UCB/ERL Technical Report, Memorandum No. **UCB/ERL M95/43**, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720

Renesting Single Appearance Schedules to Minimize Buffer Memory

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

April 1, 1995

ABSTRACT

Minimizing memory requirements for program and data are critical objectives when synthesizing software for embedded DSP applications. In prior work, it has been demonstrated that for graphical DSP programs based on the widely-used synchronous dataflow model, an important class of minimum code size implementations can be viewed as parenthesizations of lexical orderings of the computational blocks. Such a parenthesization corresponds to the hierarchy of loops in the software implementation. In this paper, we present a dynamic programming technique for constructing a parenthesization that minimizes data memory cost from a given lexical ordering of a synchronous dataflow graph. For graphs that do not contain delays on the edges, this technique always constructs a parenthesization that has minimum data memory cost from among all parenthesizations for the given lexical ordering. When delays are present, the technique may make refinements to the lexical ordering while it is computing the parenthesization, and the data memory cost of the result is guaranteed to be less than or equal to the data memory cost of all valid parenthesizations for the initial (input) lexical ordering.

Thus, our dynamic programming technique can be used to post-optimize the output for any algorithm that schedules SDF graphs into minimum code size loop hierarchies. On several practical examples, we demonstrate that significant improvement can be gained by such post-optimization when applied to two scheduling techniques that have been developed earlier. That is, the result of each scheduling algorithm combined with the post-optimization often requires significantly less data memory than the result of the scheduling algorithm without post-optimization. We also present an adaptation of our dynamic programming technique for post-optimizing an arbitrary (not necessarily minimum code size) schedule to optimally reduce the code size.

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

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

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

1. Background

This paper develops a dynamic programming technique for reducing memory requirements when synthesizing software from graphical DSP programs that are based on the synchronous dataflow (SDF) model [10]. Numerous DSP design environments, including a number of commercial tools, support SDF or closely related models [9, 14, 13, 15, 16]. In SDF[1], a program is represented by a directed graph in which each vertex (**actor**) represents a computation, an edge specifies a FIFO communication channel, and each actor produces (consumes) a fixed number of data values (**tokens**) onto (from) each output (input) edge per invocation.

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

The first step in compiling an SDF graph is to construct a **valid schedule**, which is a sequence of actor invocations that invokes each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each edge. For each actor in the valid schedule, a corresponding code block, obtained from a library of predefined actors, is instantiated. The resulting sequence of code blocks is encapsulated within an infinite loop to generate a software implementation of the SDF graph [8].

In [10], efficient algorithms are presented to determine whether or not a given SDF graph has a valid schedule, and to determine the minimum number of times that each actor must be fired in a valid schedule. We represent these minimum numbers of firings by a vector $\mathbf{q}_G$, indexed by

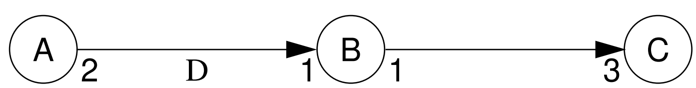

Figure 1. A simple SDF graph.

1. This should not be confused with the use of “synchronous” in synchronous languages [1].

the actors in G. We refer to $\mathbf{q}_G$ as the **repetitions vector** of G. Given an edge e in G, we define

$$TNSE(e) \equiv \mathbf{q}_G(src(e)) \times prod(e) = \mathbf{q}_G(snk(e)) \times cons(e) . \tag{1}$$

Thus, $TNSE(e)$ is the total number of tokens produced onto (consumed from) e in one period of a valid schedule. The equality of the two products in (1) follows from the definition of the repetitions vector. For Fig. 1, $\mathbf{q}(A, B, C) = (3, 6, 2)$, and $TNSE((A, B)) = TNSE((B, C)) = 6$.

One valid schedule for Fig. 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. 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**. A **looped schedule** is a finite sequence $V_1V_2\ldots V_k$, where each V_i is either an actor or a schedule loop.

A more compact valid schedule for Fig. 1 is $(3A)(2(3B)C)$. We call this schedule a **single appearance schedule** since it contains only one lexical appearance of each actor. To a good first approximation, any valid single appearance schedule gives the minimum code size cost for in-line code generation. This approximation neglects second order affects such as loop overhead and the efficiency of data transfers between actors. Systematic synthesis of single appearance schedules is described in [3].

