, A_R\}, G \right)$ that minimizes
the buffer memory requirement.

if $(L = R)$ **output** A_L
else
    $s = \mathrm{SplitPositions}[L, R]$; $i_L = \mathrm{GCD}[L, L+s] / \mathrm{GCD}[L, R]$;
    $i_R = \mathrm{GCD}[L+s+1, R] / \mathrm{GCD}[L, R]$;
    output $(i_L \mathrm{ConvertSplits}(L, L+s))\ (i_R \mathrm{ConvertSplits}(L+s+1, R))$;

Figure 4.

chain A_i, A_{i+1}, A_{i+2} is stored in entry $[i, i+2]$ of the array Subcosts, and the index of the edge corresponding to the split is stored in entry $[i, i+2]$ of the SplitPositions array. This data is then examined to determine the minimum buffer memory requirement for each four-actor subchain, and so on, until the minimum buffer memory requirement for the n-actor subchain, which is the original graph G, is determined. At this point, procedure ConvertSplits is called to recursively construct an optimal R-schedule from a top-down traversal of the optimal split positions stored in the SplitPositions array.

Assuming that the components of $\mathbf{q}_G$ are bounded, which makes the *gcd* computations elementary operations, it is easily verified that the time complexity of ScheduleChainGraph is dominated by the time required for the innermost **for** loop — the (**for** $i = 0, 1, \ldots, \text{chain\_size} - 2$) loop — and the running time of one iteration of this loop is bounded by a constant that is independent of n. Thus, the following theorem guarantees that under our assumptions, the running time of ScheduleChainGraph is $O(n^3)$ and $\Omega(n^3)$.

Theorem 2: The total number of iterations of the (**for** $i = 0, 1, \ldots, \text{chain\_size} - 2$) loop that are carried out in ScheduleChainGraph is $O(n^3)$ and $\Omega(n^3)$.

Proof: This is straightforward; see [4] for the derivation.

4 Example: Sample Rate Conversion

Digital audio tape (DAT) technology operates at a sampling rate of 48 kHz, while compact disk (CD) players operate at a sampling rate of 44.1 kHz. Interfacing the two, for example, to record a CD onto a digital tape, requires a sample rate conversion.

The naive way to do this is shown in Figure 5(a). It is more efficient to perform the rate change in stages. Rate conversion ratios are chosen by examining the prime factors of the two sampling rates. The prime factors of 44,100 and 48,000 are $2^2 3^2 5^2 7^2$ and $2^7 3^1 5^3$, respectively.

Thus, the ratio $44{,}100 : 48{,}000$ is $3^1 7^2 : 2^5 5^1$, or $147 : 160$. One way to perform this conversion in four stages is $2{:}1$, $4 : 3$, $4 : 7$, and $5 : 7$. Figure 5(b) shows the multistage implementation. Explicit upsamplers and downsamplers are omitted, and it is assumed that the FIR filters are general polyphase filters [7].

Here $\mathbf{q}(A, B, C, D, E, F) = (147, 147, 98, 28, 32, 160)^T$; the optimal looped schedule given by our dynamic programming approach is $(\infty(7(7(3AB)(2C))(4D))(32E(5F)))$; and the associated buffer memory requirement is 264. In contrast, the alternative schedule $(\infty(147A)(147B)(98C)(28D)(32E)(160F))$ has a buffer memory requirement of 1021 if a separate buffer is used for each edge and a buffer-memory requirement of 294 if one shared buffer is used. This is an important savings with regard to current technology: a buffer memory requirement of 264 will fit in the on-chip memory of most existing programmable digital signal processors, while a buffer memory requirement of 1021 is too high for all programmable digital signal processors, except for a small number of the most expensive ones. The savings of 30 ($10\,\%$) over using a single shared buffer can also be significant on chips that only have on the order of 1000 words of memory. It can be verified that the latency of the optimally nested schedule is given by $146T + E_A + E_B + 2E_C + 4E_D + E_E$, as opposed to

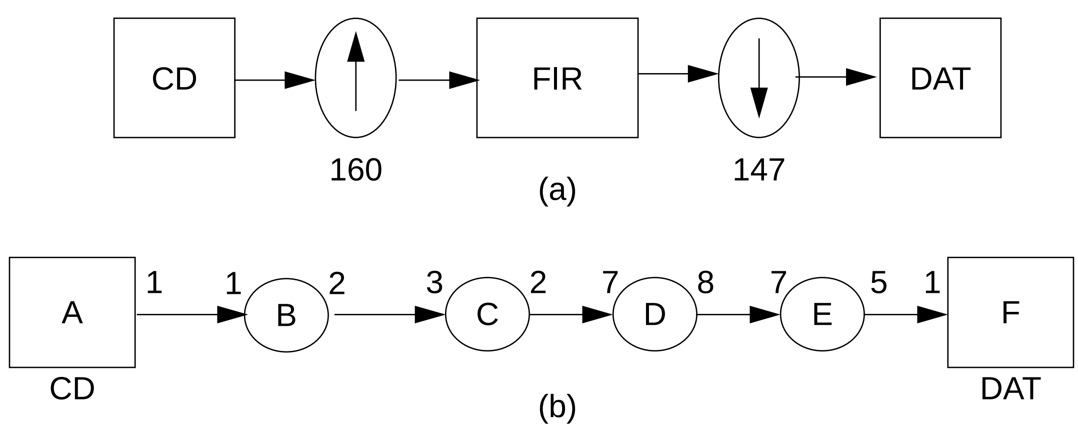

Figure 5. (a). CD to DAT sample rate change system.
(b). Multi-stage implementation of a CD to DAT sample rate system.

$146T + E_A + 147E_B + 98E_C + 28E_D + 32E_E$ for the naive schedule. If we take $T = 500$ (for example, a 22.05Mhz chip has 22.05Mhz/44.1khz = 500 instruction cycles in one sample period of the CD actor), and $E_A = 10, E_B = E_C = E_D = E_E = 100$, then the two latencies are 73810 and 103510 instruction cycles; the nested schedule has 29% less latency.

One more advantage that a nested schedule can have over the naive schedule with shared buffering is in the amount of *input buffering* required. Some DSP chips have a feature where a dedicated I/O manager can write incoming samples to a buffer in on-chip memory, the size of which can be programmed by the user. If the single appearance schedule spans more than one sample period, then input buffering is a useful feature since it avoids the need for interrupts. Interrupts can be expensive; for example, on the Motorola 56000, input buffering is not available and interrupts have to be used; each interrupt entails a loss of about 70 instruction cycles. Chips that have input buffering include the Analog Devices ADSP 2100 and the Sproc processor from Star Semiconductor. If we compute the amount of input buffering required by the naive schedule, we find that it is $((147 + 98 + 28 + 32)100 + 160 \times 10) / 500 \cong 65$, whereas for the optimally nested schedule, it is given by $(100 + 200 + 400 + 3200 + 1600) / 500 \cong 11$.

5 An Efficient Heuristic

Our dynamic programming solution for chain-structured graphs runs in $O(m^3)$ time, where m is the number of actors. As a quicker alternative solution, we developed a more time-efficient heuristic approach. The heuristic is simply to introduce the parenthesization on the edge where the minimum amount of data is transferred. This is done recursively for each of the two halves that result. The running time of the heuristic is given by the recurrence

$$T(n) = T(n-k) + T(k) + O(n) , \tag{5}$$

where k is the actor at which the split occurs. This is because we must compute the *gcd* of the repetitions vector components of the k actors to the left of the split, and the *gcd* of the repetitions of the $n-k$ actors to the right. This takes $O(n)$ time assuming that the repetitions vector components are bounded. Computing the minimum of the data transfers takes a further $O(n)$ time since there are $O(n)$ edges to consider. The worst case solution to this recurrence is $O\left(n^2\right)$, but the average case running time is $O(n \bullet \log n)$ if $k = \Omega(n)$. It can be verified that this heuristic gives the R-schedule with the minimum buffer memory requirement, $(\infty(3(3A)(4B))(4(3C)(2D)))$, for Figure 1.

We have evaluated the heuristic on $10,000$ randomly generated 50-actor chain-structured SDF graphs, and we have found that on average, it yields a buffer memory requirement that is within 60% of the optimal cost. For each random graph, we also compared the heuristic's solution to the worst-case schedule and to a randomly-generated R-schedule. On average, the worst-case schedule had over 9000 times higher cost than the heuristic's solution, and the random schedule had 225 times higher cost. Furthermore, the heuristic outperformed the random schedule on 97.8 percent of the trials. We also note that in over 99% of the randomly generated 50-actor chain-structured SDF graphs, the shared-buffer cost for the naive single appearance schedule was worse than the cost of the nested schedule given by the heuristic. Unfortunately, the heuristic does not perform well on the example of Figure 5 — it achieves a buffer memory requirement of 565, which is over double of what is required by an optimum R-schedule. In comparison, the worst R-schedule for Figure 5 has a buffer memory requirement of 755.