Given an SDF graph G, a valid schedule S, and an edge e in G, $max\_tokens(e, S)$ denotes the maximum number of tokens that are queued on e during an execution of S. For example, if for Fig. 1, $S_1 = (3A)(6B)(2C)$ and $S_2 = (3A(2B))(2C)$, then $max\_tokens((A, B), S_1) = 7$ and $max\_tokens((A, B), S_2) = 3$. We define the **buffer memory requirement** of a schedule S by $buffer\_memory(S) \equiv \sum max\_tokens(e, S)$, where the summation is over all edges in G. Thus, $buffer\_memory(S_1) = 7 + 6 = 13$, and $buffer\_memory(S_2) = 3 + 6 = 9$.

The **lexical ordering** of a single appearance schedule S, denoted $lexorder(S)$, is the sequence of actors $(A_1, A_2, \ldots, A_n)$ such that each A_i is preceded lexically by $A_1, A_2, \ldots, A_{i-1}$. Thus, $lexorder((2(3B)(5C))(7A)) = (B, C, A)$. Given an SDF graph, an **order-optimal schedule** is a single appearance schedule that has minimum buffer memory requirement from among the valid single appearance schedules that have a given lexical ordering

In the model of buffering implied by our “buffer memory requirement” measure, each buffer is mapped to an independent contiguous block of memory. Although perfectly valid target programs can be generated without this restriction, it can be shown that having a separate buffer on each edge is advantageous because it permits full exploitation of the memory savings attainable from nested loops, and it accommodates delays without complication [11]. 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 [11]. Combining the analysis and techniques that we develop in this paper with methods for sharing storage among multiple buffers is a useful direction for further study.

In this paper we present a dynamic programming technique for post-processing a single appearance schedule with the goal of generating a modified single appearance schedule that has a significantly lower buffer memory requirement. The technique is an extension of the algorithm developed in [12] for constructing single appearance schedules for chain-structured SDF graphs, and a basic version of this technique, called **Dynamic Programming Post Optimization (DPPO)**, that applies to delayless, acyclic graphs was outlined in [12]. In this paper, we give a detailed specification of DPPO, we generalize DPPO to handle delays and arbitrary topologies, and we show that while DPPO always preserves the lexical ordering of the input schedule, our generalization of DPPO, called **GDPPO**, may change the lexical ordering (for a graph that has delays) and thus, that it may compute a schedule that has lower buffer memory requirement than all single appearance schedules that have the same lexical ordering as the input schedule. Since the introduction of the basic version of DPPO in [12], we have also developed and implemented two heuristics, called APGAN and RPMC, for efficiently constructing single appearance schedules that have low buffer memory requirement [2, 11] for acyclic graphs. In this paper, we present experimental results that demonstrate the ability of GDPPO to significantly improve the schedules constructed by APGAN and RPMC for practical SDF systems.

A distinguishing characteristic of this work, as compared to other work on memory optimizations for SDF and related models, is that it focuses on the joint reduction of both code and data memory requirements for uniprocessor implementations. In contrast, Govindarajan, Gao, and Desai have developed scheduling algorithms to minimize data memory requirements, without considering code size, in a parallel processing context [7]. Also, Lauwereins, Wauters, Ade, and Peperstraete have proposed an extension to SDF called *cyclostatic dataflow*, which allows an important class of applications to be described in such a way that significantly less token traffic is required than the buffer activity that would result from the corresponding pure-SDF implementations [4]. However, the impact of this model on code size has not been explored in depth, nor have code size optimizations been developed that exploit the unique features of cyclostatic dataflow.

In [17], Ritz, Willems, and Meyr present techniques for minimizing the memory requirements of a class of single appearance schedules that minimize the rate of context-switches between actors. These schedules are called *flat* schedules since they do not apply any nested loops. As implied above, the memory requirements associated with flat schedules are often significantly larger than the nested-loop schedules that we discuss in this paper, even if techniques are applied to share memory among multiple buffers for the flat schedules [11]. For example for the mobile satellite receiver example discussed in [17], an *optimum* single appearance schedule under the criterion of Ritz, Willem and Meyr requires 1920 units of memory. In contrast, we have found that the minimum achievable buffer memory requirement over all (not necessarily flat) single appearance schedules is 1542 for this example, and this minimum value is achieved by the APGAN heuristic, which is discussed in [2].