6 Extensions

In this section we present three useful extensions of the dynamic programming solution developed in Section 3. First, the algorithm can easily be adapted to optimally handle chain-structured graphs that have delays on one or more of the edges. This requires that we modify the computation of $c_{i,j}[k]$, the amount of memory required to split the subchain $A_i, A_{i+1}, \ldots, A_j$ between the actors A_k and A_{k+1}. This cost now gets computed as

$$c_{i,j}[k] = \frac{1}{r}\mathbf{q}_G(A_k)\,produced(\alpha_k) + delay(\alpha_k)\text{, where } r = gcd\left(\{\mathbf{q}_G(A_m) \mid (i \le m \le j)\}\right),$$

if $delay(\alpha_k) < \frac{1}{r}\mathbf{q}_G(A_k)\,produced(\alpha_k)$; otherwise (if

$delay(\alpha_k) \geq \frac{1}{r}\mathbf{q}_G(A_k)\,produced(\alpha_k)$), $c_{i,j}[k]$ gets computed as $c_{i,j}[k] = delay(\alpha_k)$. Accordingly, if the optimum split extracted in a given invocation of *ConvertSplits* (Figure 4) corresponds to a split in which the latter condition applied in the computation of $c_{i,j}[k]$, then *ConvertSplits* returns $(i_R \mathrm{ConvertSplits}(L+s+1, R))\,(i_L \mathrm{ConvertSplits}(L, L+s))$; otherwise, *ConvertSplits* returns $(i_L \mathrm{ConvertSplits}(L, L+s))\,(i_R \mathrm{ConvertSplits}(L+s+1, R))$, as in the original version. This requires a method for keeping track of which condition applies to each of the optimum subchain splits, which can easily be incorporated, for example, by varying the sign of the associated entry in the *SplitPositions* array.

Second, as mentioned in Section 1, the technique applies to the more general class of well-ordered SDF graphs. A well-ordered graph is one where the partial order is a total order; chain-structured graphs are a special case of these. Again, this requires modifying the computation of $c_{i,j}[k]$. Here, this cost gets computed as

$$c_{i,j}[k] = \frac{\sum_{\alpha \in S_{i,j,k}} \mathbf{q}_G(A_k)\,produced(\alpha_k)}{gcd\left(\{\mathbf{q}_G(A_m) \mid (i \leq m \leq j)\}\right)}, \qquad (6)$$

where

$$S_{i,j,k} \equiv \{\beta \mid \left(source(\beta) \in \{A_i, A_{i+1}, \ldots, A_k\}\right) \quad ; \quad \textbf{and} \left(sink(\beta) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\}\right)\}$$

that is, $S_{i,j,k}$ is the set of edges directed from one side of the split to the other side.

The dynamic programming technique of Section 3 can also be applied to reducing the buffer memory requirement of a given single appearance schedule for an arbitrary acyclic SDF graph (not necessarily chain-structured or well-ordered). To explain this extension, we need to define the concept of a **topological sort**. A topological sort of a directed acyclic graph consisting of the set of vertices V and the set of edges E is an ordering $v_1, v_2, \ldots, v_{|V|}$ of the members of V such that for each $e \in E$, $((source(e) = v_i) \textbf{ and } (sink(e) = v_j)) \Rightarrow (i < j)$; that is, the source vertex of each edge occurs earlier in the ordering than the sink vertex.

Suppose we are given a valid single appearance schedule S for an acyclic SDF graph and again for simplicity, assume that the edges in the graph contain no delay. Let $\Psi = B_1, B_2, \ldots, B_m$ denote the sequence of lexical actor appearances in S (for example, for the schedule $(\infty(4A(2FD))C)$, $\Psi = A, F, D, C$). Thus, since S is a single appearance schedule, Ψ must be a topological sort of the associated acyclic SDF graph. The technique of Section 3 can easily be modified to optimally "re-parenthesize" S into the optimal single appearance schedule (with regard to buffer memory requirement) associated with the topological sort Ψ. The technique is applied to the sequence Ψ, with $c_{i,j}[k]$ computed as in (6). It can be shown that the algorithm runs in time $O(|V|^3)$, where $|V|$ is the number of nodes in the graph.

Thus, given any topological sort Ψ^* for a consistent acyclic SDF graph, we can efficiently determine the single appearance schedule that minimizes the buffer memory requirement over all valid single appearance schedules for which the sequence of lexical actor appearances is Ψ^*.

Another extension applies when we relax the assumption that each edge is mapped to a separate block of memory, and allow buffers to be overlaid in the same block of memory. There are several ways in which buffers can be overlaid; the simplest is to have one memory segment of size

$$CS_{i,j} \equiv \frac{max\left(\{produced(\alpha_k) \times \mathbf{q}(A_k) \mid (i \le k < j)\}\right)}{gcd\left(\{\mathbf{q}(A_i), \mathbf{q}(A_{i+1}), \ldots, \mathbf{q}(A_j)\}\right)} \tag{7}$$

for the subchain $A_i, A_{i+1}, \ldots, A_j$ (as explained in Section 1.1). We follow this computation with

$$b'[i,j] = min\left(\{b[i,j], CS_{i,j}\}\right), \tag{8}$$

to determine amount of memory to use for buffering in the subchain $A_i, A_{i+1}, \ldots, A_j$. In general, this gives us a combination of overlaid and non-overlaid buffers for different sub-chains. Incorporating the techniques of this section with more general overlaying schemes is a topic for future work.

7 Acyclic SDF Graphs

Consistent acyclic SDF graphs are guaranteed to have single appearance schedules since a naive schedule corresponding to any topological sort is a valid single appearance schedule. For arbitrary graphs (not necessarily acyclic), necessary and sufficient conditions are given in [4] for single appearance schedules to exist, and efficient algorithms are given to find such schedules whenever they exist. These techniques require decomposing each strongly connected component into an acyclic graph that consists of *clusters*, or supernodes, of smaller strongly connected components, constructing a single appearance schedule for this acyclic graph, and then recursively applying this procedure to each of the clustered strongly connected components to obtain the subschedule for the corresponding supernode. For each decomposed strongly connected component, there is a "top-level" cost associated with the edges that are not contained in any of the associated clusters, and thus the total buffering cost of a general SDF graph involves the top-level cost for each strongly connected component in the cluster hierarchy, in addition to the buffering cost for each acyclic graph that occurs in the hierarchy. Furthermore, to attain the lowest buffering cost, it may be necessary to increase the extent of some strongly connected components by clustering neighboring actors together with actors in the strongly connected components before decomposing the components [4]. Hence, graphs with cycles are significantly more difficult to construct buffer-optimal single appearance schedules for than acyclic graphs.

The number of topological sorts in an acyclic graph can be exponential in the size of the graph; for example, a complete bipartite graph with $2n$ nodes has $(n!)^2$ possible topological sorts. Each topological sort gives a valid naive single appearance schedule. An optimal reparenthesization of this schedule is then computed by applying the dynamic programming algorithm. The problem is therefore to determine the topological sort that will give the lowest buffer memory requirement when nested optimally. For example, the graph in Figure 6 shows a bipartite graph with 4 nodes. The repetitions vector for the graph is given by $(12, 36, 9, 16)^T$, and there are 4 possible topological sorts for the graph. The naive schedule corresponding to the topological sort $ABCD$ is given by $(\infty\,(12A)\;(36B)\;(9C)\;(16D))$. This can be parenthesized as $(\infty\,(3\,(4A)\;(3\,(4B)\;C))\;(16D))$, and this schedule has a buffer memory requirement of 208. The naive schedule corresponding to the topological sort $ABDC$, when parenthesized optimally, gives the schedule $(\infty\,(4\,(3A)\;(9B)\;(4D))\;(9C))$, with a buffer memory requirement of 120.

A heuristic solution for this problem can be based on extending the main idea that was used in the heuristic for the chain-structured graph case: find the *cut* (a partition of the set of actors) of the graph across which the minimum amount of data is transferred and schedule 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 nodes 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 nodes in the subsets produced by the partition, thus reducing the buffer memory requirement. To see that having gcd’s greater than one for the subsets produced is beneficial to memory reduction, consider figure 6. If we formed the partition that had actor B on one side of the cut and actors A, C, D on the other

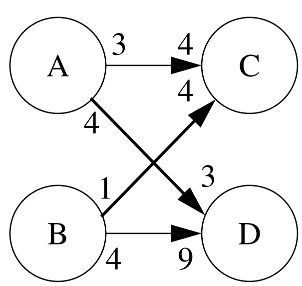

Figure 6. A bipartite SDF graph to illustrate the different buffer memory requirements possible with different topological sorts.

side of the cut, we get the loop bodies $(36B)$ and $((12A)(9C)(16D))$ and do not immediately see a reduction in buffering requirements since the repetitions of A, C, D are co-prime. However, a partition with A, B, C on the same side of the cut immediately gives us a reduction since the schedule body $((12A)(36B)(9C))$ can be factored as $(3(4A)(12B)(3C))$, and this reduces the memory for the subgraph consisting of actors A, B, C. In general, by constraining the sizes of the partition, we increase the probability of being able to factor schedule bodies so that a reduction in memory is obtained in each stage of the recursion. Needless to say, this is a greedy approach which is likely to fail sometimes but has proved to be a good rule of thumb for most instances.

7.1 A Heuristic to find Minimum Legal Cuts into Bounded Sets

Suppose that G is an SDF graph, and let $V = actors(G)$ and $E = edges(G)$. A cut G is a partition of the vertex set 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 $source(e) \in V_L$ and $sink(e) \in V_R$. Given a *bounding constant* $K \le |V|$, the cut results in bounded sets if it satisfies

$$|V_R| \le K\ ,\ \ |V_L| \le K\ . \tag{9}$$

The weight of an edge e is defined as

$$w(e) = \mathbf{q}_G(source(e)) \times produced(e)\ . \tag{10}$$

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 (10). Since the related problem of finding a minimum cut (not necessarily legal) into bounded sets is NP-complete [9], and the problem of finding an acyclic partition of a graph is NP-complete [9],