When there is enough memory to accommodate the minimum context-switch schedules of [17], it is possible that these schedules will result in somewhat higher throughput than the schedules discussed in this paper, although the difference in context-switch overhead can often be significantly mitigated by the techniques described in [15]. However, when memory constraints are severe, the techniques discussed in this paper are superior.

The basic structure of the dynamic programming techniques developed in this paper and in [12] was inspired by Godbole's dynamic programming algorithm for matrix-chain multiplication, which is presented in [6].

2. Dynamic Programming Post Optimization

Suppose that G is a connected, delayless, acyclic 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 $(G, lexorder(S))$. If G contains at least two actors, then it can be shown [2] that there exists 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

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

$$E_s = \{e \mid (src(e) \in \{A_1, A_2, \ldots, A_x\} \textbf{ and } snk(e) \in \{A_{x+1}, A_{x+2}, \ldots, A_n\})\}$$ 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 are 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 [11].

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

$$b[i, k] + b[k+1, j] + c_{i,j}[k], \tag{2}$$

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 [2][1]

$$c_{i,j}[k] = \frac{\sum_{e \in E_s} TNSE(e)}{gcd\left(\{\mathbf{q}_G(A_x) \mid (i \le x \le j)\}\right)}, \tag{3}$$

where

$$E_s = \{e \mid \left(src(e) \in \{A_i, A_{i+1}, \ldots, A_k\} \textbf{ and } snk(e) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\}\right)\} \tag{4}$$

is the set of edges that cross the split.

3. Extension to Arbitrary Topologies

DPPO can be extended to efficiently 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 $lexorder(S)$, 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

1. The symbol *gcd* denotes the greatest common divisor.

$S = (6A)\,(5\,(2C)\,(3B))$ for Fig. 2. 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 (4) no longer generally gives all of the edges that cross the parenthesization split. We must also examine the set of *back edges*

$$E_b = \{e \,|\, (snk(e) \in \{A_i, A_{i+1}, \ldots, A_k\} \textbf{ and } src(e) \in \{A_{k+1}, A_{k+2}, \ldots, A_j\})\} . \tag{5}$$

Each $e \in E_b$ must satisfy

$$delay(e) \geq \frac{TNSE(e)}{gcd(\{\mathbf{q}_G(A_x) \,|\, (i \leq x \leq j)\})}, \tag{6}$$

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 (2) for the given split gets replaced with

$$b[i, k] + b[k+1, j] + \frac{\sum_{e \in E_s} TNSE(e)}{gcd(\{\mathbf{q}_G(A_x) \,|\, (i \leq x \leq j)\})} + \sum_{e \in E_s} delay(e) + \sum_{e \in E_b} delay(e) . \tag{7}$$

Expression (7) gives the cost of spliting the subsequence $(A_i, A_{i+1}, \ldots, A_j)$ between A_k

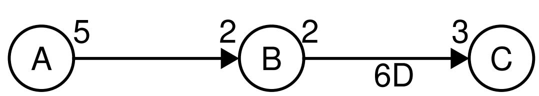

Figure 2. An SDF graph used to illustrate GDPPO applied to SDF graphs that have nonzero delay on one or more edges. Here $\mathbf{q}(A, B, C) = (6, 15, 10)$.

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 (6) is satisfied *for all* "forward edges" $e \in E_s$, it may be advantageous to interchange the lexical 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

$$b[i,k] + b[k+1,j] + \frac{\sum_{e \in E_b} TNSE(e)}{gcd\left(\{ \mathbf{q}_G(A_x) \mid (i \le x \le j) \}\right)} + \sum_{e \in E_b} delay(e) + \sum_{e \in E_s} delay(e) \quad (8)$$

is less than the *forward split cost* computed from (7) — that is, whenever

$$\sum_{e \in E_b} TNSE(e) < \sum_{e \in E_s} TNSE(e) . \quad (9)$$

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.

Fig. 3 shows an example where multiple applications of GDPPO are 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 (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 (8), 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 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.

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

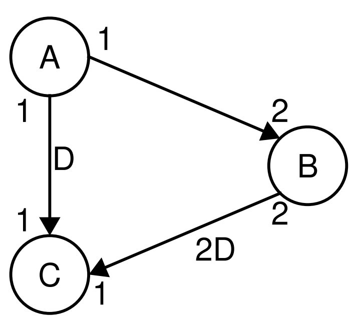

Figure 3. An illustration of iterative application of GDPPO.