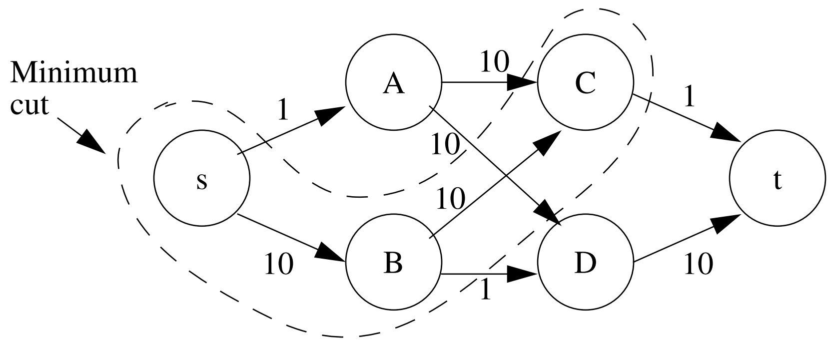

Figure 7. The min-cut given by the max-flow-min-cut theorem is not equal to the min-legal cut for this graph.

we believe this problem to be NP-complete as well even though we have not discovered a proof. Kernighan and Lin [13] devised a heuristic procedure for computing cuts into bounded sets but they considered only undirected graphs. Methods based on network flows [8] do not work because the minimum cut given by the max-flow-min-cut theorem may not be legal and may not be bounded. The graph in Figure 7, where the weight on the edge denotes the capacity of that edge, illustrates this. The maximum flow into vertex t is seen to be 3 (1 unit of flow along the path $sBCt$, 1 unit along $sADt$ and 1 unit along $sBDt$) and this corresponds to the cut where $V_L = \{s, B, C\}$ and $V_R = \{A, D, t\}$. The value of the cut is given by $1 + 1 + 1 = 3$ (note that the definition of the value of a cut in network flow theory is defined as sum of the capacities of the edges crossing the cut in the s to t direction only) but the cut is not legal because of the reverse edge from A to C. Indeed, the minimum weight legal cut for this graph has a value of 11, corresponding to the cut where $V_L = \{s\}$.

Therefore, 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. A vertex v is defined to be a **descendant** of a vertex u if there is a directed path from u to v and a vertex v is a **ancestor** of vertex u if there is a directed path from v to u. A vertex u is **inde-**

```
procedure MinimumLegalCutIntoBoundedSets
input: weighted digraph G = (V, A), and a bound b. output: V_R, V_L.
for each u ∈ V
        S = desc(u), S̄ = V\S
        cutVal = cut(S̄, S)
        T_L(u) ← independent(u) ∩ boundary(S̄)
        for each a ∈ T_L(u)
            E(a) = Σ_{x ∈ S} w(a, x)
            I(a) = Σ_{x ∈ S̄} w(x, a)
            D(a) = I(a) − E(a)        /* Cost difference if this vertex is moved over */
        end for
        [D, Idx] ← sort(D)
        k ← 1
        while (|S| < b & D(k) < 0 & k < |T_L(u)|)
            S ← S ∪ {Idx(k)}
            S̄ ← S̄ \ {Idx(k)}
            cutVal ← cutVal + D(k)
            k ← k + 1
        end while
        minCutVal ← min(minCutVal, cutVal)
        if (mincutVal ≡ cutVal), V_L ← S̄, V_R ← S, end if
        P = ancs(u), P̄ = V \ P
        T_R(u) ← independent(u) ∩ boundary(P̄)
        for each a ∈ T_R(u)
            E(a) = Σ_{x ∈ P} w(x, a)
            I(a) = Σ_{x ∈ P̄} w(a, x)
            D(a) = I(a) − E(a)
        end for
        /* Carry out the same type of steps as above to determine the partition */
end for
/* minCutVal, V_L, V_R correspond to the minimum legal cut. */
```

pendent of v if u is neither a descendant nor an ancestor of v. Define the set of ancestors as $ancs(v) = \{v\} \bigcup ancestors(v)$, and descendants as $desc(v) = \{v\} \bigcup descendants(v)$, and consider a cut produced by setting $V_L = ancs(v)$, $V_R = V \setminus V_L$ for some vertex v. Consider the set $T_R(v)$ of independent, *boundary* nodes of v in V_R. A **boundary** node in V_R is a node that is not the predecessor of any other node in V_R. Following Kernighan and Lin [13], for each of these nodes, we can compute the cost difference that results if the node is moved into V_L. This cost difference for a node a in $T_R(v)$ is defined to be the difference between the total weight of all the arcs out of a and the total weight of all arcs into a. We then move those nodes across that reduce the cost. We apply this optimization step for all cuts of the form $ancs(v)$ and $desc(v)$ for each vertex v in the graph and take the best one as the minimum cut. The algorithm is shown in Figure 8. Since a greedy strategy is being used to move nodes across, and only the boundary nodes are considered, examples can be constructed where the heuristic will not give optimal cuts. Since there are $|V|$ nodes in the graph, $2|V|$ cuts are examined. Moreover, the cut produced will have bounded sets since cuts that produce unbounded sets are discarded. For example, one of the cuts examined by the heuristic for the graph in Figure 7, with bounding constant $K = |V| - 1$, is $ancs(A) = \{s, A\}$. This cut has a value of 30. The set of independent, boundary nodes of A in V_R is $\{B\}$, and the cost difference for B is given by $11 - 10 = 1$. Hence, B will not be moved over. The cut produced by considering $ancs(C) = \{s, A, B, C\}$ has a value of 12. The cost difference for the independent vertex D is given by $10 - 11 = -1$; hence, D is moved into V_L to yield a cut of value 11, and thus, in this example, the heuristic finds the minimum weight legal cut.