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 (6) 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 (6). 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 (6), 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 $O\left(n_v^3\right)$[1] 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.

4. Experimental Results

In [2], two heuristics, called APGAN and RPMC, are described for constructing single appearance schedules that minimize the buffer memory requirement. APGAN is a bottom-up clustering technique that has been found to perform well for graphs that have regular topological structures and sample-rate changes. RPMC is a top-down technique based on a generalized mini-

1. A function $f(x)$ is $O(g(x))$ if for sufficiently large x, $f(x)$ is bounded above by a positive real multiple of $g(x)$.

mum-cut operation that usually does not perform as well as APGAN on regular graphs, but often significantly outperforms APGAN on graphs that have irregular structure. Thus, APGAN and RPMC are complementary, — when one of the techniques fails to construct a good schedule, the other can be expected to be find one [2].

Fig. 4 shows a screendump from the Ptolemy prototyping environment [5] that contains an SDF description of a four channel, nonuniform filter bank. The single appearance schedule obtained by APGAN on this system is

$$(3\,(3k\,(3ab)\,cd\,(2e)\,h)\;(2nugj)\;(2f\,(2i)\,)\,)\;(4l\,(2o)\,q)\;(4mpr)\;(3\,(2\,(2s)\,t)\;(3\,(2w)\,xv\,(3yzA)\,)\,)\ ,$$

and the resulting buffer memory requirement is 153. After post-processing the schedule with GDPPO, we obtain the schedule

$$(3\,(3\,(3ab)\,cd\,(2e)\,kh)\;(2gnufj)\,)\;(4\,(3i)\,l\,(2o)\,qmpr\,(3s)\,)\;(3\,(2t)\;(3\,(2w)\,xv\,(3yzA)\,)\,)\ ,$$

which has a buffer memory requirement of 137 — a 10.5% improvement. Notice that in this example, GDPPO has changed the lexical ordering, and thus, one or more reverse splits were found to be beneficial.

The schedule returned by RPMC has a buffer memory requirement of 131, which is lower than that obtained by the combination of APGAN and GDPPO. However, GDPPO is able to improve the schedule obtained by RPMC even further. The result computed by GDPPO when applied to the schedule derived by RPMC is

$$(3\,(3\,(3ab)\,dc)\;(2\,(3e)\,fgnj)\;(3kh)\,)\;(4\,(3i)\,mpl\,(2o)\,qr\,(3s)\,)\;(3\,(2tu\,(3w)\,)\;(3xv\,(3yzA)\,)\,)\ ,$$

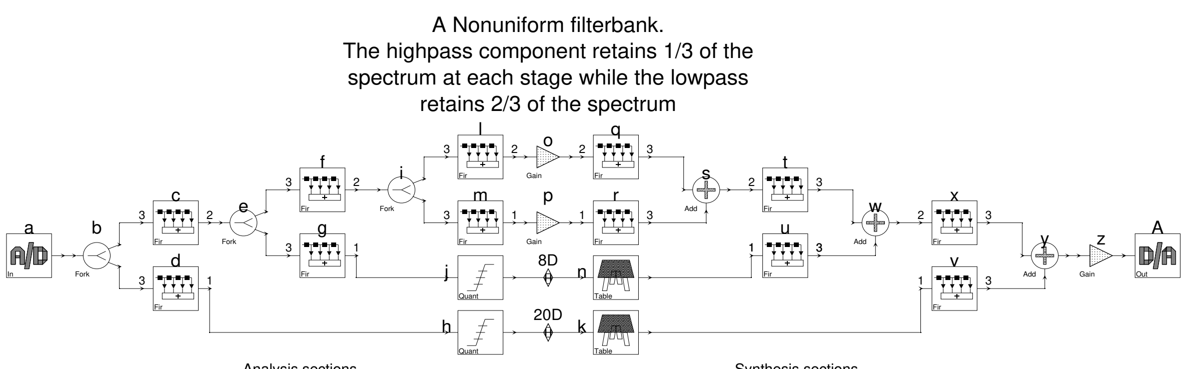

Figure 4. Non-uniform filterbank example. The SDF parameters have been shown wherever they are different from unity.

which has a buffer memory requirement of 128.

Table 1 shows the results of applying GDPPO to the schedules generated by APGAN and RPMC on several practical SDF systems. The columns labeled "% Impr." show the percentage of buffer memory reduction obtained by GDPPO. The QMF tree filter banks fall into a class of graphs for which APGAN is guaranteed to produce optimal results [2], and thus there is no room for GDPPO to produce improvement when APGAN is applied to these two examples. Overall, we see that GDPPO produces an improvement in 11 out of the 14 heuristic/application combinations. A "significant" (greater than 5%) improvement is obtained in 9 of the 14 combinations; the mean improvement over all 14 combinations is 9.9%; and from the CD-DAT and DAT-CD examples, we see that it is possible to obtain very large reductions in the buffer memory requirement with GDPPO.

Table 1. Performance of GDPPO on several practical SDF systems.

Application	APGAN only	APGAN +DPPO	% Impr.	RPMC only	RPMC + DPPO	% Impr.
Nonuniform filter bank (1/3, 2/3 splits, 4 channels)	153	137	10.5	131	128	2.34
Nonuniform filter bank (1/3, 2/3 splits, 6 channels)	856	756	11.7	690	589	14.6
QMF tree filter bank (8 channels)	78	78	0	92	87	5.43
QMF tree filter bank (6 channels)	166	166	0	218	200	8.26
Two-stage fractional decimation system	140	119	15.0	133	133	0
CD-DAT sample rate conversion	396	382	3.54	535	400	25.2
DAT-CD sample rate conversion	205	182	11.2	275	191	30.5

5. Adaptation to Minimize Code Size for Arbitrary Schedules

We have applied the basic concept behind DPPO to derive an algorithm that computes an optimally compact looped schedule (minimum code size) for an arbitrary sequence of actor firings. For example, consider the SDF graph in Fig. 5, and suppose that we are given the valid firing sequence[1] $\sigma = ABABCBC$ (this firing sequence minimizes the buffer memory requirement over all valid schedules). If the code size cost (number of program memory words required) for a code block for A is greater than the code size cost for C, then the optimally compact looped schedule for σ is $(2AB)CBC$, whereas $ABA(2BC)$ is optimal if C has a greater code size cost than A.

To understand how dynamic programming can be used to compute an optimally compact loop structure, suppose that $\sigma = A_1 A_2 \ldots A_n$ is the given firing sequence, and let $\sigma' = A_i A_{i+1} \ldots A_j$ be any subsequence of σ ($1 \le i < j \le n$). If the optimal loop structures for all $(j - i)$-length subsequences of σ are available, then we determine the optimal loop structure for σ' by first computing $C_k \equiv Z_{i,k} + Z_{k+1,j}$ for $k \in \{i, i+1, \ldots, j-1\}$, where $Z_{x,y}$ denotes the minimum code size cost for the subsequence $A_x, A_{x+1}, \ldots, A_y$. The value of k that minimizes C_k gives an optimum point at which to "split" the subsequence *if* $A_i A_{i+1} \ldots A_j$ *are not to be executed through a single loop*.

To compute the minimum cost attainable for σ' if $A_i A_{i+1} \ldots A_j$ are to be executed through a single loop, we first determine whether or not $\sigma' = A_i A_i \ldots A_i$. If this holds, then σ' can be executed through a single loop $((j - i + 1)A_i)$, and the code size cost is taken to be the code size cost of A_i plus the code size overhead C_L of a loop. If $\sigma' \ne A_i A_i \ldots A_i$, we determine whether or not $\sigma' = A_i A_{i+1} A_i A_{i+1} \ldots A_i A_{i+1}$, and if so, then σ' can be implemented as $(((j - i + 1)/2) A_i A_{i+1})$, and the code size cost is taken to be the sum of C_L and the costs of

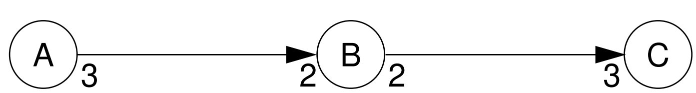

Figure 5. An SDF graph used to illustrate the problem of finding an optimally compact loop structure for an arbitrary firing sequence.