Delays on arcs are handled as follows. If the number of delays D on some arc e satisfies

$$D \geq \mathbf{q}_G(source(e)) \times produced(e), \tag{11}$$

then the size of the buffer on this arc need not be any greater than D. However, if e crosses the cut, then the size of the buffer will become $D + \mathbf{q}_G(source(e)) \times produced(e)$. Hence, an arc that has D delays, where D satisfies equation 11, is *tagged*; a tagged arc does not affect the legality of the cut (in other words, the heuristic ignores tagged arcs when it constructs the legal cut) but affects the cost of the cut: if a tagged arc crosses the cut in the reverse direction, the cost of the arc is given by D, and if the tagged arc crosses the cut in the forward direction, the cost is given by $D + \mathbf{q}_G(source(e)) \times produced(e)$. This will discourage the heuristic is choosing partitions where tagged arcs cross the cut in the forward direction.

The running time of the heuristic for computing the legal minimum cut into bounded sets can be determined as follows. Computing the descendents or ancestors of a vertex can be done by using breadth-first-search; this takes time $\Theta(|V| + |E|)$. The breadth-first-search will also give us the independent nodes in the complement set. Finding and computing the cost difference for each of the boundary nodes in the set of independent nodes takes at most $O(|E|)$ steps. Sorting the cost differences takes $O(|V| \bullet \log(|V|))$ steps at most, and moving the nodes that reduce the cost takes $O(|V|)$ time at most. Since a cut is determined for every vertex twice, the total running time is $O(|V||E| + |V|^2 \bullet \log(|V|))$.

The heuristic for generating an schedule for the acyclic graph now proceeds by partitioning the graph by computing the legal minimum cut and forming the schedule body $(r_L S_L)(r_R S_R)$ 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 schedule bodies for G_L and G_R respectively. The schedule bodies S_L, S_R are obtained recursively by partitioning G_L and G_R. Once the entire schedule body has been constructed, the dynamic programming algorithm is run to re-parenthesize the schedule to possibly give a better nesting. Letting $n = |V|$, the running time for this heuristic can be determined by solving the recurrence $T(n) = T(n-k) + T(k) + O(n|E| + n^2 \bullet \log(n))$, where $k = |V_L|$ and $n - k = |V_R|$. If we choose the bound K in (9) to be a constant factor of the graph size, for example, 3/4, then it can be shown easily that $T(n) = O(|V||E| + |V|^2 \bullet \log(|V|))$. If we do not bound the size of the sets to be a constant factor of the graph size, then the worst case running time is $O(|V|^2|E| + |V|^3 \bullet \log(|V|))$. The reparenthesizing step that is run at the end uses the dynamic programming algorithm and requires $O(|V|^3)$ running time. Thus the overall running time is given by $O(|V|^3)$.

7.2 Experimental Results

The heuristic was tested on hundreds of randomly generated 20 vertex, 30 vertex and 50 vertex SDF graphs. The random graphs were sparse, having $2|V|$ edges on average. The numbers produced and consumed on the arcs were restricted to be less than or equal to 10 in order to prevent huge rate changes (and thus, repetitions vectors) from occurring. The bounding constant $K = 3(|V|/4)$ was used in the heuristic for generating legal minimum cuts into bounded sets; other bounds gave inferior results. The costs given by the heuristic were compared to the best cost determined by just constructing a number of random topological sorts, and nesting each optimally to determine the cost. Since a random topological sort can be found in linear time, the time to determine a random schedule that has been nested optimally is given by $O(|V|^3)$. A measurement of the actual running time of the heuristic on a 50 node graph shows that we can construct and examine 2 random schedules in approximately the same time that the heuristic takes to construct its schedule. Hence, the heuristic was tested on random graphs against the cost of the better of two randomly constructed schedules. The heuristic gave a better schedule more than 75% of the time. However, if the number of random schedules that are examined is increased to 10 (with a 5-fold increase in running time), the best of the random schedules was better than that given by the heuristic 75% of the time. Hence, by increasing the number of random schedules that are examined, we can construct better schedules than the heuristic, although this comes at the expense of increased running time. Moreover, even when the heuristic produces schedules worse than randomly constructed ones, it is still very close to the best random schedule, whereas the random schedules can produce very bad schedules. Hence, the heuristic gives good schedules almost all the time, even if slightly better ones could be constructed by examining a large number of random schedules. In addition, we tested this heuristic against many other heuristics that can be devised for the problem, including several that are based on the PGAN approach reported in [6], which is outlined below.