1. By a *firing sequence*, we simply mean a schedule that contains no loops.

A_i and A_{i+1}. Next, if $\sigma' \neq A_iA_i\ldots A_i$ and $\sigma' \neq A_iA_{i+1}A_iA_{i+1}\ldots A_iA_{i+1}$, then we determine whether or not $\sigma' = A_iA_{i+1}A_{i+2}\ldots A_iA_{i+1}A_{i+2}$. If this holds then σ' can be implemented as $(((j-i+1)/3)\,S_L(A_iA_{i+1}A_{i+2}))$, where $S_L(A_iA_{i+1}A_{i+2})$ is an optimal loop structure for $A_iA_{i+1}A_{i+2}$. It is easily seen that an optimal loop structure L for executing $A_iA_{i+1}\ldots A_j$ through a single loop can be determined (if one exists) by iterating this procedure $\lfloor (j-i+1)/2 \rfloor$ times, where $\lfloor * \rfloor$ denotes the *floor* operator. If one or more loop structures exist for executing $A_iA_{i+1}\ldots A_j$ through a single loop, then the code size cost of the optimal loop structure L is compared to the minimum value of C_k, $i \leq k < j$, to determine an optimal loop structure for $A_iA_{i+1}\ldots A_j$; otherwise the optimal loop structure for $A_iA_{i+1}\ldots A_j$ is taken to be that corresponding to the minimum value of C_k.

The time complexity of this technique for finding an optimal loop structure for the subsequence $A_iA_{i+1}\ldots A_j$, given optimal looping structures for all $(j-i)$-length subsequences, is $O\left((j-i+1)^2\right)$, and time complexity of the overall algorithm is $O\left(n^4\right)$, where n is the number of firings in the input firing sequence. Thus, the problem of determining an optimal looping structure is of polynomial complexity when the size of a problem instance is taken to be the number of firings in the given firing sequence. However, it should be noted that there is no polynomial function $P(m)$ such that the number of firings in a valid schedule (defined as the sum of the entries in the repetitions vector) is guaranteed to be less than $P(m)$ for an arbitrary m-actor SDF graph. Thus, unlike DPPO and GDPPO, the algorithm developed in this section is not of polynomial complexity in the size of the given SDF graph.

6. Conclusion

This paper has developed a dynamic programming post optimization, called GDPPO, for reducing the data memory cost of software implementations of synchronous dataflow graphs that minimize code size for in-line code generation. We have presented data on several practical examples that shows that GDPPO can produce significant improvements when appended to either of two existing heuristics, APGAN and RPMC, for constructing minimum code size schedules. Since these two heuristics are fundamentally different in structure — one is based on top-down partitioning, and the other is based on bottom-up clustering — we expect that GDPPO can yield similar benefits when used to improve the performance of alternative scheduling algorithms. We have also presented an adaptation of GDPPO to minimize the code size of an arbitrary synchronous dataflow schedule.

References

[1] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, September, 1991.

[2] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Two Complementary Heuristics for Translating Graphical DSP Programs into Minimum Memory Implementations,* Memorandum No. UCB/ERL M95/3, Electronics Research Laboratory, University of California at Berkeley, January, 1995, WWW URL: http://ptolemy.eecs.berkeley.edu/papers/PganRpmcDppo.

[3] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, "A Scheduling Framework for Minimizing Memory Requirements of Multirate DSP Systems Represented as Dataflow Graphs," *VLSI Signal Processing VI*, IEEE Special Publications, 1993.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Static Scheduling of Multi-Rate and Cyclo-Static DSP-Applications," *VLSI Signal Processing VII*, IEEE Press, 1994.

[5] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, April, 1994.

[6] S. S. Godbole, "On Efficient Computation of Matrix Chain Products," *IEEE Transactions on Computers*, September, 1973.

[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*, August, 1994.

[8] 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.

[9] 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.

[10] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs

for Digital Signal Processing," *IEEE Transactions on Computers*, February, 1987.

[11] 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, November, 1994, WWW URL: http//ptolemy.eecs.berkeley.edu/papers/jointCodeDataMinimize.

[12] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, "Minimizing Memory Requirements for Chain-Structured Synchronous Dataflow Programs," *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, April, 1994.

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

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

[15] D. B. Powell, E. A. Lee, and W. C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, March, 1992

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

[17] S. Ritz, M. Willems, and H. Meyr, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis," *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, to appear, 1995.