One of the earliest techniques for jointly optimizing both code and data requirements for SDF graphs was the PGAN (*pairwise grouping of adjacent nodes*) approach [6]. This approach, which was devised for general SDF graphs (not necessarily acyclic), involves constructing a cluster hierarchy by clustering two vertices at each clustering step. The cluster selection is based on frequency of occurrence — the pair of adjacent actors is selected whose associated subgraph has the highest repetition count. In [6] it is shown that the approach naturally favors nested loops over "flat" hierarchies, and thus reduces the buffer memory requirement over naive schedules. We have evaluated the PGAN heuristic, and several related heuristics that we derived by maintaining the pairwise grouping property, and changing the function that is used to prioritize cluster candidates. In each case, the dynamic programming extension of Section 6 was applied as a post-processing step to optimally reparenthesize the PGAN schedule. We found that on 50 node random graphs generated from the same model as the one we used to evaluate the heuristic of Section 7, the original PGAN heuristic outperformed a single random schedule 77% of the time, and the best modified PGAN heuristic outperformed a single random schedule 82% percent of the time, while the heuristic of Section 7 outperformed a single random schedule 86% of the time.

7.3 An Example for Acyclic Graphs

Figure 9 shows the implementation of a non-uniform, near-perfect reconstruction filterbank. The lowpass filters retain 2/3 of the spectrum while the highpass filters retain 1/3 (instead of the customary 1/2,1/2 for the octave QMF). Rate changes in the graph are annotated wherever the

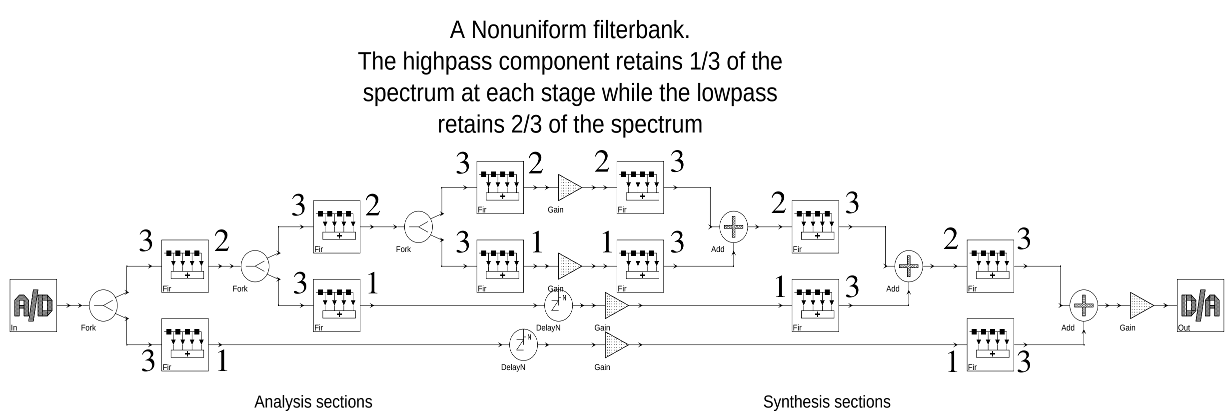

Figure 9. Non-uniform filterbank example.

number produced or consumed is different from unity. The gain actors on the limbs between the analysis and synthesis sections enable the use of the filterbank as a simple 4-channel equalizer. The repetitions vector of this graph is given by

$q = [27, 27, 9, 9, 18, 6, 6, 9, 12, 6, 9, 4, 4, 6, 8, 4, 4, 4, 12, 6, 6, 9, 18, 9, 27, 27, 27]$. The heuristic, when run on this graph, obtains a schedule with a buffering cost of 100; the worst case naive schedule (for any topological sort) would have a buffering cost of 438. The best schedule obtained by examining 30 random topological sorts had a cost of 125 for this graph and the best schedule obtained by examining 60 random topological sorts had a cost of 120. The best of the various PGAN heuristics found a schedule of cost 117. This example clearly shows that, in practice, the performance of the new heuristic is likely to be better than that suggested by its performance on random graphs.

8 Related Work

In [2], Ade, Lauwereins, and Peperstraete develop upper bounds on the minimum buffer memory requirement for a number of restricted classes of SDF graphs. The graphs considered each consist of a chain-structured subgraph, together with zero or more edges directed between distinct actors in the chain-structured subgraph. Ade et al. present an efficiently computable upper bound on the minimum buffer memory required over all valid schedules, and they present simulation data that demonstrates that on average, the computed bounds are close to the corresponding actual minima. 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 [14], Lauwereins, Wauters, Ade, and Peperstraete present a generalization of SDF, called *cyclo-static dataflow*. In cyclo-static dataflow, the number of tokens produced and consumed by an actor can vary between firings as long as the variations form a certain type of periodic pattern. For example, consider an actor that routes data received from a single input to each of two outputs in alternation. In cyclo-static dataflow, this operation can be represented as an actor that consumes one token on its input edge, and produces tokens according to the periodic pattern $1, 0, 1, 0, \ldots$ (one token produced on the first invocation, none on the second, one on the third, and so on) on one output edge, and according to the complementary pattern $0, 1, 0, 1, \ldots$ on the other output edge. A cyclo-static dataflow graph can be compiled as a cyclic pattern of pure SDF graphs, and static periodic schedules can be constructed in this manner. 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 get schedules that are as compact as single appearance schedules for pure SDF graphs but have lower buffering requirements than that arising from 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 [11]. 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 [11] address the problem of selecting a schedule that minimizes buffering cost from among the set of *rate-optimal* schedules. This problem does not take code space constraints into account. Instead, it focuses on another dimension of scheduling that the techniques of our paper do not consider — parallel processing.

9 Conclusion

In this paper, we have presented algorithms for constructing schedules that minimize buffer usage from among the schedules that minimize program memory usage (called buffer-optimal single appearance schedules) for programs expressed as SDF graphs. We defined the class of R-schedules and showed that there is always an R-schedule that is a buffer-optimal single appearance schedule. It is possible to construct buffer-optimal R-schedules for the class of well-ordered SDF graphs by using a dynamic programming algorithm. We showed the efficacy and the usefulness of our algorithm on a practical example. We also showed that the problem of determining buffer-optimal single appearance schedules appears to be a much more complicated problem for general acyclic SDF graphs. Instead, we have presented heuristics that perform well in practice.

There are still many open problems left to be solved in this area of compiler design for SDF graphs. The complexity of the problem of determining buffer-optimal single appearance schedules for acyclic graphs is unknown, even though we believe it to be NP-complete. If the problem is NP-complete, it would be interesting to see if better heuristics can be developed. In particular, it would be interesting to see what effect a better heuristic for finding minimum weight legal cuts into bounded sets would have on the quality of the schedules. Recall that the very idea of using minimum cuts is a heuristic; hence, even if we were able to determine the optimal legal minimum cuts (which is unlikely since that problem appears to be NP-complete as well), we wouldn't always produce buffer-optimal single appearance schedules. However, it might improve the quality of the schedules somewhat. A reduction of this problem (determining buffer-optimal single appearance schedules for acyclic graphs) to integer linear programming (ILP) would be useful as it would allow evaluations of heuristics against the optimal answer. We also gave some reasons why the problem of constructing buffer-optimal single appearance schedules becomes even more complicated for arbitrary SDF graphs. Heuristic solutions for this problem are a topic for further study. Finally, techniques for systematically trading program compactness for buffer usage are also a topic for further study.

Acknowledgment

Thomas M. Parks, a graduate student at the University of California at Berkeley, conceived, designed, and implemented the rate-change system of Figure 5(b).

References

[1] W. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie, "On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations," *IEEE Transactions on Computers*, vol.C-30, (no.5):341-56, May, 1981.

[2] 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.

[3] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

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

[5] S. S. Bhattacharyya and E. A. Lee, "Looped Schedules for Dataflow Descriptions of Multirate Signal Processing Algorithms," *Journal of Formal Methods in System Design,* to appear, December 1994.

[6] S. S. Bhattacharyya and E. A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," *Journal of VLSI Signal Processing*, vol.6, (no.3):271-88, December, 1993.

[7] 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, p. 1245-8 vol.2, April, 1991.

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

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

[10] S. S. Godbole, "On Efficient Computation of Matrix Chain Products," *IEEE Transactions on Computers*, vol.C22, (no.9):864-7, September, 1973.

[11] 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*, p. 75-86, San Francisco, August, 1994.

[12] W. H. Ho, E. A. Lee, and D. G. Messerschmitt, "High Level Dataflow Programming for Digital Signal Processing," *VLSI Signal Processing III*, IEEE Press, 1988.

[13] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, vol.49, (no.2):291-308, February 1970.

[14] 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.

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

[16] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. S. Bhattacharyya, "Gabriel: A Design Environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol.37, (no.11):1751-62, November, 1989.

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

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

[19] 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 1995.

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

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

[22] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.

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